CSCI 232: Data Structures: Project 3

Due Wednesday, Nov 23 @ 11:00pm

Project Description

Imagine we want to transmit a message (like "HELLOWORLD") across a communication link that can only send 0s and 1s. We can agree on a way to represent each possible letter, and it might look something like this:

	code # 1
H	000
L	001
W	010
O	011
E	100
D	101
R	110

"HELLOWORLD"→ 000 100 001 001 011 010 011 110 001 101

Spaces have been added for clarity, but they are not a part of the encoding. Now what about a code in which different letters are encoded to bit sequences of different lengths, like this:

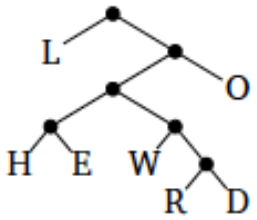
	code # 2
H	1000
L	0
W	1010
O	11
E	1001
D	10111
R	10110

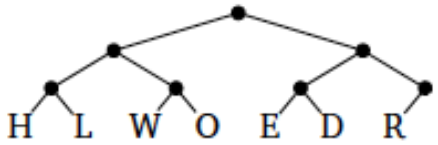
"HELLOWORLD"→ 1000 1001 0 0 11 1010 11 10110 0 10111

Even though some letters are encoded with as many as five bits, code #2 produces a more efficient encoding for the string "HELLOWORLD" (29 bits instead of 30).

Of course, what good is a code that cannot be decoded by the intended receiver? In this problem, we want to be able to decode the 0s and 1s back into the original message (there will be no spaces to mark the ends of characters, just 0s and 1s), and this is only really possible when no codeword is a prefix of another codeword. For example, if A encodes to "0" and B encodes to "00", then do you decode "00" as AA or as B?

A code with this property can be represented by a binary tree. The leaves of the tree are the letters to be encoded. The path from root to leaf gives you the encoding of the letter stored at that leaf, where "0" corresponds to moving left and "1" corresponds to moving right. Both codes listed above are valid codes as represented by the trees below:





The **input** to this program will be in a text file consisting of:

- Several lines, each containing a single character (A–Z), followed by a space, followed by a sequence of 0s and 1s. These represent the encoding scheme (see examples below).
- An empty line.
- A line containing a sequence of 0s and 1s, representing a message that is to be decoded.

Your **output** printed to the console should be one of the following:

- The string "Error: not a valid code" if the given encoding scheme is not a valid code.
- The string "Error: cannot decode message" if the encoding scheme is valid but the message cannot be decoded.
- The string "Success: " followed by the decoding of the message under the encoding scheme, otherwise.
- The number of bits read in, the number of characters in the original message, and the *compression factor*. For example, the original message above contains 10 characters which would normally requires 80 bits of storage (8 bits per character). The compressed message uses only 30 bits with code #1, or 37.5% of the space required without compression. The compression factor depends on the frequency of characters in the message, but ratios around 50% are common for English text. Note that for large messages the amount of space needed to store the description of the tree is negligible compared to storing the message itself, so we have ignored this quantity in the calculation.

Sample data:

Input:	H 000 L 001 W 010 O 011 E 100 D 101 R 110 000100001001011010011110001101	A 100 B 01 C 0101 D 010 10001	A 100 B 01 C 101 D 1100 100111101
Output	Success: HELLOWORLD Number of bits = 30 Number of characters = 10 Compression ratio = 37.5%	Error: not a valid code	Error: cannot decode message

Hints: First, starting with an empty binary tree, insert each new character at the position specified by its codeword. You may have to create several new internal nodes along the way as you are inserting. It is during this phase that you should be able to detect when the encoding scheme is not a valid code (how will you check?). This part could be recursive.

Now that you have the binary tree, scan through the bits of the message. Starting at the root, traverse the tree according to the 0s and 1s until you find a leaf. Add that leaf to the output, then reset to the root again. Be sure to check for errors! This part might be best done iteratively, by moving a finger through the nodes of the tree (since we only ever move the finger "down" the tree).

Test Cases:
Creating your own test cases to validate your program's correctness is a critical part of programming. After you have successfully tested the 3 cases in the table above, you should create additional test cases that push the limits of your program and illustrate that it works correctly given a wide variety of inputs, both valid and invalid.

To stress the importance of this aspect of programming, 10% of this project's total grade will be based off the test cases you create and use. Create a separate file entitled ***Reimer_proj3_testing***, where Reimer is your last name instead of mine, and upload it to the project dropbox along with your program files. For a minimum of 3 additional test cases, clearly indicate the input, the expected out, and why you included it as a test case (e.g., what specifically where you testing in your program with it?)

Project Deliverables

Submit your Python source files and your test cases document to the Moodle drop-box by the due date and time.

General Policies & Grading of Projects

Ground rules:

- Start early, and do not hesitate to seek early feedback from me!
- Your final submission should represent ***your*** understanding of the problems.
- You can talk with your classmates about this problem, but do so in terms of general ideas of how to solve, on a whiteboard. Never copy code from any source, and never give your code to someone else. This constitutes cheating.

Grading Rubric

Name:

Grading Criteria	Max Score	Comments
Program runs correctly and Adheres to Project Requirements		
• The string "Error: not a valid code" if the given encoding scheme is not a valid code.	20	
• The string "Error: cannot decode message" if the encoding scheme is valid but the message cannot be decoded.	20	
• The string "Success: " followed by the decoding of the message under the encoding scheme, otherwise.	25	
• The *compression factor* correctly determined and displayed for each successful decoded string.	10	
Output Program output is nicely formatted and clear.	5	
Code organization Program is clearly organized and modular. Sensible variable names and function calls are used.	10	
Test Cases A minimum of 3 additional test cases is provided, along with explanations of why they were included.	10	
TOTAL	100	