Department of Computer Science

CS260 Data Structures

Programming Assignment IV

1 Regulations

Due Date	: *Check Blackboard*
Late Submission Policy	: *The assignment can be submitted at most 2 days past the due date. Each late submission will be subjected to a 10 point per day penalty.*
Submission Method	: *The assignment will be submitted via Blackboard Learn/Gradescope*
Collaboration Policy	: *The assignment must be completed individually.*
Cheating Policy	: *Do not use code from sources except lecture slides. "Borrowing" code from sources such as friends, web sites, AI tools for any reason, including "to understand better" is condiered cheating. All parties involved in cheating get a 0 for the assignment and will be reported to the university.*

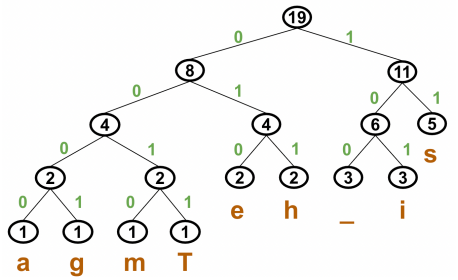
2 Lossless Data Compression and Huffman Coding

ASCII, abbreviated from American Standard Code for Information Interchange, is a character encoding standard for electronic communication. ASCII codes are used in representing text in computers and telecommunications equipment where each character is encoded with 8 bits. Since there are 256 different values that can be encoded with 8 bits, there are potentially 256 different characters in the ASCII character set. As an example, the letters a,b,c are encoded in ASCII as 'a' = 01100001, 'b' = 01100010, and 'c' = 01100011.

Data compression allows encoding information economically, which becomes handy when storing large files as well as for transferring them over a network. An efficient way of compressing text files is by accounting for characters that are more frequent than others in a certain file. Huffman code is one such approach that is commonly used for lossless data compression. You can find information about the logic of Huffman Coding along with a pseudo-code of the encoding algorithm in the recommended reading text Algorithms Unlocked, Thomas H. Cormen, Chapter 9 (available online via Drexel Library). You can also benefit from this website to visualize animation of how Huffman encoding algorithm will work for a given input text. Also, refer to the slides on Huffman Coding from class resources.

Text to be encoded: This is his message

Character	a	g	m	T	e	h	_	i	s
Frequency	1	1	1	1	2	2	3	3	5



Code Table	
s	11
e	010
h	011
_	100
i	101
a	0000
g	0001
m	0010
T	0011

Steps to generate Huffman code table for the sentence *This is his message*.

In this assignment, you will write a C program that encodes a given text file using Huffman Coding and decodes the encoded file that you generated. While doing that, you are strongly suggested to use heaps and binary trees as part of your implementation.

3 Road map

You will write a program that does Huffman 1) encoding and 2)decoding of a given text file (Note: decoding is for extra credit). Below is a rough road map of what is needed to be done to achieve this goal.

1. **Encoder:**
   (a) You will read input from a text file which consist of alphanumeric characters and punctuation marks (a sample text file will be provided). Details about input/output specifications are provided below in detail.
   (b) Calculate the frequency of each character in this text file.
   (c) Implement a priority queue data structure to be used in the Huffman Coding algorithm. (Note: You need to determine what will be the structure of the data to be kept inside the priority queue)
   (d) Implement a binary tree data structure to be used in the Huffman Coding algorithm. (Note: you need to determine what will be the node structure of the binary tree)
   (e) Implement Huffman Coding algorithm, as described in the referenced text in the previous section as well as covered in the lecture, by using the priority queue and binary tree that you implemented.
   (f) Run the Huffman Coding algorithm on the set of character frequencies that you calculated to obtain a Huffman Code tree.
   (g) Parse that tree to generate a Huffman Code table, where each character is assigned a binary code. (Note: you might want to consider storing your code table in a *closed hash table* to speed up your encoding in the next step: Think about how to implement a super simple closed hash table if you already know that the characters you will encode is the 256 characters that ASCII can encode)
   (h) Parse the input text file and encode the text in binary using the Huffman codes that you saved on your code table.
   (i) Write the Huffman code table and the encoded text into two separate files. You should store the Huffman code table in a way that you can easily reconstruct the code tree when the decoder part of your program is run.
   (j) Calculate compression statistics and print it to standard output. Your statistics should consist of the size of the original file in terms of bits (number of characters * 8, as we assume that the text is initially encoded using ASCII), the size of the encoded text in bits (i.e., count of 0/1 bits that you generated in your encoded text), and the compression ratio (the ratio of these two numbers in percentage).
   (k) For debugging purposes, you should write a print function for your Huffman code tree, that traverses the tree and prints the nodes . (Note: pre-order printing might be a good idea. Anything else that works you visualize the tree is also fine. You won't be tested against this part. However, without writing a tree-print function, I cannot imagine how you'd be able to debug your code!!)

2. **Decoder (30 points extra credit):**

   (a) Read the Huffman code table and reconstruct the Huffman Code tree (which will be a binary tree!).

   (b) Implement Huffman *decoding* algorithm.

   (c) Read the encoded text file, parse it by going through the Huffman tree and decode the characters in file using your algorithm.

   (d) Write decoded text into a new file.

IMPORTANT: Do not use available C libraries (such as qsort() function or already existing data structures native to C such as hash, heap, or list) that implement data structures and algorithms. You can use your own code from prior CS260 assignments. You can also use C library function rand() if you need a random number generator or strcmp() strcpy() type of string manipulation algorithms.

4 Starter Package

- Along with this assignment instructions document, you will be provided with:
  - a **starter code** that has a sample function to read input from file and another sample function to write into file. You might need to change these functions to match the input/output specifications of this assignment. These functions are given to you to merely serve as a helper.
  - a sample **text input file** that consists of a sentence written with alpha numeric characters, space, and punctuation marks. Test cases will not include new line character, so you don't need to worry about that special case.
  - an **executable** *huffman* **program**, which you can compare against your program to match input/output specifications.
  - a sample **code table** output file and a sample **encoded version of the text input** file that are generated by the executable that is provided. You can take these as a reference for what the output would look like.

5 Input/Output Specifications

- Your **executable** (named "huffman" for representation in this document) is going to take **four command line arguments** depending on its **mode** of encode or decode.
  - **Encode:** In order to encode a file, your program will accept **four** command line arguments as follows:

    /> ./huffman encode plainText.txt codeTable.txt encodedText.txt

    where *encode* indicates that the program will be in encoding mode, *plainText.txt* will be the path of the file that contains the text that you will encode (any file path can come here!), *codeTable.txt* will be the path of the file that you will write the code table to, and *encodedText.txt* will be the path of the file that you will write the encoded text to.
  - **Decode:** In order to decode a file, your code will accept **four** command line arguments as follows:

    /> ./huffman decode codeTable.txt encodedText.txt decodedText.txt

    where *decode* indicates that the program will be in decoding mode, *codeTable.txt* will be the path of the file that you will read the code table from, *encodedText.txt* will be the path of the file that you will read the encoded data from, and the *decodedText.txt* will be the path that you will write the decoded text to.

- **Structure of the files** will be as follows:
  - **plainText.txt** : will be a text file that contains alphanumeric characters and punctuation marks. The input text will **not** include new line and tab character, to make the implementation a bit easier (so, you don't need to worry about them). The text will end with and EOF, which you do not need to encode. A sample file will be provided along with this instruction file. Below is an example text that could be in a text file:

    She sells seashells.
  - **codeTable.txt**: will contain the code table, where each line will consist of a character (space character will be represented as a white space), the binary Huffman code to represent this character, and the frequency of the character in the file represented with an integer number, each separated by tab character ('\t'). The *codeFile.txt* should be as follows for the example sentence above (note that, depending on how you order the ties, Huffman codes can change for characters that has the same frequency).

    s 10 5
    l 00 4
    e 111 4
    ␣ 010 2
    h 1101 2
    S 0110 1
    a 0111 1
    . 1100 1

    Note that the fourth line above indicates space character (i.e., it should be replaced by a space character in your output. It is shown here as a visible character to make sure you understand what is going on). Also note that in each line, the character, its ASCII code, and frequency are separated by a tab character (i.e., $'\texttt{t}'$ in C).
  - **encodedText.txt** : will be a text file that contains the encoded text in binary. For the example provided above, this file should consist of the following:

    01101101111010101110000100101011101111011011110000101100
  - **decodedText.txt** : (for the extra credit) will be the text file that contains the decoded text, which should look identical to the plainText.txt file above. Note that, you will need to put EOF to the end of each file.

- **Compression Statistics** for encoding will be printed to standard output. For the example provided above, the output should look as follows (where compression ratio is printed up to two decimal places):

  Original: 160 bits
  Compressed: 56 bits
  Compression Ratio: 35.00%

- **Test cases and grading:** Your program will be tested across several test cases, and grading for each test case will be as follows:
  1. You get full credit from a test case if your output is both correct and is produced within at most 1.5 time to produce the output compared to the provided executable. Correctness will be tested according to the compression ration for encoding, and the string matching for decoding.
  2. You get half the credit from a test case if your output is correct but it took longer than the above mentioned time.
  3. You get no credit from a test case if your output is wrong.

6 Submission

- Even if you develop it elsewhere, your code must run on **tux**. So, make sure that it compiles and runs on tux prior to submitting it.
- Your submission will consist of two separate files (do not compress/zip the files).
  1. A single C file named **main.c**, which includes your code for the assignment.
  2. A single file named **self_evaluation**, without any file extension, which contains answers to the following questions. Your answers can be as long or as short as you would like.
     (a) How many hours did you spend on this assignment?
     (b) What did you struggle with the most?
     (c) What did you learn from this assignment?
- Submit your assignment by following the Gradescope link for the assignment through Blackboard Learn.

7 Grading

- Assignment will be graded out of 100 points.
- You will earn 10 points for submitting a file with your self-evaluation.
- The remaining 90 points will be awarded according to how many test cases your program is able to pass.
- **Extra Credit:** You can earn up to 30 points of extra credit for implementing the decode functionality and passing the associated tests.