CS 445 – Data Structures – Assignment#5[1]

Due: Tuesday Dec. 12th@ 11:59pm

All source files plus a completed Assignment Information Sheet zipped into a single .zip file and submitted properly to CourseWeb.

Late submission: Thursday Dec. 14 @11:59pm with 10% penalty per late day

OVERVIEW

Purpose and Goal: We have discussed binary trees (BTs) and you will utilize them in a rough implementation of a famous compression algorithm: Huffman Compression. In this program you will utilize a pre-made Huffman compression tree to compress and decompress selected text strings.

DETAILS

Huffman Compression is a very useful compression algorithm that is utilized in several commonly used compression schemes. The details of Huffman compression, its viability and how the Huffman tree is initially created are not necessary for this project, but may be of interest to you. If you would like more information on the details of Huffman Compression, see: http://en.wikipedia.org/wiki/Huffman_coding

In this assignment, you must do the following:

1) Read in a representation of a Huffman tree from a text file and generate a Huffman tree of binary nodes. See more details below about the format of the file and requirements for the tree.

2) Use the tree generated in 1) above to create a Huffman encoding table. See more details below about the format of the encoding table.

3) Initiate an interactive session with the user that will continue until the user elects to quit. Each iteration has the following options:

    a. Encode a word. In this case, show the user the set of valid characters and prompt the user to enter a word in normal text. Your program should output the equivalent Huffman codes as strings of 0's and 1's, with the code for each character being shown on a separate line. If the input String cannot be encoded properly, indicate this to the user. See example output in the files specified below.

    b. Decode a Huffman code. In this case, show the user the Huffman encoding table and prompt the user to enter a Huffman string of 0s and 1s on a single line. There should be no

[1] Assignment adapted from Dr. John Ramirez's CS 445 class.

spaces in the line. Your program should output the equivalent text string on a single line. If the Huffman code cannot be decoded properly, indicate this to the user. See example output in the files specified below.

c. Quit the program

Note that for a real Huffman code, the encoded output would be binary bits. Since we are instead using strings of 0's and 1's, we are not actually compressing anything (rather we are making the data bigger). However, we can still see how the process works and see what the binary bits WOULD be if we completed the implementation.

Reading the tree: Creating a Huffman tree from scratch is a non-trivial process. However, reading a representation of a Huffman tree from a file can be done in a fairly simple way utilizing recursion. The input file will consist of a number of lines of text. Each line will be one of two possibilities:

1) The single letter "I". This indicates that the node is an Interior node in the Huffman tree. *For simplicity, in this case we will consider the root to also be an interior node.* Due to the nature of Huffman trees, all interior nodes will have two children.

2) The single letter "L" followed by a single space, followed by a single letter. This indicates a Leaf node in the Huffman tree. The second letter on each line can be any upper-case letter, 'A' through 'Z', but each letter in the alphabet can appear in at most one line. These second letters represent the characters that can be encoded by the Huffman tree. For simplicity, assume that any subset of the alphabet in the file will always be a contiguous sequence starting at 'A'. For example, if the file contains only 5 letters, the leaf nodes will be 'A', 'B', 'C', 'D' and 'E'. However, they do not have to appear in the file in any particular order.

Your algorithm can read the tree in the following way:

- Start with an empty binary tree, using the BinaryNode class from the text (or your extension of it) to store your tree nodes. Use BinaryNode<Character> for your node type so that you can store character values within your nodes.

- Recursively build the tree as follows:

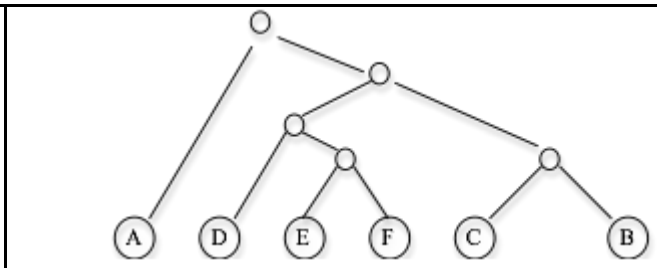
  - Read the next line from the file

  - If the line starts with an "I", it represents an interior node and thus must have two children in the tree. Create a BinaryNode with a dummy character for its data (ex: '\0') and then recursively read in its left subtree and then recursively read in its right subtree.

  - If the line starts with an "L", it represents a leaf node. Split the line and use the second character for the data value. Make a new BinaryNode using the data value. Note that this is a base case so it does not recurse.

  See various example binary tree algorithms discussed in class for help with this process. Below is a simple example of the tree building algorithm:

```
I
L A
I
I
L D
I
L E
L F
I
L C
L B
```



The file in the left box represents the Huffman tree shown in the right box. Even though the interior and leaf nodes look different in the picture, you will use the same type for all of your tree nodes.

To obtain actual Huffman codes from this tree, assign the value '0' to each left edge and the value '1' to each right edge. Thus, for example, the character 'D' would have the Huffman code "100" and the character 'F' would have the Huffman code "1011".

Creating the table: A Huffman tree can be used directly for decoding Huffman codes, but it is harder to use it for encoding. This is because to encode we need to start with a letter and end up with a Huffman code. However, in the Huffman tree the letters are leaves and we don't have an easy way of locating a particular letter, or of going back up the tree from a leaf. Thus, to encode we will first **traverse** the tree and generate an encoding table. This is actually fairly simple to do with an **inorder** traversal and a StringBuilder. As mentioned above, we will assign the bit '0' to the left child of a node and the bit '1' to the right child of a node. As we recursively traverse the tree, we add the correct bit to our StringBuilder with each recursive call, and we remove the bit when we backtrack. Upon reaching a leaf node, the current StringBuilder should contain the Huffman encoding string for that letter. We then store that value in an array of Strings. To make indexing easier, we will map the letter 'A' to location 0 and successive letters to the following locations. Given the example tree above, the resulting encoding table will be:

Index	Letter	Huffman Code
0	A	0
1	B	111
2	C	110
3	D	100
4	E	1010
5	F	1011

Encoding a text string: Once you have the encoding table built, encoding a string is a simple process. Simply iterate through the characters of the string. For each character, look up the Huffman code in the encoding table and append the results of all of the lookups together. To make it easier to grade, separate the codes of different letters with line feeds. For example, given the text string "BAD", your program would append "111", "0" and "100" to yield "111\n0\n100\n".

Decoding a Huffman string: Decoding a Huffman string works in the following way:

- Start at the root of the Huffman tree and at the first bit (character) in the Huffman string
- If the current bit is a '0', go left in the tree
- If the current bit is a '1', go right in the tree
- When a leaf in the tree is reached, append the corresponding character to the output and return to the root
- Continue until all of the bits have been consumed

For example, given the Huffman code string:

1110100

The process would be:

Go right, go right, go right – append 'B' and return to root

Go left – append 'A' and return to root

Go right, go left, go left – append 'D' and return to root

No more bits remain so the decoding is over – return "BAD"

Representing the Tree: The Huffman tree must be stored as a dynamic tree of BinaryNodes. You may use the BinaryNode class as provided by the author (see the attached code), or you may modify it as necessary to make your implementation simpler. Either way, be sure to include all files that you use for this assignment, whether you have written them or the textbook author has written them.

This program has a lot to it so you definitely want to start early. However, none of the various parts is in itself extremely difficult. Proceed step by step carefully and you should be able to complete it. To give you an idea of how the finished product would look, you can find some test runs. See the attached files a5out1.txt and a5out2.txt.

SUBMISSION REQUIREMENTS

Be sure to submit ALL files necessary to compile and run your program, as well as your completed Assignment Information Sheet. This includes all source files (both yours and the textbook author's) as well as all test data files and any other files that you may have used. Your files should be zipped and submitted as a single .zip file to the CourseWeb.

The idea from your submission is that your TA can compile and run your program WITHOUT ANY additional files OR PROCESSING. This means compilation and execution on the command line using the javac and java commands. Test your programs from the command line before submitting – especially if you use an IDE such as NetBeans or Eclipse to develop your project.

As you did in previous assignments, be sure to document your code adequately, especially in segments where the logic is not obvious.

Extra CREDIT

As a challenging extra credit option, add code to generate the initial Huffman tree by processing a text file and creating the tree based on the relative frequencies of the characters in the file. Once the tree has been generated, save it in the format described above so that it can be read in again later if necessary. This is a lot of work so don't attempt it unless you have already completed the required elements of the assignment. You may need to consult various sources to see how the Huffman tree is initially generated.

RUBRICS

Please check the grading rubric on CourseWeb.