CSL 201 Data Structures
Lab 4 Due on October 1, 11.55pm

Instructions

- You are to use Java as the programming language. Use Eclipse as the IDE.
- You have to work individually for this lab.
- You are not allowed to share code with your classmates nor allowed to use code from the internet. You are encouraged engage in high level discussions with your classmates; however ensure to include their names in the report. If you refer to any source on the Internet, include the corresponding citation in the report. If we find that you have copied code from your classmate or from the Internet, you will get a straight F grade in the course.
- The submission must be a zip file with the following naming convention - rollnumber.zip. The Java files should be contained in a folder named after the question number.
- Include appropriate comments to document the code. Include a `read me` file containing the instructions on for executing the code. The code should run on institute linux machines.
- Upload your submission to moodle by the due date and time. Do not email the submission to the instructor or the TA.

This lab will improve your understanding of tree structures, heaps, priority queues, and hash tables. This is a lengthy lab, so get started early to complete it on time.

1 Huffman Encoding and Decoding (25 points)

Codes are often used to compress data that will be transmitted over a network. The simplest way that you are familiar is the ASCII code, a fixed length 7 bit binary code that encodes 2^7 characters. Another option is to use Unicode that is a 16 bit code used in Java. Transmitting data over a network is expensive and hence it is essential to achieve maximal compression on the data. That is, we would like to minimize the total number of bits that will be transmitted over the network. A fixed length code such as the ASCII code or Unicode does not do a good job of reducing the total number of bits transmitted. This is because these codes give equal importance to every character that appears in the text. In reality however, there are some characters that occur more frequently than others. Thus a coding scheme that takes into account the frequency of characters is likely to do a better job at compressing the data.
Suppose we want to transmit the text 'ABRACADABRA'. Employing a fixed length coding scheme that uses 3 bits to encode a single character, the total number of bits that will be transmitted will be $11 * 3 = 33$. Let us create a code based on the frequencies of different characters. The frequencies of the words are given in the second row of Table 1. We can now generate a variable length codes based on this frequency given in the third row of Table 1. This coding scheme requires us to transmit only 21 bits! Notice that as we process the bits there is no ambiguity in each character's code. They are all

A	B	C	D	R
5	2	1	1	2
0	10	1110	1111	110

Table 1: Variable length coding

unique. Thus the decoding process is simple. As soon as we find a character's code, we can translate it back to the character. However, we would need to know the code book for decoding.
This is an example of Huffman code. It is optimal in the sense that it encodes most frequent characters in a message with the fewest bits, and the least frequent characters with the most bits. It is also a prefix-free code. This means that no code word is a prefix for another code word. As a result the decoding process in easy. As the bits stream in, as soon as a code word is detected, the decoding can take place as no other code word will have this sequence as its prefix. Huffman encoding is performed by first finding the frequencies of characters being encoded, and then constructing a Huffman tree based on these frequencies.
The Huffman tree is a binary tree. An internal node contains the sum of frequencies of characters appearing in the sub trees rooted at the node. The root node contains the character length of the document. The leaf nodes of the Huffman tree represent the characters to be encoded along with their frequencies. The connection from a node it its left and right children are labeled as 0 and 1. The Huffman code for a character is obtained by concatenating the labels of the edges of the path from the root node of the tree to the leaf node.
The algorithm to construct the Huffman tree is as follows:

1. Input: a set of characters and their frequency of appearance in a document.
2. Create a single node tree for each character and its frequency and place into a priority queue with the lowest frequency at its root. So at the beginning, if there are n characters, there will n trees in the "forest"
3. Until the forest contains only 1 tree do the following
   (a) Remove the two nodes with the minimum frequencies from the priority queue.
   (b) Make these 2 nodes children of a new combined node with frequency equal to the sum of the 2 node frequencies.
   (c) Insert this new node into the priority queue.

Refer the lecture slides for a walk through this algorithm for constructing the Huffman code for 'ABRACADABRA'.
Input Format
The input to your program for encoding a text file will be `e filename`. `filename` is the name of the text file that has to be encoded. The format for decoding an already encoded file in binary format will be `d filename1 filename2`. `filename1` corresponds to name of the encoded file in binary format. `filename2` is the name of the text file containing the Huffman codes that will be used for decoding.
Program Specification
In stead of encoding individual characters, you will be encoding tokens. Tokens can be words, numbers, spaces and punctuation marks. You will have to implement a hash table ADT with a suitable hash function to keep track of the tokens in the document along with their frequencies of occurrence. After parsing through the text file and having determined the frequencies of all the tokens, we will construct the Huffman tree. Implement a Heap based priority queue for construction of the Huffman tree. The Huffman tree itself will be stored using linked structure representation of a binary tree.
Once the Huffman tree is constructed, update the hash table token entries with the corresponding Huffman code. Parse through the input text file again, this time outputting the Huffman code for the tokens as they appear on to a binary file. The Huffman code should also be output into a text file.
During decoding, read the contents of the file storing the Huffman codes and store them in a hash table. This time the hash function should be based on the binary code and not the token. Read the encoded file bit by bit and obtain the corresponding token by querying the hash table. Output the reconstructed token to a text file. You can verify the program by comparing the original input text file and the reconstructed text file.

2 Documentation of Code (5 points)

Documentation of code will fetch 5 points.