Spring
2017

COP 5536: Advanced Data Structures – Programming Project

RAHEEN MAZGAONKAR
UFID : 47144316
Email : raheenmz@ufl.edu

Project Description

The goal of the project is to create a data compression algorithm for transfer of large amount of data between video streaming site MyTube and software giant Toggle. For this Huffman algorithm has been used.

The project is divided into 2 parts : encoding in which we take as input the data file and produce code table and encoded file as output and decoding which uses this code table and encoded file to reproduce original data.

In the encoding phase, we need to find out which priority queue structure (out of binary, 4-way cache optimized and pairing heap) give the best performance. Then use this priority queue structure to create Huffman tree.

Program Structure

The program flow can be given as follow

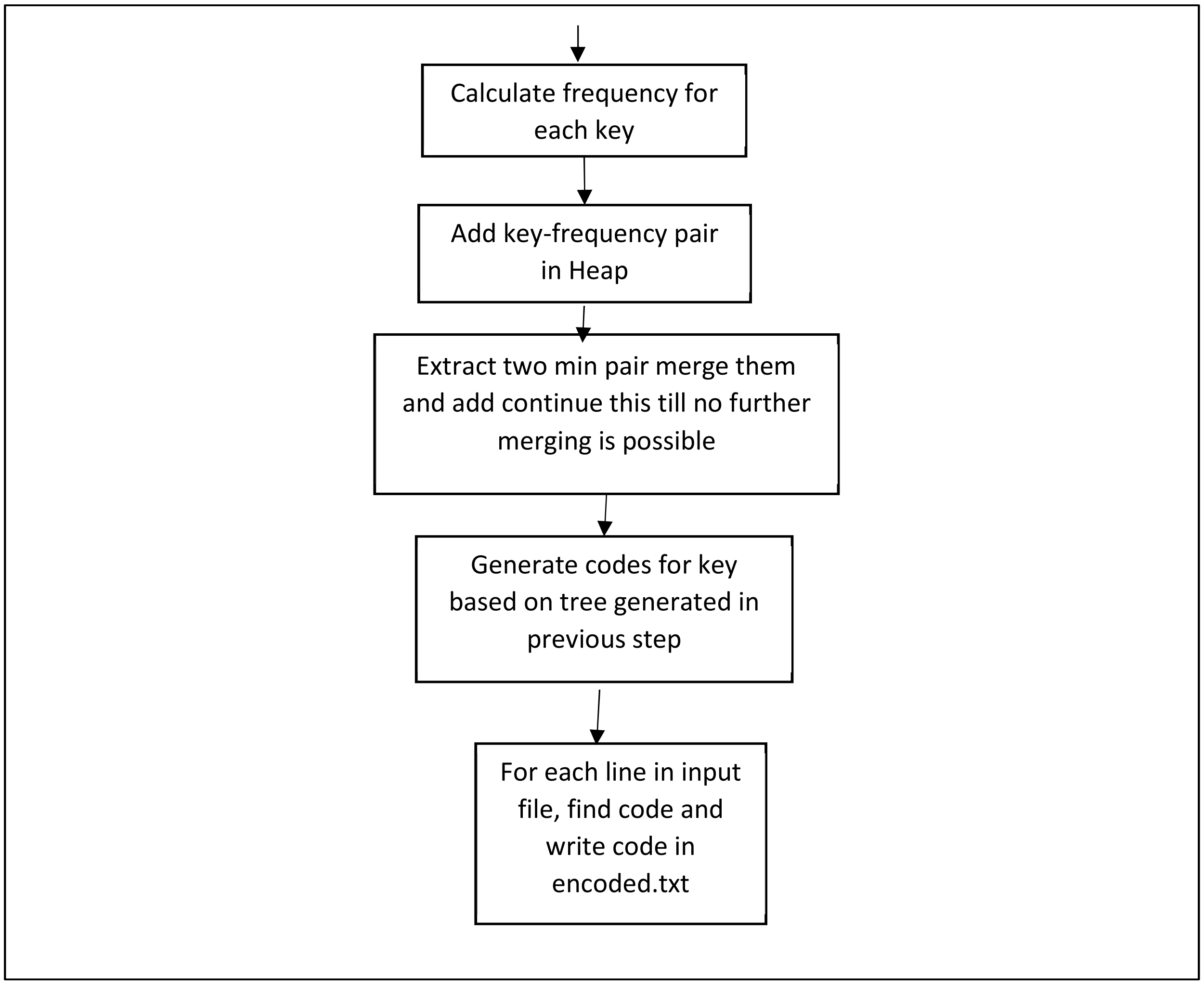
Calculate frequency for each key
Add key-frequency pair in Heap
Extract two min pair merge them and add continue this till no further merging is possible
Generate codes for key based on tree generated in previous step
For each line in input file, find code and write code in encoded.txt

Fig. 1: Flow of Encoder

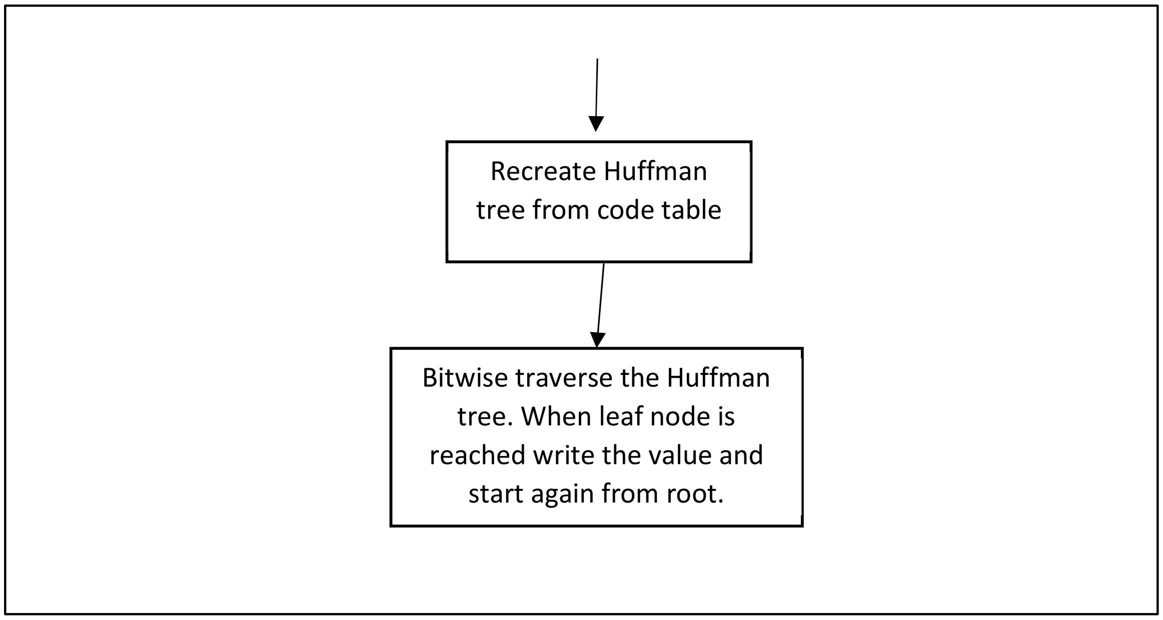
Recreate Huffman tree from code table
Bitwise traverse the Huffman tree. When leaf node is reached write the value and start again from root.

Fig. 2: Flow for Decoder

The project can be divided into 3 classes : Encoder, Decoder, File Operation

A) Encoder: Consist of functions to calculate frequency list, create Huffman tree, and finally encode the document.

B) Decoder: Consist of functions to create Huffman tree and produce decoded text

C) FileOper: Consist of functions for reading from and writing into text and binary files.

Function Prototype (Only for functions used in Final Encoder and Decoder)

1. **CLASS: Encoder**

   i) ***HashMap<Integer,Integer> FreqGen(String file)***
   Parameter: Data file
   Return Type: Frequency List
   This function calculates number of times each number appear in an input file and stores it into a Hash Map for faster retrieval.

   ii) ***String generateCode(BufferedWriter bw,Node n, String code)***
   Parameters: Buffered writer of code_table, Root of Huffman tree, Code string
   Return Type: String
   This function recursively creates code for key by traversing the Huffman tree.

iii) void insert(Node n)

<u>Parameters:</u> Node n
<u>Return Type</u>: void
This function insert node n in heap.

iv) Node extractMin()

<u>Parameters:</u> None
<u>Return Type:</u> Node
This function removes smallest element from heap and returns it.

v) ***void encode (String file)***

<u>Parameters:</u> Input file
<u>Return Type:</u> void
This functions encodes the input file using code Table.

1) CLASS: Decoder

i) DNode recreateHuffmanTree (String file)

<u>Parameters:</u> Name of the file which consist of keys and their respective code. According to our project specification, it is code_table.txt.
<u>Return Type:</u> Huffman tree
This function takes code table as input and generates Huffman tree from it.

ii) Void decode(DNode root, String file)

<u>Parameters</u>: Huffman tree, File of encoded text
<u>Return Type</u>: Void.
This function takes creates original data from Huffman tree and encoded file.

iii) DNode addToLeft(DNode curr)

<u>Parameters:</u> DNode
<u>Return Types</u>: DNode
This function returns node to the left of the current node. It no such node exists, then it creates a node to the left and then returns it.

iv) DNode addToRight(DNode curr)

<u>Parameters:</u> DNode
<u>Return Types</u>: DNode
This function returns node to the right of the current node. It no such node exists, then it creates a node to the right and then returns it.

v) DNode goLeft (DNode curr)

<u>Parameters:</u> DNode
<u>Return Types</u>: DNode

This function returns node to the left of the current node.

vi) ***DNode goRight(DNode curr)***
Parameters: DNode
Return Types: DNode
This function returns node to the right of the current node.

Performance analysis result and explanation

Time taken for generation of Huffman tree with the 3 heaps was as follows:

i) Binary Heap: approx. 4800 ms
ii) 4-way Heap: approx. 3500 ms
iii) Pairing Heap: approx. 7000 ms

As time required for 4-way heap was least. It was used for generation of Huffman tree.

Total time required for encoding sample_input_large was 60 seconds and for decoding was approximately 76 seconds.

Decoding algorithm and Complexity

The decoding algorithm is divided into 2 parts. In the first part Huffman tree is created using the code table file provided. In the next part, bits are read from the file and simultaneously the Huffman tree is traversed to find the code for the tree.

A) Recreating Huffman Tree
   i) Read Value of element and code line by line code table
   ii) For each code start from the root of the tree as current node. Scan the code bit by bit. If bit is 0 then go to the left of the current and if bit is 1 then go to the right of current. If left or right does not exist, then create them and then move.
   iii) When the node for last bit is created add set value of that node as key.

B) Decoding File
   i) Read two bytes from the file.
   ii) Starting from the root, scan the bits one n by one, if bit encountered is 0 go to the left of the tree and if bit encountered is 1 go to the right of the tree.
   iii) Continue step (ii) till you reach and external node. Upon reaching external node write the value of that node in a file.
   iv) Repeat steps (i)-(iii) till the entire file is read.

Complexity

Let n be the number of key-code pair present in the code table. For each n, it has to add it to tree. So total time to create tree will be O(n log n)

Let b be the number of bits present in the file, since for each bit we have to traverse the huffman tree. Time for decoding will be O(b).

Therefore, total time complexity will be O(nlogn + b) .

Conclusion

Priority Queue were implemented using Binary, 4-way and Pairing heap. Out of these, 4-way heap was found to be the fastest. Using 4-way heap as priority queue Huffman Encoder and Decoder were created. With this compression rate of approximately 2.8 was achieved. This will make transfer of data between MyTube and Toggle faster.