CS251
Huffman Coding and Decoding
(Part 3 due 02/26/2018 at 11:59pm)

Project3 slides

Overview

Compression tools are used to reduce the size of a file by reformatting its data while providing a way to decompress it back into its original format. You have interacted with the compression format *tar* before when obtaining the files for each lab in this course, and have likely also seen files compressed in the zip format. In this lab, you will get to implement your own compression/decompression tool from scratch! You will implement a compression/decompression tool for data files by using Huffman encoding.

A standard ASCII character occupies 8 bits no matter what its frequency is (in this project, we consider 8-bit ASCII codes instead of the standard 7-bit codes). We call this scheme a fixed-length encoding. For example, the string "hello" would be represented using the following bit codes (each ASCII code occupies 8 bits):

01101000	01100101	01101100	01101100	01101111
h	e	l	l	o

The string "hello" was represented using 30 bits. In this project you will learn to reduce the number of bits required to encode characters by associating variable-length bit codes to input characters.

Huffman encoding associates a variable-length code to every character in the file. The most frequent characters in the file get smaller codes (smaller in terms of the

number of bits) than the least frequent ones, which makes the compressed file smaller in size. Huffman encoding is roughly comprised of three main steps:

1. Building a frequency table containing the frequency of each character in the input file. Create Huffman tree leaf nodes containing those frequencies and index them in a Min-Heap.
2. Building the Huffman tree from the Min-Heap
3. For each character in the input file, replace it with its Huffman code and write the resulting file to the disk

In this project you will implement Huffman encoding (or compression) and decoding (or decompression) as well. This project is comprised of three parts, each has its deadline. The parts are dependent on each other, e.g., you will not be able to implement part2 without successfully completing part1.

Before starting the lab, make sure you review the lecture and project3 slides to understand how Huffman encoding/decoding works.

Setting up your environment

Remote login to data by typing:
ssh <your-user-name>@data.cs.purdue.edu

Copy the initial project3 files by typing:
cp /homes/cs251/Spring2018/project3/project3-src.tar .
tar -xvf project3-src.tar
cd project3-src

What's in the tarball
You are provided with all the skeleton files you need to implement your project. The header files (*.h) contain the class declarations while the *.cpp files contain the definitions of the class methods. You are provided with some method

declarations in the header files, but you need to add more methods to them (the provided header files are incomplete).

Reference implementation

You are also provided with the solution's executable in this project. You should run "huffman.org" to get familiar with the compressor/decompressor you are going to implement. Your final program should produce compressed/decompressed files that are identical to those produced by "huffman.org".

Tests

In the tests/ directory, there is a script "testall.sh" that you will use to test your project. You can also run it to tests different parts of the project separately, for example "./testall.sh -p1" will test part1 of the project only.

Part 1: Building the Min-Heap (Due 02/12/2018 at 11:59pm)

As shown on the project3 slides, the first step towards building a Huffman Tree is to record the frequency of each ASCII character in the input file and then build a Min-Heap that will index those frequencies. You first need to build the frequency table and then implement a Min-Heap data structure. The intuition behind using a Min-Heap to store the frequencies is because we want the *least* frequent characters to be at the lower part of the Huffman tree, which means they will get longer bit codes than those that are more frequent (look at the project3 slides for an example).

Building the frequency table from a decoded file

First off, we need to record the frequency of each ASCII character in the input (decoded) file. To this end, you will implement the method (buildFrequencyTable) in **Encoder.h**. That method will fill up the **frequency_table** array declared in **Encoder.h**, where frequency[i] will be the frequency of character with ASCII code i in the input file.

Building the Min-Heap data structure

The **TreeNode** class is the building block for the Huffman tree (which we'll build later). A Huffman tree node contains: **val** and **frequency**, which represent the character and its frequency respectively. It also contains two pointers (left and right) to the left and right child. These pointers initially point to **NULL**. Complete the implementation of the methods in **TreeNode.h**.

Now, we would like to implement a Min-Heap data structure from scratch to index the TreeNode objects based on the value of the field (**frequency**). Implement the methods in in **MinHeap.h.** Your implementation of the Min-Heap will be in the file **MinHeap.cpp.** The heap will initially contain the Huffman Tree leaf nodes, which contain the character and its frequency in the input file. The frequency value is used to establish the order in the heap, i.e., the node containing the character with the minimum frequency will be the root node of the Min-Heap. The Min-Heap should grow dynamically as we add more elements to it. It should **not** have a fixed capacity.

What you need to implement in this part

1. Build the frequency table from the decoded file: implement the method "buildFrequencyTable" in Encoder.h.
2. Implement the TreeNode data structure with its methods described in TreeNode.h (except the method join which you'll implement later).
3. Implement the Min-Heap data structure: implement all the methods in MinHeap.h in the file MinHeap.cpp. After implementing the Min-Heap, you should be able to insert and retrieve Huffman Tree nodes that have the minimum frequency values.
4. To test part1 of your project, type: tests/testall.sh -p1

Turning in part 1 **(Due 02/12/2018 at 11:59pm)**

Execute the following turnin command:
turnin -c cs251 -p project3-1 <your_folder_name>
(Eg: turnin -c cs251 -p project3-1 john_smith)

(Important: previous submissions are overwritten with the new ones. Your last submission will be the official one and therefore graded).

Verify what you have turned in by typing
turnin -v -c cs251 -p project3-1
(Important: Do not forget the -v flag, otherwise your submission would be replaced with an empty one).

Part 2: Building the Huffman Tree (Due 02/19/2018 at 11:59pm)

A Huffman tree is a binary tree (meaning its nodes have at most two children - one left and one right) which is built bottom-up by continually adding the next two nodes of smallest frequency as the left and right children of a new internal node, until only a single node remains that has no parent. This node is called the root of the Huffman tree. Particularly, the algorithm works as follows:

1. Get two nodes n_i, n_j from the Min-Heap (which are the ones with the smallest frequency among other nodes in the tree)
2. Join the two selected nodes through a parent p_{ij}. p_{ij}'s left child will point to n_i and its right child will point to n_j. p's frequency will be (n_i.frequency + n_j.frequency). p_{ij} is referred to as an *internal node.*
3. Insert p_{ij} into the Min-Heap.
4. If the Min-Heap contains more than 1 node, go back to step 1. If he Min-Heap contains a single node n, it means we are done building the Huffman Tree. n will become the Huffman Tree's root node.

Refer to the project3 slides for a running example.

Generating the Huffman Codes
After building the Huffman tree, you now can generate the Huffman codes for each input character. To this end, you will traverse the tree from the root to the leaf node and whenever we traverse the left child we append "1" to the Huffman code and

“0” whenever we traverse the right child. This way, each path from the root to a leaf node will generate a bit code that we refer to as the Huffman code. Generating the codes will take place in the method “generateCodes” in HuffTree.h. Refer to the lab slides for a running example.

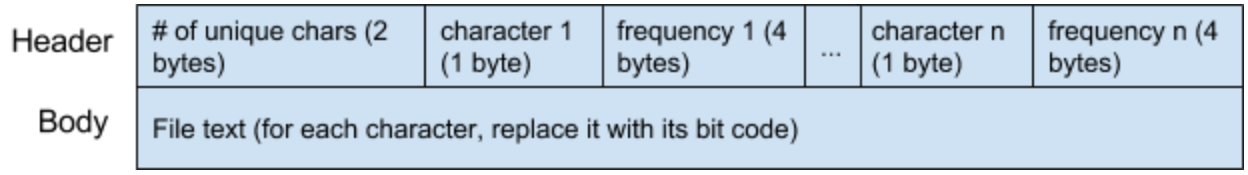
Compression

After successfully building the Huffman tree and generating the Huffman codes, you can now compress the given input (decoded) file. In the previous step you have assigned a Huffman code to every unique input character.

To compress an input file, your program will be run as follows:
./huffman -e [decodedfilename] [encodedfilename]

The above command should compress the file **decodedfilename** into a compressed file **encodedfilename.** In this project, we assume all the compressed files have the extension “.huff”.

The compressed file has the following format:



We want to include two items in the compressed file:

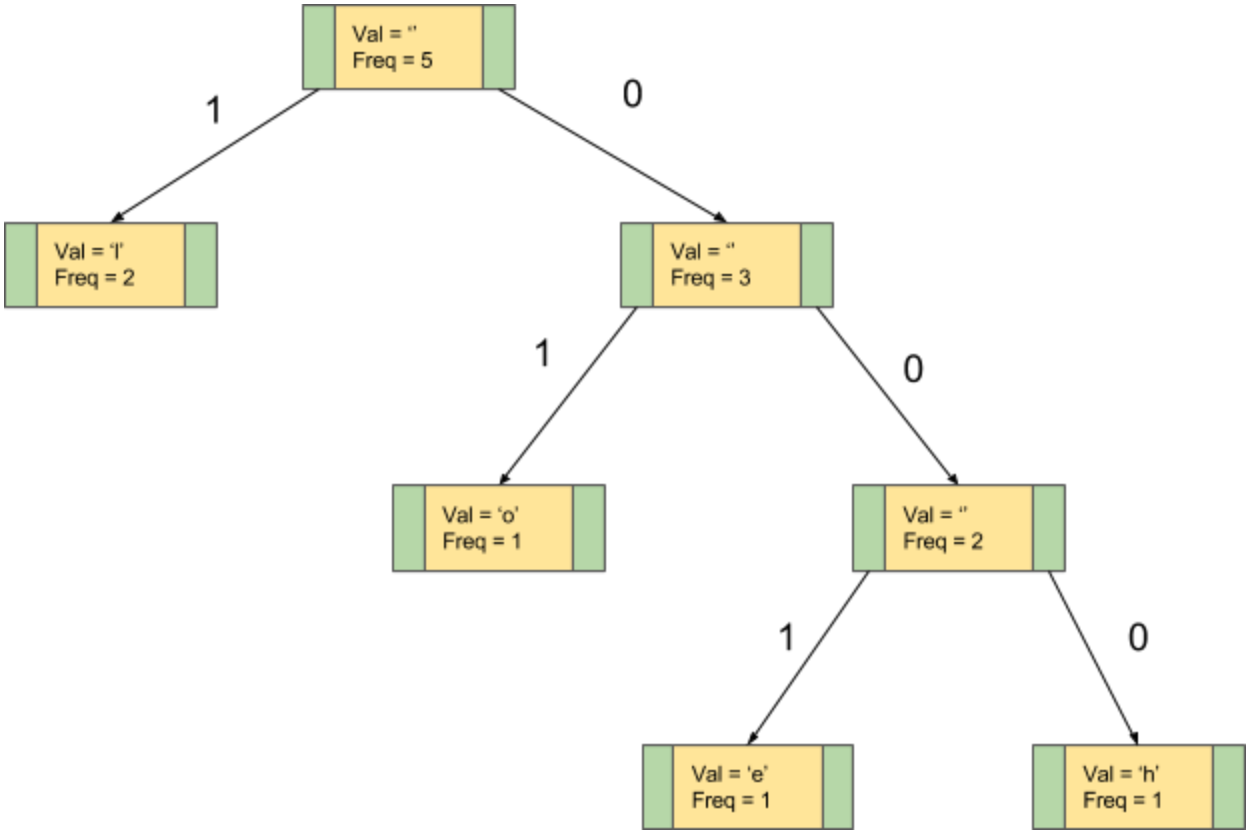
1. **Header:** Data necessary to decompress the file. We store the number of unique characters in the file along with the frequency of each ASCII character at the beginning of the compressed file. This will allow us to reconstruct the Huffman tree when we decompress the file.

2. **Body**: This part will contain all the text of the file, but instead of storing the ASCII codes, we store their corresponding huffman codes. For example if the code of "a" is 10 and of "b" is 11, then the text "abbb" will be represented as "10111111" in the body part of the compressed file.

Make sure you implement all the methods in the **Encoder.h** file.

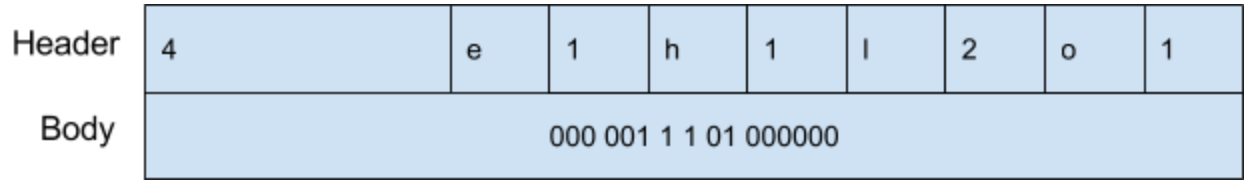
Example

Assume we have a file containing the string: "hello". The tree for such a file is as follows:



You can write a simple function to print the encoded file as a string of bits; the result will be similar to the following table (except that the values in the header will be represented as bitstrings, and the bits in the body won't have spaces between each character's encoding). **Note that at the end of the file body there**

are 6 additional 0 bits, which are needed because you need to write bits to the file a whole byte at a time.



What you need to implement in this part

1. In order to build the Huffman Tree, you will implement the method "buildTree" in "HuffTree.h". This method expects the initial Min-Heap that contains the leaf nodes.
2. Implement the methods "generateCodes(), printHuffmanTree(), getCharCode" and the other methods in HuffTree.h to generate and print the Huffman codes from the Huffman tree.
3. Implement all the methods declared in Encoder.h in the file Encoder.cpp. After completing this part, you should be able to encode files.
4. To test part2 of your project, type: tests/testall.sh -p2

Turning in part 2 (Due 02/19/2018 at 11:59pm)

Execute the following turnin command:
turnin -c cs251 -p project3-2 <your_folder_name>
(Eg: turnin -c cs251 -p project3-2 john_smith)
(Important: previous submissions are overwritten with the new ones. Your last submission will be the official one and therefore graded).

Verify what you have turned in by typing
turnin -v -c cs251 -p project3-2
(Important: Do not forget the -v flag, otherwise your submission would be replaced with an empty one).

Part 3: Decompression (Due 02/26/2018 at 11:59pm)

To decompress an input file, your program will be run as follows:
./huffman -d [encodedfilename] [decodedfilename]

The above command should decompress the file **encodedfilename** into an uncompressed file **decodedfilename.** In this project, we assume all the compressed files have the extension ".huff".

In order to decompress a compressed file, we will do the following:

1. Reconstruct the Huffman tree from the compressed file, including assigning a Huffman code to each ASCII code.
2. Scan each bit-code in the compressed file and replace it with its corresponding ascii code.

In order to be able to reconstruct the Huffman tree, we will read the total number of characters and their frequencies from the compressed file header. You will write the function **buildFrequencyTableFromFile (Decoder.h)** to read the frequencies and store them in an array. Once you produce this array, you will follow the same steps mentioned previously to construct the Huffman tree.

We are now ready to read the bit-codes from the compressed file and match them with their corresponding ASCII code. We will read the file body one bit at a time and use the value of that bit to determine if we should go to the left child or right child starting from the root of the Huffman tree. Once we reach a leaf node, the program spits out the corresponding ASCII code that the leaf node represents. Write the function **writeUncompressedFile** to produce the uncompressed file. Remember that the compressed file may have a few additional 0 bits at the end to finish the last byte; as you write characters to the file, keep track of how many you have written and stop writing to the file once the number of characters you have written equals the frequency in the root node of the Huffman tree.

Make sure you implement all the methods found in **Decoder.h** in Decoder.cpp.

Testing

You are provided a testing script (testall.sh) to test your program against the test solution huffman.org. Running "testall" will run all possible test cases against the sample solution. You can also run test each milestone separately by running "testall [-p1 | -p2 | -p3]".

Turning in part 3 (Due 02/26/2018 at 11:59pm)

Execute the following turnin command:

turnin -c cs251 -p project3-3 <your_folder_name>

(Eg: turnin -c cs251 -p project3-3 john_smith)

(Important: previous submissions are overwritten with the new ones. Your last submission will be the official one and therefore graded).

Verify what you have turned in by typing

turnin -v -c cs251 -p project3-3

(Important: Do not forget the -v flag, otherwise your submission would be replaced with an empty one).