316 Data Structures

Assignment 3 - Huffman Coding

(Due: Oct. 17, 2012)

Huffman coding is a scheme that assigns variable-length bit-codes to characters, such that the lengths of the codes depend on the frequencies of the characters in a typical message. As a result, encoded messages take less space (as compared to fixed-length encoding) since the letters that appear more frequently are assigned shorter codes. This is performed by first building a Huffman coding tree based on a given set of frequencies. From the tree, bit-codes for each character are determined and then used to encode a message. The tree is also used to decode an encoded message as it provides a way to determine which bit sequences translate back to a character.

Assignment:

For this assignment, you will write three programs:

1) ***buildtree*.** Given a file that contains the frequency distribution for all 26 letters of the alphabet plus four special characters (space, period, new-line, and end-of-message), *buildtree* will generate two files: a huffman coding tree file and a file containing bit-code assignments.
2) ***encode.*** Given the bit-code file generated by *buildtree*, and a text file containing a message, *encode* will produce a binary file that contains the encoded message.
3) ***decode.*** Given the huffman coding tree file generated by *buildtree*, and an encoded message generated by *encode*, *decode* will reproduce the original message and output that message to a text file.

You may assume that the messages are written in lower-case letters. The frequency file is a 30-line text file, where each line contains a letter (or a special character) followed by a space and a positive integer (string of digits). For the purposes of this frequency file, the special characters have the following codes: '-' for space, '.' for period, '!' for new line, and '+' for end-of-message.

The tree file is a sequential representation of the tree built in *buildtree* to be read and recreated in *decode*.

The bit-code file is a 30-line text file. Each line in this file contains a letter (or a special character) followed by a space and then a string of 1's and 0's. The order of the characters in this file should be as follows: 'a' through 'z', space, period, new-line, end-of-message.

The message file consists of lines containing lower case letters, actual spaces (not '-'), and periods. When interpreting the message file, a new-line character is implied at the end of each line, and an end-of-message character is implied at the end of the file.

The encoded file is a stream of bytes that encodes the corresponding bits. In the event that the resulting number of bits is not a multiple of 8, add enough bits (with value 0) to form the final byte.

When running decode, note that the end-of-message character is important since it signifies when to stop decoding, particularly if there are remaining bits in the last byte of the encoded message.

Sample files are presented later in this document.

Data Structures:

You will use a dynamic node implementation of a binary tree to represent the Huffman coding tree. Note that internal nodes and leaf nodes in the tree contain different data. For simplicity, use the same tree data structure for *buildtree* and *decode*.

Building a Huffman tree involves iteratively combining two minimum-valued nodes or sub- trees. In case of ties for minimum values, disambiguate using the following rules:

• Multiple-node subtrees take precedence over single nodes.

• The most recently-created subtree takes precedence over other subtrees.

• A single node that occurs earlier in an alphabetized listing of the letters takes precedence over other single nodes. Space, period, new-line, and end-of-message (in this order) come after 'z' in this alphabetized listing.

The simplest way to implement the rules given above is to maintain a list of trees sorted by weight. The list begins as a list of single nodes ordered by weight, using their alphabetical order if the weights are tied. You will then repeatedly remove the first two items in the list and combine them to form a subtree. The resulting subtree is re-inserted in the list according to its resulting weight. In case there are elements in the list with the same weight as the subtree, insert the subtree right before the first item that has that weight.

Sequential representations of a binary tree are discussed in the class. More specifically, list node values in the order they would be visited by a preorder traversal. The tree file needs to retain tree structure for reconstruction.

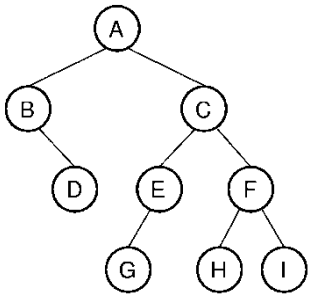

Figure 1. Sample Binary Tree

For example, for the binary tree of Figure 1, the corresponding sequential representation would be as follows (assuming that '/' stands for NULL.): **AB/D//CEG///FH//I//**

You are allowed to invent your own file format as long as the file you generate is a readable text file that can be interpreted by the TAs.

Program Invocation:

The programs are to be invoked as follows:

buildtree <frequencyfile> <treefile> <codefile>

encode <codefile> <messagefile> <encodedmessagefile>

decode <treefile> <encodedmessagefile> <messagefile>

If the command line contains an insufficient number of arguments or if any of the specified input files does not exist, the program should print an appropriate error message and exit.

Submission: follow the instructions at http://cs.uakron.edu/~echeng/Submission_How.html. Use course number 3460:316.

Sample Input and Output Files:

Refer to the website for more complete versions of these files.

frequency.txt

```
c  4
d  2
-  15
a  11
b  7
e  20
!  4
+ 1
f  5
.  0
z  0
w 0
...
(all other   letters have frequency  0)
```

codes.txt

```
a  101
b  001
c  0001
d  00001
e  11
f  1001
g  ...
h  ...
...
z  ...
-  01
.  ...
!  1000
+ 000001
```

message.txt

```
a bad
face
```

encodedmessage.bin
(this is a six-byte binary file)

10101001 10100001 10001001 10100011 11000000 00100000