Université Paris Dauphine
IUP Génie Mathématique et Informatique
IUP1 – Année 2001-2002
Algorithmique et structures de données

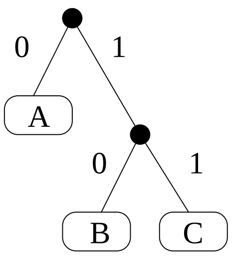
Devoir à rendre le 14 janvier 2002
Compression de données par la méthode de Huffman

Si vous n'avez pas rendu le devoir sur les tableaux extensibles, il est *impératif* *de rendre ce devoir. Si vous décidez de rendre ce devoir et que vous avez également rendu le précédent devoir dans les temps, seule la meilleur des deux notes sera comptabilisée.*
Comme d'habitude, chaque question sera accompagnée du listing du code C correspondant, ainsi que du listing du résultat de l'exécution des programmes demandés.

Le but de ce devoir est de concevoir un *compresseur de données* utilisant la méthode de Huffman.

Dans un fichier texte, chaque caractère est codé par un *octet* (type C `char`), donc par 8 *bits* (chiffres binaires). Or, dans la pratique, on se rend compte que, dans un texte, certaines lettres sont utilisées plus souvent que d'autres (le *e* est souvent utilisé en français tandis que le *w* est très rare). L'idée du codage de Huffman est d'utiliser des codes de taille variable : moins de bits pour les caractères souvent utilisés, et plus de bits pour les autres, de manière à diminuer le nombre total de bits nécessaires pour coder l'ensemble du texte.

Un *codage* est représenté par un *arbre binaire* dont les arêtes gauches et droites sont étiquetées par, respectivement, 0 et 1 (un bit), et les feuilles par un caractère. Les nœuds internes ne sont pas étiquetés. Étant donné un caractère, le code binaire compressé associé est la suite des étiquettes des arêtes rencontrées sur le chemin menant de la racine de l'arbre à la feuille contenant ce caractère. On obtient le *texte compressé* en concaténant les codes compressés correspondant aux caractères du texte décompressé.



Exemple : dans l'exemple de gauche, les caractères A, B et C sont codés respectivement par 0, 10 et 11, ce qui est avantageux si A apparaît plus souvent que B et C. Le mot “BAAB” est donc représenté par la séquence de bits 100010.

De plus, le codage de Huffman est capable, étant donné la fréquence d'apparition de chaque caractère, de construire automatiquement un arbre correspondant à un *codage optimal.*

Question 1 : Décodage.

Un nœud de l'arbre de Huffman est codé par une cellule de type `node_t` contenant une étiquette `car` (valable uniquement si le nœud est une feuille) ainsi qu'un pointeur vers la cellule du fils gauche et un pointeur vers la cellule du fils droit (ou `NULL` si le nœud est une feuille).
Donnez la description en langage C du type `node_t`.

Pour simplifier, le texte compressée est stocké dans une chaîne terminée par un `'\0'`, et contenant des caractères `'0'` et `'1'` (1 bit compressé occupe un caractère).

Programmez les fonctions suivantes, permettant le décodage :
- `node_t* huf_leaf(char c)`
  qui renvoie un arbre réduit à une feuille étiquetée par le caractère `c` ;
- `node_t* huf_node(node_t* zero, node_t *one)`
  qui renvoie un arbre ayant pour fils gauche le sous-arbre `zero` et pour fils droit `one` ;

– `char huf_get(node_t* t, char* code, int* n)`
  qui décode et renvoie le premier caractère de la chaîne compressée `code` ; il suffit pour cela de parcourir l'arbre `t` selon la séquence de caractères `'0'` et `'1'` de `code` de la racine à une feuille ; le nombre de caractères de `code` utilisés pour construire ce caractère décompressé est renvoyé dans `*n` (ainsi `code+*n` pointe sur le début du code du prochain caractère à décompresser) ;
– `void huf_decode(node_t* t,char *code)`
  qui affiche sur l'écran la suite des caractères codés par la chaîne compressée `code` (utilisez la fonction `huf_get`) ;
– `void huf_free(node_t* t)`
  qui libère l'arbre `t` entièrement (récursivement).

Faites un petit programme de test qui construit l'arbre de l'exemple ci-dessus et affiche le décodage de la chaîne `"010011"`.

Question 2 : Encodage.

Étant donné un arbre `huf_t`, il est difficile d'obtenir le code compressé correspondant à un caractère donné. Pour simplifier le processus, on décomposera l'encodage en deux fonctions :
– `char** huf_table(node_t* t)` renvoie un tableau de 256 chaînes de caractères ; à l'indice `i`, ce tableau contient le code compressé correspondant au caractère décompressé `(char)i` ;
– `char* huf_code(char** table, char* texte, int n)` utilise la table `table` générée par la fonction précédente pour retourner le texte compressé correspondant au texte décompressé `texte` de `n` octets de long.

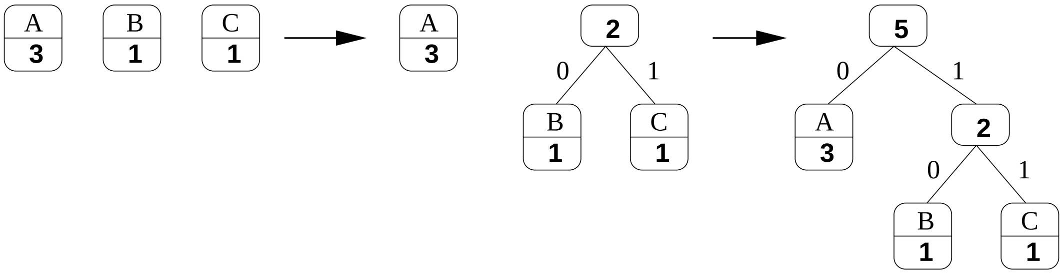
Faites un petit programme de test qui construit l'arbre de l'exemple et affiche le codage de la chaîne `"CABAAAB"`.

Question 3 : Construction de l'arbre de Huffman.

Pour construire l'arbre de codage optimal, il faut d'abord extraire des informations statistiques sur le texte à compresser.
Programmez la fonction `int* huf_stat(char* a, int n)` qui renvoie un tableau indiquant, pour chaque caractère, le nombre d'occurrences de ce caractère dans le texte pointé par `a` et de longueur `n`.

L'algorithme de construction d'arbre optimal fonctionne par fusions successives de sous-arbres ; il manipule donc une liste d'arbres. Au départ, la liste contient pour chaque caractère présent au moins une fois dans le texte un arbre réduit à une feuille ; dans cette feuille est stocké le nombre d'occurrences du caractère. À chaque étape de fusion, on choisit les deux arbres `a1` et `a2` dont les racines ont le plus faible nombre d'occurrences ; ces deux arbres sont remplacées par un arbre unique `a` qui a pour sous-arbre droit `a1` et pour sous-arbre gauche `a2`, et pour nombre d'occurrences, la somme du nombre d'occurrences des racines de `a1` et `a2`. On s'arrête quand il ne reste plus qu'un seul arbre dans la liste ; c'est l'arbre de Huffman recherché.

Exemple :



Modifiez le type `node_t` et les fonctions `huf_leaf` et `huf_node` pour qu'elles tiennent compte du nombre d'occurrences des lettres. `huf_leaf` a maintenant un argument supplémentaire, mais le prototype de `huf_node` reste inchangé.

Programmez la fonction `huf_t* huf_build(int *stat)` qui, à partir des informations statistiques `stat` calculées par `huf_stat`, construit l'arbre de Huffman optimal. Cette fonction utilisera les fonctions `huf_leaf` et `huf_node` modifiées. Quelle est la complexité de cette fonction ?

Faites un petit programme de test qui construit l'arbre et la chaîne compressée correspondant au texte décompressé `"BLABLA BLIBLI BLU !"`. Dessinez sur votre copie l'arbre de Huffman calculé.