Advanced Data Structures (COP 5536)

Spring 2017

Programming Project Report

Shikha Dharmendra Mehta

UFID 48519256

shikha.mehta@ufl.edu

PROJECT DESCRIPTION

The goal of this project is to implement a system that uses Huffman coding so that when enormous amount of data needs to be transferred, the overall data size is reduced. This is done in three phases: **Huffman Coding**, **Encoder** and **Decoder**.

In the first phase, I developed a program to generate Huffman codes using 4-way cache optimized heap. It takes a frequency table (generated from input file) as input, and outputs a code table. This was done after a preliminary analysis of the run time of 3 priority queue structures for performance: Binary Heap, 4-way cache optimized heap, and Pairing Heap. In my analysis (given on page 7), 4-way cache optimized heap yielded the best performance on the sample input data file – sample_input_large.txt, and is therefore used in the program to perform priority queue operations.

In the second phase, I built an encoder that reads an input file (to be compressed), and generates two output files – the compressed version of the input file and the code table. This was done by first constructing the frequency table from the input file and storing it into a HashMap data structure in Java. Then, I invoked the program from phase one and outputted the code table. Once the code table is built, it was used to encode the original input file by replacing each input value by its code. The complete encoded message is outputted in binary format.

In the third phase, I wrote a decoder program that reads two input files – encoded message and code table, and yields the decoded file as output. The decoded message is generated from the encoded message using a decode tree. The algorithm I used for constructing this decode tree from the code table is described on page 8.

WORKING ENVIRONMENT

Minimum Hardware Requirements

Hard Disk space: 4 GB

Memory: 512 MB

CPU: x86

Operating System

LINUX/UNIX/MAC OS (For other OS, make command won't work)

Compiler

javac

INSTRUCTIONS FOR EXECUTION

The project has been compiled and tested on thunder.cise.ufl.edu and Java compiler on local machine.

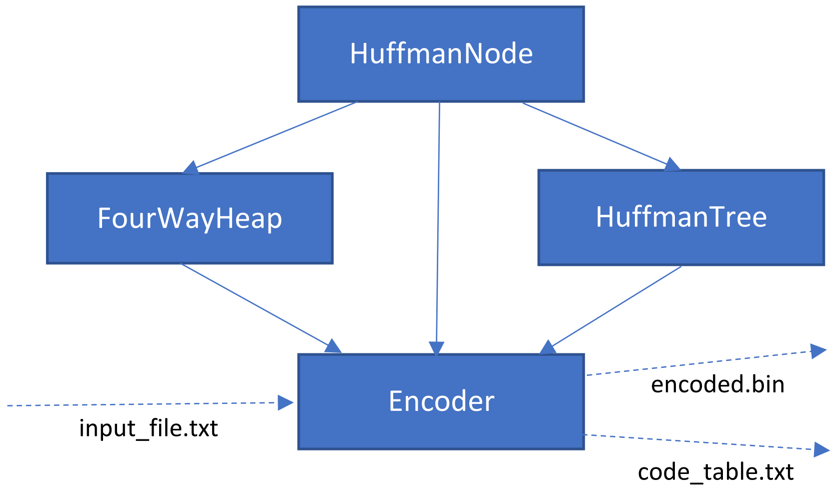
To execute the programs, you can remotely access the server using SSH (username@thunder.cise.ufl.edu) and extract the contents of the zipped folder. Then, run the following commands-

1) **make**
2) **java encoder <input_file_name>**
   I've also included sample_input_large.txt in the zipped folder
3) **java decoder <encoded_file_name> <code_table_file_name>**

Note that all file names used as command line inputs should specified complete file paths.

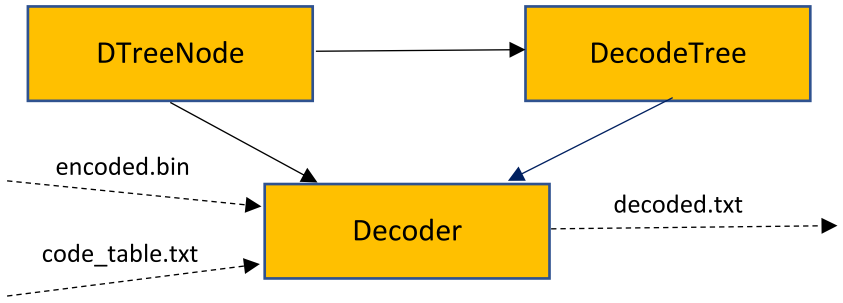
PROGRAM STRUCTURE AND FUNCTION PROTOTYPES

There are four classes used in the program for Encoder, related as shown below.



HuffmanTree.java consists of classes HuffmanNode and HuffmanTree
FourWayHeap.java consists of class FourWayHeap
Encoder.java consists of class Encoder

There are three classes used in the program for Decoder, related as shown below.



DecodeTree.java consists of classes DTreeNode and DecodeTree
Decoder.java consists of class Decoder

Encoder.java

Function Name	Return Type	Parameters	Description
private static HashMap<Integer,Integer> **build_freq_table**(String filename)	HashMap <Integer, Integer>	String filename	Builds the frequency table and stores it into a HashMap keyed on the integers found in the input file (filename) with values as frequency count.
private static HuffmanNode **build_tree_using_fourWay Heap**(HashMap<Integer, Integer> freq_table)	Huffman Node	HashMap <Integer, Integer> freq_table	Uses the 4-way min heap structure for constructing the Huffman tree from the frequency table.
private static HashMap<Integer,String> **generateHuffmanCodes** (HuffmanNode root,String huffmanCode,HashMap <Integer,String> code_table)	HashMap <Integer, String>	HuffmanNode root, String huffmanCode, HashMap <Integer, String> code_table	Generates Huffman codes recursively from the Huffman tree root and stores them into a code table (implemented as a HashMap keyed on the integers from input and values as Huffman codes).
private static HashMap<Integer,String> **build_tree_and_code_table** (HashMap<Integer,Integer> freq_table)	HashMap <Integer, String>	HashMap <Integer, Integer> freq_table	Builds the tree for encoding and returns the code table.
private static void **encode_data**(String filename, HashMap<Integer, String> code_table)	void	String filename, HashMap <Integer, String> code_table	Encodes the input file using the code table and generates an encoded output file.

FourWayHeap.java

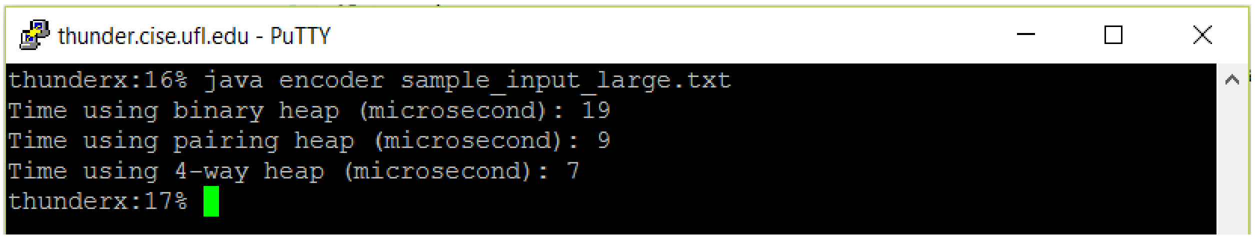
Function Name	Return Type	Parameters	Description
public boolean **isEmpty**()	boolean	none	Returns true if heap is empty, and false otherwise.
public void **insert**(HuffmanNode z)	void	HuffmanNode z	Inserts tree with root node z into the min heap.
public HuffmanNode **removeMin**()	HuffmanNode	none	Removes tree from heap with minimum frequency value at root. Returns the root node.

Decoder.java

Function Name	Return Type	Parameters	Description
private static DTreeNode **construct_decode_tree**(String codesfile)	DTreeNode	String codesfile	Constructs the decode tree from the code table.
private static void **decode_file**(String encodedfile, DecodeTree tree)	void	String encodedfile, DecodeTree tree	Uses the decode tree to generated the decoded output file from the encoded binary file.

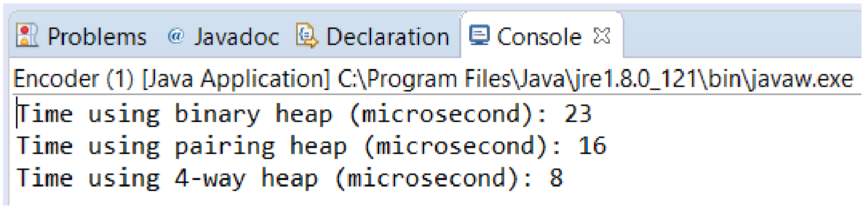
PERFORMANCE ANALYSIS RESULTS

In phase one (Huffman Coding) of the project, three min priority queue structures: Binary heap, Pairing heap and 4-way cache optimized heap were evaluated for performance. For each of these structures, I measured the time taken to construct a Huffman tree 10 consecutive times for the large sample input file. The result of one such run on thunder.cise.ulf.edu is shown below.



```
thunderx:16% java encoder sample_input_large.txt
Time using binary heap (microsecond): 19
Time using pairing heap (microsecond): 9
Time using 4-way heap (microsecond): 7
thunderx:17%
```

On my local machine, the timings were as follows.



```
Encoder (1) [Java Application] C:\Program Files\Java\jre1.8.0_121\bin\javaw.exe
Time using binary heap (microsecond): 23
Time using pairing heap (microsecond): 16
Time using 4-way heap (microsecond): 8
```

Over multiple runs, the **4-way heap** was found to consistently perform the fastest for my version of implementation. Its running time averaged over 10 iterations has been < 0.9 microseconds, followed by Pairing heap (< 1.9 microseconds) and Binary heap (< 2.9 microseconds). Thus, I have used a 4-way heap structure in my final Huffman code program.

DECODING ALGORITHM AND COMPLEXITY

Part 1

The following algorithm iteratively builds the decode tree. At the end of its execution, all the leaf nodes have integer values corresponding to those in the code table, and a unique path from the tree root to itself.

```
Create root node of the decode tree
Read the code table file line by line
For each line, i.e., (value,code) pair in code table, do the following
    Set tree pointer to root
    Read the code string one bit at a time
        If current bit is 0
            Advance tree pointer to left child
            If no left child exists, create one and then advance to it
        Else if the bit is 1
            Advance tree pointer to right child
            If no right child exists, create one and then advance to it
    For the node the tree pointer is currently at, set node data as value
```

The worst-case running time for the above algorithm is **O(nlgn)**, where ‘n’ is the total count of (*value,code*) pairs in the code table.

Part 2

The following algorithm uses the tree constructed from above to decode an encoded binary file.

```
Set tree pointer to the root node of decode tree
Read encoded binary input file in chunks of bytes into a byte array
For each chunk, do the following
    Read one byte at a time from the array and convert it into a string of 8 bits
        Read the string one bit at a time
            If current bit is 0
                Advance tree pointer to left child
                If the new node given by the tree pointer has no left child
                    Output this (leaf) node's data
                    Set tree pointer to root
            Else if the bit is 1
                Advance tree pointer to right child
                If the new node given by the tree pointer has no right child
                    Output this (leaf) node's data
                    Set tree pointer to root
```

The worst-case running time for this algorithm is **O(n)**, where 'n' is the total number of bits in the input binary file.

CONCLUSION

The objectives of this project have been met. The three priority queue structures in phase one were successfully measured for performance. Further, for both the sample input files,

a. The size of the encoded message generated using Encoder program was unique, as expected.
b. The encoded message gets decoded correctly using Decoder program.
c. For the sequence of Input message => Encoder => Decoder => Output message, the Output message produced was the same as Input message.

REFERENCES

[1] https://en.wikipedia.org/wiki/Huffman_coding

[2] http://www.cs.ubc.ca/~hoos/PbO/Examples/Java/DaryHeap/DHeap.pbo-j

[3] http://www.sanfoundry.com/java-program-implement-d-ary-heap/

[4] http://www.sanfoundry.com/java-program-implement-pairing-heap/

[5] http://stackoverflow.com/questions/12310017/how-to-convert-a-byte-to-its-binary-string-representation

[6] http://stackoverflow.com/questions/27677519/java-how-to-write-bits-and-not-characters-in-a-file