COP 5536 Fall 2020

Programming Project

Due Date: Nov 27th 2020, 11:55 pm EST

In computer science and information theory, a Huffman code is a particular type of optimal prefix code that is commonly used for lossless data compression. In this project, you will implement the Huffman encoder and decoder with the following details.

1. Huffman Coding

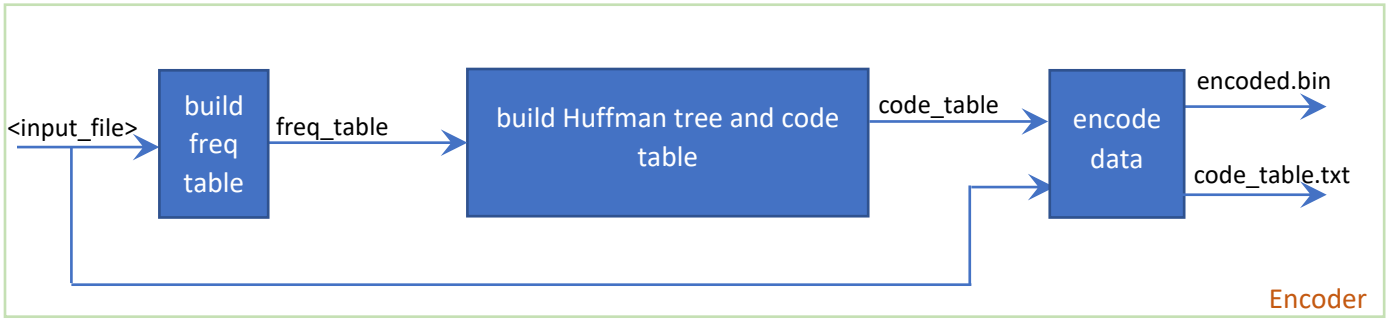
The first step in this project is to develop a program that generates Huffman codes. The algorithm for this was discussed in Lecture 7. For this project, you need to evaluate which of the following priority queue structures gives best performance: Binary Heap, 4-way cache optimized heap (lecture 10 slides 23 and 24), and Pairing Heap (lecture 15). Write code to generate Huffman trees using these three data structures and then measure run time using input data of ***sample_input_large.txt***. Use the following code as reference (this is in C++, use a similar approach if you are using some other programming language):

```
clock_t start_time;

// binary heap
start_time = clock();
for(int i = 0; i < 10; i++){                          //run 10 times on given data set
       build_tree_using_binary_heap(freq_table);
} cout << "Time using binary heap (microsecond):  " << (clock() - start_time)/10 <<
endl;

// 4-way heap
start_time = clock();
for(int i = 0; i < 10; i++){                          //run 10 times on given data set
       build_tree_using_4way_heap(freq_table);
} cout << "Time using 4-way heap (microsecond):  " << (clock() - start_time)/10 <<
endl;

// pairing heap
start_time = clock();
for(int i = 0; i < 10; i++){                          //run 10 times on given data set
       build_tree_using_pairing_heap(freq_table);
}
cout << "Time using pairing heap (microsecond):  " << (clock() - start_time)/10 << endl;
```

Once you have determined which data structure gives the best performance, finalize your Huffman code program using that data structure. The finalized Huffman code program will take as input a frequency table and output a code table. Include the timings and conclusions in your report. Once the Huffman code program is finalized, you should proceed to write the encoder and decoder.

2. Encoder

The encoder reads an input file that is to be compressed and generates two output files – the compressed version of the input file and the code table. Your encoder can follow this diagram:



Input format

Input file name will be given as a command line argument. This file can have up to 100,000,000 lines and each line will contain an integer in the range of 0 to 999,999. Consider the following input for the rest of the document:

<input_file>

```
0
2245
0
999999
2245
0
0
2245
2245
34
446
34
446
34
999999
2
```

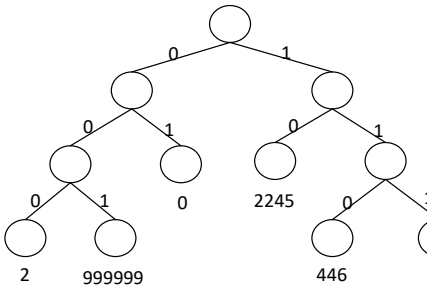
Building frequency table

First step towards Huffman encoding is to build the frequency table for each value in input. As values will be within 0 to 999999, an array can be used for storing frequency for each value. Frequency mapping for the given input file is:

0 ==> 4
2245 ==> 4
999999 ==> 2
34 ==> 3
446 ==> 2
2 ==> 1

Build Huffman tree and code table

Use the data structure with best timing from section 1 to build the Huffman code table. For the sample input given, one possible Huffman tree and corresponding code table mapping is given below. The code table can be built from the Huffman tree by doing a traversal of the tree.



Huffman tree

2 ==> 000
999999 ==> 001
0 ==> 01
2245 ==> 10
446 ==> 110
34 ==> 111

code table

Encode data

Once the code table is built, it can be used to encode the original input file by replacing each input value by its code. Please note that the values are not ASCII characters, rather **binary values**. You can use "ios::binary" flag in C++, or OutputStream in Java.
01100100110010110101111101111101110011001000

Output format

Encoder program has two output files. One is encoded message in **binary format**. It must be saved as **"encoded.bin"**. As mentioned in Encode Data phase, output encoded.bin for given data will be:

encoded.bin
01100100110010110101111101111101110011001000

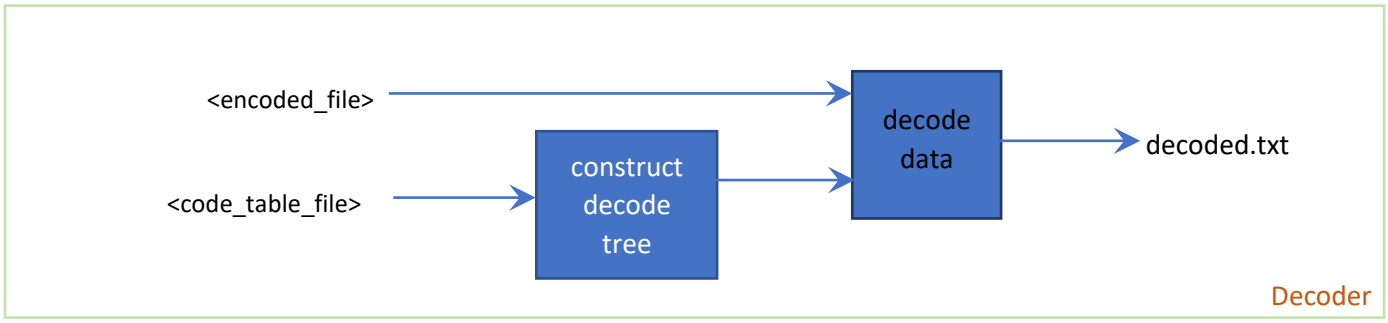
Second output is the code table. Each line in this output file will contain value of a leaf node of Huffman tree, and its code separated by a space. It must be saved as **"code_table.txt"**.

code_table.txt
2 000 999999 001 0 01 2245 10 446 110 34 111

***Note:* Both "encoded.bin" and "code_table.txt" are NOT unique. They will not be used for grading directly. However, the size of "encoded.bin" is unique, so this size will be used for grading purpose. "code_table.txt" will be used by your decoder program to decompress the encoded file.**

3. Decoder

The decoder reads two input files - encoded message and code table. The decoder first constructs the decode tree using the code table. Then the decoded message can be generated from the encoded message using the decode tree. **Your report should include a description of the algorithm used to construct the decode tree from the code table together with the complexity of this algorithm.**

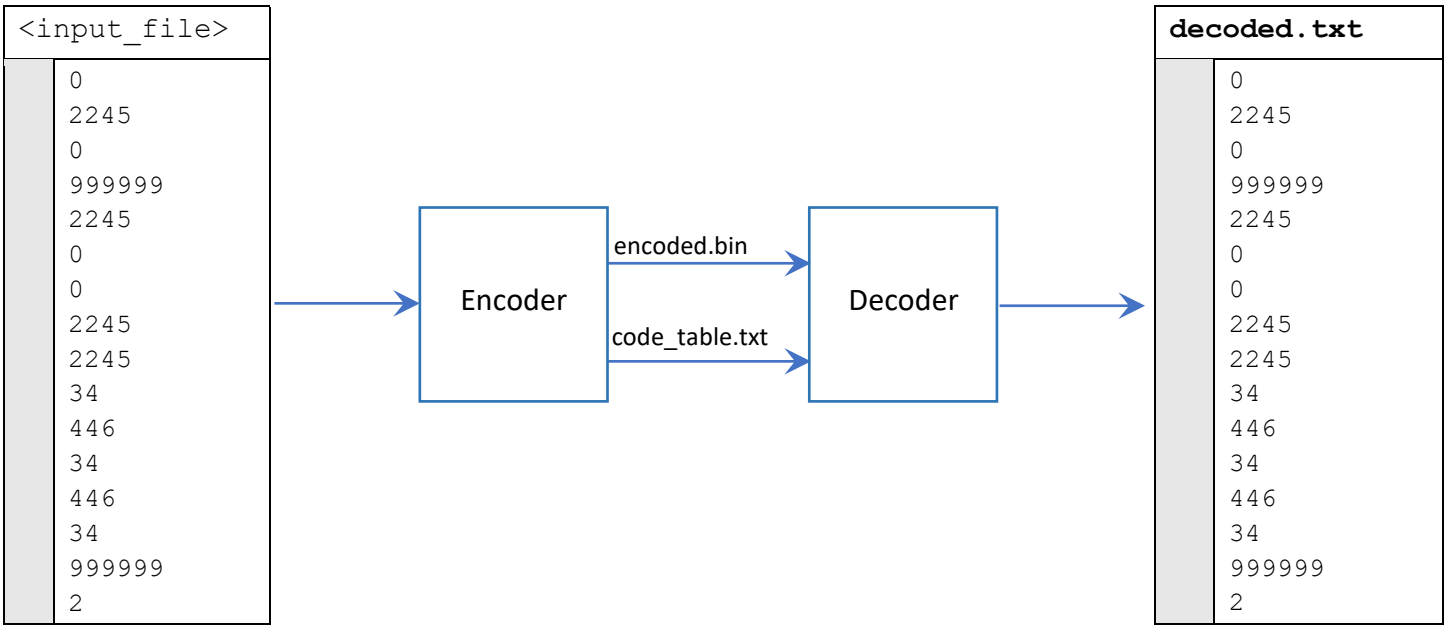


Input format

The decoder takes two input files - encoded message and code table. File names will be given as command line arguments. The format of these files is the same as the output format of the encoder, so that the output files of the encoder program can be directly used by the decoder program. **Note that the input encoded message will be in binary format, not ASCII characters. You can use ios::binary flag for C++ or InputStream for Java.**

Output format

Output of decoder program is the decoded message. It should be saved as **"decoded.txt"**. The format of this file is same as the input format for the encoder. For any sample input, the following scenario should be true:



4. Files provided in elearning

- sample1/.. ;inputs/outputs mentioned in this document
  - i. sample_input_small.txt
  - ii. encoded.bin
  - iii. code_table.txt iv. decoded.txt
- sample2/.. ;large inputs/outputs set
  - i. sample_input_large.txt ;use this file for performance measurement
  - ii. encoded.bin
  - iii. code_table.txt iv. decoded.txt
- COP5536_FA20_Project.pdf ;this file

5. Run environment

Your code must be able to run in the linux environment. You can test your program either on CISE **storm.cise.ufl.edu** or **thunder.cise.ufl.edu** server (for which you need to login using your CISE account). Write a Makefile so that we can build your code using following command at terminal:

```
$ make
```

This command should produce two binary files: **encoder** and **decoder.**
As mentioned before, **encoder** should take one input file. We will run it using following command:

```
$ ./encoder <input_file_name>                    [For C++]
$ java encoder <input_file_name>                 [For Java]
```

Running **encoder** program must produce the output files with exact name **"encoded.bin"** and **"code_table.txt".**
On the other hand, **decoder** will take two input files. We will run it using following command:

```
$ ./decoder <encoded_file_name> <code_table_file_name>      [For C++]
$ java decoder <encoded_file_name> <code_table_file_name> [For Java]
```

Running **decoder** program must produce output file with exact name **"decoded.txt".** *Any violation of these rules will incur 25% penalty.*

6. Submission

Your submission must contain the following files:

1. **Makefile:** To build your program.
2. Your source codes. Do not use any subdirectory.
3. **Report.pdf**:
   a. Should be in pdf format.
   b. Should contain your name and UFID.
   c. Present function prototypes and structure of your program.
   d. Include performance measurement results and explanation.
   e. Include what decoding algorithm you used and its complexity.

To submit, compress all your files together into LastName_FirstName.zip and upload to *elearning*. Please do not submit directly to a TA. All email submissions will be ignored without further notification. Please note that the due date is a hard deadline. No late submission will be allowed.

7. Grading Policy

1. Report: 30%
2. Performance Analysis: 10%
3. Correct implementation and execution: 60% Correctness will be tested by:
    a. For encoder only: Size of encoded message
    b. For decoder only: Correctness of decoded message
    c. For both encoder and decoder: Input message => encoder => decoder => output message. Output message should be same as input message

8. Others

- For C++ and Java, do not use any standard library container other than vector. You must implement all three types of priority queue by yourself.
- Again, go through the Run Environment section and make sure your program does exactly as required.