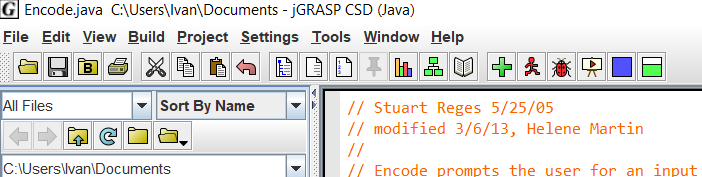
Ivan Fransazov

Huffman Encoding Explanation

This is a word document explaining the algorithm behind Huffman Coding, and how to use my program that I created. First of all, if you would like to test my program (HuffmanTree.java) you will need the support java files in the zipped file I sent (I.E: BitInputStream.java, MakeCode.java, and so on. Please note that I did not create these programs, I only created HuffmanTree.java). To execute the code, please run: java -jar out\_artifacts/HuffmanEncoding.jar (this assumes you have java JDK and JRE installed, and paths properly set). Alongside this, if you prefer to run/debug with an editor, you will need a Java IDE, it might be easiest to use jGrasp since that is what I am using and will be explaining Huffman Encoding with (however any will work). If you wish to use jGrasp here is the link (<https://www.jgrasp.org/>) click on the download jGrasp button on the right-hand side. To use the program properly, please make sure all files (.java or otherwise) are saved in the same location on your device, otherwise jGrasp will not be able to execute the compression. One last thing, please compile each .java file. To do this, find the green + sign at the top left to compile. Once this is done, you will be ready to run my program! (Note: to run a program in jGrasp, click on the button next to the green + sign of a “running” man)



Why I Did This

A little bit of background on why I did this project. At the time of writing this document, I have completed Post-AP Datastructures (Sister course at UW: CSE 143). The final mandatory project in this class regarded the use of binary trees and recursive backtracking (20 Questions program). However, we had a week after our final to cover anything we wanted. Our teacher (David Basham) told us there was one more project in the curriculum that we did not have time to get to called Huffman Coding. As a class, we ran through the PowerPoint and rushed through two weeks of lectures in two days. David told us that the Huffman Encoding project was optional and gave everyone the support files and specs required to complete it. I worked on this project for about two days on and off and got everything working and submitted it (only two people submitted it and David gave 30/30 points if he found that it was working. He did not check things like internal efficiencies, logic, internal classes, etc.). This was by far the hardest project I have ever completed and am extremely proud that I got it to work.

What HuffmanTree.java Actually Does

I will do my best to keep this as short as I possibly can and will resort to using more technical language in this section. If the reader wishes to understand this project in even more depth (which I highly recommend), they can find a more thorough explanation (with pictures) in the specs given in the attached zipped file. To start, we can split the project into two sections: encoding and decoding. Regarding encoding, the constructor for the HuffmanTree is given an array (length of 256) and the value at index i corresponds to the occurrences of that ASCII value (which is determined by the index, i). Using this, we create a PriorityQueue and fill it with single HuffmanNodes (have the following member variables: HuffmanNode left, HuffmanNode right, char value, and int frequency). So, each node will have a left and a right node so there can be a Huffman Tree, and will also have a frequency count (only used for building the tree encoding side), and its associated character value (ex: ‘a’, ‘y’, EOF). Then, once the PriorityQueue has been populated, the program sorts it into a tree by combining the two lowest frequency nodes into a tree and repeats the process until there is a Huffman Tree (only nodes that actually have character values are the leaves, that is how this becomes a lossless compression algorithm). Once that has been done, then the helper programs can call write(). This function converts the tree into a code file by recursive backtracking. On the decoding side, the constructor of the HuffmanTree is given a Scanner on a code file. The goal here is to rebuild a HuffmanTree with only a code file, so that the program can convert the encoded bits into characters (sorry, I couldn’t find a better way to explain this part without going off for a few hundred words). Once the HuffmanTree has been rebuilt from the code file, the helper program decode.java calls decode(). This program uses recursive backtracking to move along the inputted encoded bits (compressed file) and traverses the tree accordingly. Once it reaches a leaf, it prints that character and starts from the top of the tree but does not move its place along the encoded bit chain (this is to ensure proper decompression). I did leave out some details on the inner-workings of my program and the algorithm in general, such as the EOF character, however I decided it was best to give a general sense of how HuffmanTree.java worked.

How to Use and Expected Results

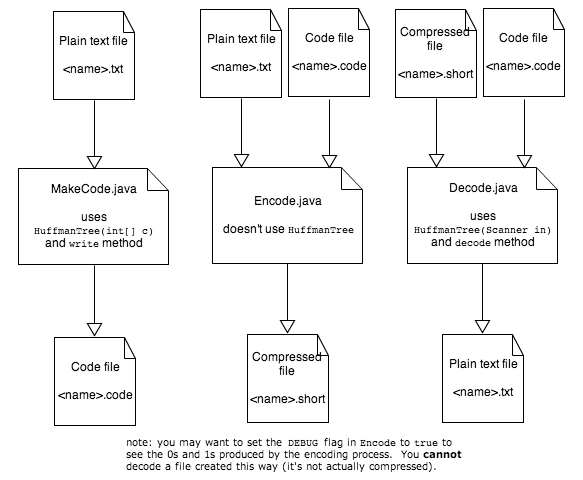
To start off, there are four files you will need to understand. The first of which is the .txt file, this is the file of text that you wish to compress. The second of which is the .code file, this is created by the MakeCode.java program. Here is an example of what the contents should look like with example.txt (which is in the attached file):

32  
00  
99  
010  
256  
011  
98  
10  
97  
11

What is happening here is explained in the specs, however I can summarize this quickly. Essentially, the first line (32) is the ASCII value for a space, it is followed by 0s and 1s which direct the computer to where the space is on the Huffman Tree. (0 being right and 1 being left). In this case, the space will be two nodes to the right of the root of the tree. This pattern continues on until the last section of ASCII and directions of 97 and 11. The third file is the .bytes file. This is only used for debugging and can show the sequence of bits as actual 0s and 1s in text. This DOES NOT compress the file at all, to make sure you compress a file go to Encode.java and find this line:

public static final boolean DEBUG = false; // set to true to print ASCII 0s  
 // and 1s for debugging  
make it “ DEBUG = false” to properly compress a file. This file should not come up for testing my program unless the reader is interested in comparing their .bytes file to the expected output (example.bytes in the attached file). The fourth, and final, file is the .short file. The difference between this one and the .bytes file is that encode.java has DEBUG set to true for .bytes and DEBUG is set to false in .short. The .short file is the actual compression of a text file, and is the one used in Decode.java to decode it back into a .txt file. With Huffman Coding, there is usually a 40% compression rate, which means that with a 100KB file, it will be 60KB when compressed as a .short file.

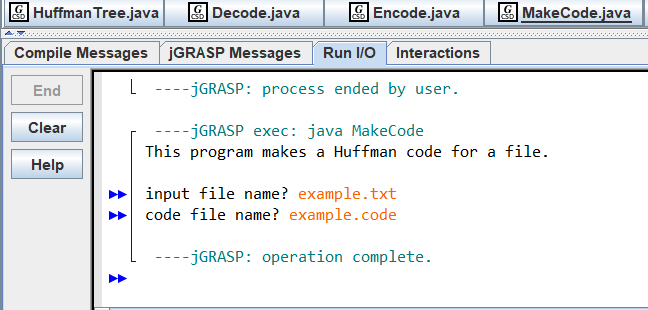
Now, the image below shows the control flow of Huffman Coding:



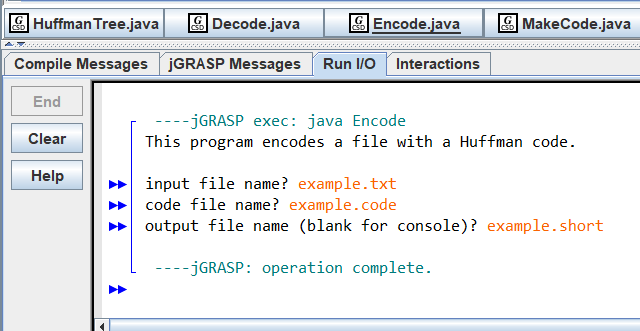
There are two steps for compression and one for decompression. To compress a .txt file, make sure that you have that .txt file saved in the same location as the other .java files (that way they can access it). The first step in compression is to create a .code file, which comes from running MakeCode.java. It will ask for the input file, type in “example.txt” (or whatever the file name you wish to compress) and enter. It will also ask for the code file name, write what you wish the code file’s name to be (I.E example.code) and enter to run. Once that step is finished, run Encode.java, and write out example.txt and enter for “input file name?” and example.code and enter for “code file name?”. If you wish to merely see the compressed characters for this, press enter again, and the following string should be printed (with example.txt): ╟í3. If you wish to make a .short file instead, write out the name of the file followed by .short (example.short). Once this step is done, you have compressed a text file! If you wish to decompress your .short file, run Decode.java and input “example.short” for encoded file name and press enter, then input “example.code” for code file name and press enter. If you wish to only read the message, press enter and it should print out “ab ab cab” (for example.short). Otherwise, you can write in a new name and add .txt to the end to send that string int a text file.

Explaining how to run this in a document was a rather cumbersome task. I have included images of example console interaction below to aid in any difficulty that may arise from following these instructions. If there is an issue that I overlooked to address, please look at the specs and try to piece together how to solve your problem or look up the error message in Google. I truly hope that does not happen. (Also, please note that the program is not designed for all corner cases such as compressing an empty file, because this would have taken more class time (which we did not have) to explain how to do).

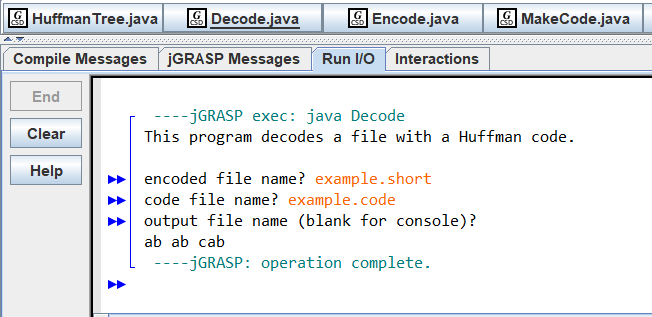
Example MakeCode.java console interaction:



Example Encode.java interaction:



Example Decode.java interaction:



SecretMessage

Alongside this project we were tasked with creating a creative text file that David would decompress and read while grading our programs. I have included it here, and if you wish to see it, run Decode.java and input secretmessage.short and secretmessage.code respectively and press enter to view the message. If you want to be surprised do not scroll further to see the expected output :)

Expected secretmessage decode output:

Here is my favorite programming joke:  
Why do programmers always mix up Halloween and Christmas? Because Oct 31 = Dec 25.