

Advanced Data Structures (COP5536)

SPRING 2017

Programming Project Report

Harshitha Gandamalla

UFID: 4993-5697

harsh1thagandama@ufl.edu

INTRODUCTION

Huffman tree encoding and decoding was implemented by building the Huffman tree using three data structures: Binary heap, 4-way heap and Pairing heap. 4-way heap turned out to be the fastest among the other data structures used. Therefore encoding and decoding of the files has been done by using 4-way heap.

PROGRAM STRUCTURE and FUNCTION PROTOTYPES

The code_table.txt and encoded bin files are generated by encoder.java files. The implementation is as follows.

The input file is taken and used to construct the frequency table using the generateCodeTable.java file. It is used to output a hashmap consisting of the respective keys and their corresponding frequencies of the input file given. This hashmap generated is used to build the tree using the four way heap data structure implementation. This tree is given as input to the encoder.java file which generates the code_table.txt and encoded.bin file which contains the code of the keys in the input and the concatenated string of codes of the input keys respectively.

The output of the encoder is given as input to the decoder which gives the decoded.txt file as output. This file basically is the input file used by the encoder. The decoder.java file takes the encoded.bin file and decodes the string using the code_table.txt as reference.

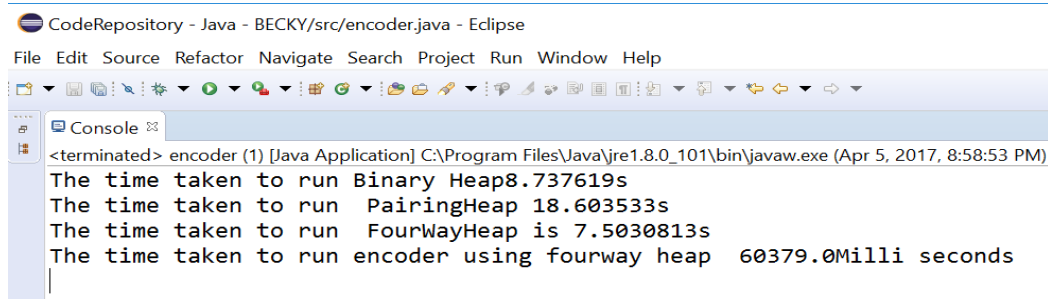
The different classes used for the project implementation are:

- Node: This class is used to define the structure of the node of Huffman tree.
- generateCodeTable: This generates the frequencies of keys of the given input file.
- FourWayHeap: This is the entire implementation of the 4-way data structure which includes buildTree, removeMin and insert methods which are crucial.
- buildTree: This method uses removeMin and insert to extract the minimum element of the tree twice and inserts the node with frequency obtained by adding the previous two remove mins frequencies and adds it back to the tree. This process is repeated until a single node representing the tree is left which is returned to the encoder.
- encoder: This class uses the frequency table and input file and generates encoded.bin and code_table.txt files
- decoder: This class builds the decoder tree for the keys in the code_table.txt using their corresponding codes and decodes the string in encoded.bin whose output is written to decoded.txt file

PERFORMANCE RESULTS and EVALUATION

On building the tree using the three heaps: binaryheap,4-way and pairing based on the running times taken to build the Huffman tree 4-way heap has proven to be the fastest of the three.

The run times of the three are:



```
<terminated> encoder (1) [Java Application] C:\Program Files\Java\jre1.8.0_101\bin\javaw.exe (Apr 5, 2017, 8:58:53 PM)
The time taken to run Binary Heap8.737619s
The time taken to run PairingHeap 18.603533s
The time taken to run FourWayHeap is 7.5030813s
The time taken to run encoder using fourway heap 60379.0Milli seconds
```

From above results it is clear that Fourway heap is the fastest.

EVALUATION:

Fourway heap was implemented using the cached optimized implementation. Doing this has improved the running time of the data structure tremendously. This has produced better run times for 4-way than binaryheap implementation.

When the input is huge i.e., when the heap size exceeds the size of computers cache memory, the 4 way heap runs faster. As the binary heap has more cache misses and page faults when compared to 4 way we have the expected output of the 4 way running faster than binary heap implementation.

DECODING ALGORITHM and IT's ANALYSIS

The decoding process starts by building the decoder tree. The decoder tree is built by using the code table generated by the encoder. The tree has for its leaves the keys of the elements in the code table and the path to it is given by its corresponding code.

The implementation details:

The code of every key is traversed and the tree is built with left and right pointers from the current node depending on if the code is a 0 bit or a 1 bit. After exhausting the code for a given key the value at the leaf is verified. If it is not equal to the value of the key then a new leaf node is created with the key as its value. The finished tree has paths for all the corresponding keys of the code table file.

The second step is to decode the bytes in the encoded.bin file.

This is done by using the decoder tree built in the first step. The bytes in the encoded.bin file is converted into a string and is traversed from the beginning. For each of the bits in the string, the decoder tree is traversed till we reach the leaf. If we encounter a 0, then we traverse the left path from the current node else we go right. The value of the leaf node is written out to a decoded.txt output file and the process continues until the entire string is exhausted.

TIME COMPLEXITY:

The decoding algorithm time complexity depends upon the time to build the decoder tree and the time taken to decode the encoded.bin file

Time to build decoder tree= $O(\text{sum of the number of bits in the codes in the code table file})$
Time to decode the encoded file= $O(\text{number of bits in the encoded.bin file})$

