COP 5536 Advanced Data Structures
Spring 2017

Huffman Encoding and Decoding

Name:	Suyog Daga
UFID:	57735749
Email:	suyogdaga@ufl.edu

Project Description:

We use Huffman Coding to reduce data size for large amount of data transfer. To perform Huffman Coding we need to generate Huffman tree, which we evaluate with the use of three data structures : binary heap, 4-waycache optimized heap and pairing heap. This tree gives us variable length code words so that frequent characters are given short code words and the infrequent characters are given long code words. The we encode the data and decode it using decoder on other side.

Structure of the Program:

The project includes the following java files:

Node.java :

- This program is used to generate Node objects, which are used in encoding and decoding process of Huffman coding. It also has a class TreeNode, which is specifically used to generate Huffman tree during decoding process.

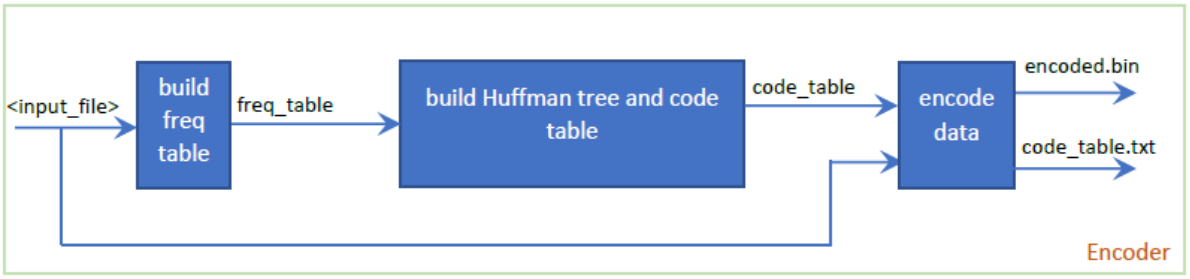
FourwayHeap .java:

- This class has the implementation of 4 way cache optimized heap data structure and has various important functions like insert ,delete and frequencyTable . This all operations are performed in such a way that min heap property of the data structure is maintained.

Encoder.java :

- This program is used for compressing the input file and generate two output files:-
  encoded.bin : Compressed version of the data
  code_table.txt: Mapping file for data values respective codes.

Following is the diagram and below are the steps:



1.Frequency Table Generation:

- Encoder class reads an input file which is passed as a frequency table method via command line argument ,which is stored in a HashMap of the value and the frequency of the value.

2. Huffman Tree and Code Table:

- Huffman tree is constructed using the 4-way cache optimized data structure using the Hashmap generated from frequencyTable() function.

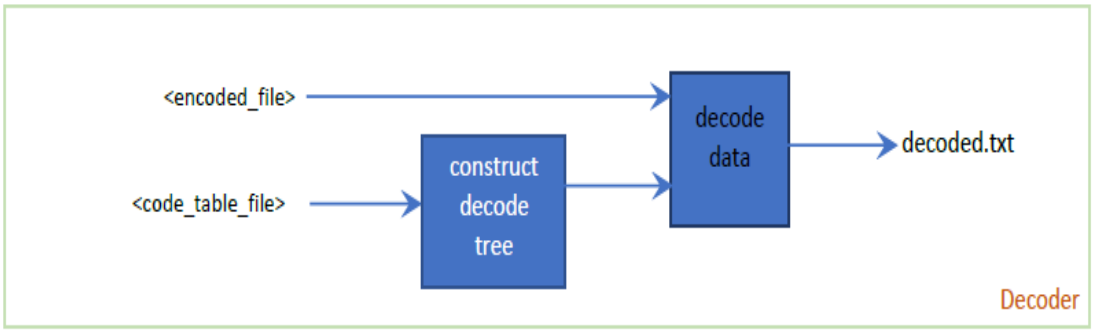
- A Huffman Tree is created by combining two nodes having the lowest frequencies and inserting it back into data structure. While combining, set parent's value to -1 indicating that it is an internal node whereas the frequency of this node is the sum of the frequencies of the two deleted nodes. The left and right field of the combined node is set appropriately. The process is repeated until only one node is left in the heap.
- Using this root value, printPreorder() function is called. It is used for preorder traversal and used to generate the codes. It maintains a string which stores the code word for a given symbol during the traversal. The function is recursive and appends '0' to the string if it recurse on the left child and appends '1' if it recurse on the right child. When a root node is encountered, it checks if it is a leaf node. If it is a leaf node, then it inserts the value(symbol) of the node and the string in the HashMap.
- The HashMap is then written to code_table.txt file using the PrintWriter function in Java.

3. Encoding the data:

- This step generates the encoded.bin file. Each of the value present in the input file is replaced by the corresponding code word as stored in the HashMap obtained from above step The encoded data is written in binary format into encoded.bin file, one byte at a time.

Decoder:

- Decoder.java takes two input files: encoded.bin and code_table.txt for the decoding process.
  Decoding is done in two steps, following is the diagram and below are the steps:



A. Construction of Decode Tree:

- For this tree, a new Node structure is created called Node which stores only the left and right child along with the value (symbol).
- The code_table.txt file is parsed and for every value in this file and create decode tree which will be used to decode the encoded binary data.

B. Decoding Data:

Decoding Algorithm:

- This file is traversed and at a time 4096 elements are fetched and stored in a byte array.
- Then each element in byte array is converted to String and is processed using the decode tree , which is explained above.
- Now, we have two condition's , when the element is '0', left subtree is parsed and right subtree when element is '1'.
- Above mentioned steps are performed, till whole of the string is exhausted or current node is leaf node. Once we reach a leaf node, then its value is written in decoded.txt file
- If we reach the end of the String temp, the next 4096 elements are fetched from encoded.bin and above steps are repeats until the end of the file.
- Decoding of data is completed once all the bytes are read from the file.

Decoding Complexity:

The decoded data is stored in decoded.txt file. The complexity of the entire decoding process is calculated as:

- Reconstruction of Huffman Tree:
  The Huffman Tree size depends on the size of the code table generated. Thus, decode tree construction takes (code_table_size * log n) time where n is the number of nodes in the decode tree.
- Decoding data:
  The running time of decoding data depends on total number of bits in the binary file. Lets assume it has n bits ,so the complexity is O(n).

Function Prototypes:

- Node Class

Variables /Methods	Type	Description
Value	Integer	Stores the value
Frequency	Integer	Stores the frequency of value
Left	Node	Left child
Right	Node	Right Child
Node()	Constructor	Default constructor for initialization
Node(value, frequency)	Constructor	Constructor for setting specific value and frequency
Node(Node A, Node B)	Constructor	This constructor is used for adding two nodes frequency when combining .

- TreeNode Class

Variable/Functions	Type	Description
value	Integer	Stores the value
left	TreeNode	Left child
right	TreeNode	Right Child
TreeNode()	Constructor	Default constructor for initialization

- Huffman Class

Variable/Functions	Type	Description
root	Node	Root Node
hashy	HashMap	Used for storing keys which are data values and values which are Huffman codes
HuffmanTree(Node x)	Constructor	Constructor for setting values
printPreorder(ode node,String s)	Method	This method is used for preorder traversal and to determine Huffman codes.

- Encoder Class

Variable/Functions	Type	Description
Build_tree_using_fourway_heap()	Method	Assists for building HashMap of frequency table.
k	Hashmap	Stores the code table, key is the data value and value is code.
filepath	String	File to be encoded
outfilepath	String	File where encoded data will be stored.
printPreorder(Node Root, String S)	Method	Preorder traversal for generating codes.
Public static void main(String args[])	Main method	Builds code table, encodes data .

- Decoder class

Variable/Functions	Type	Description
Void decode()	Method	Assists in decoding file, using codetable and encoded file.
encoded_file	String	Name of file containing encoded data
code_table	String	Name of file containing codes.
Public static void main(String args[])	Main method	Writes the decoded data into file.

- FourwayHeap class

Variable/Functions	Type	Description
FourwayHeap()	Constructor	Constructor to initialize class and instance variables
Boolean Empty()	Method	Checks if the heap is empty
Int Size	Method	Returns size.
Int findMin()	Method	Returns frequency of smallest element in data structure.
Hashmap frequencyTable()	Method	Assists in generating frequency table.
Node Insert (Node a)	Method	Inserts a node in fourway heap and maintains its property.
Node deleteMin ()	Method	Returns and deletes the minimum element in heap.
Filepath	String	This filepath is passed from command prompt via frequency table method to generate hashmap of frequency.

Performance Analysis:

	Binary Heap(in milliseconds)	4-way cache optimized heap (in milliseconds)	Pairing Heap (in milliseconds)
Small Input file	4	4	3
Large Input file	3587	2944	4607

This implementation is specific to my machine. 4-way cache optimized heap beats binary heap and pairing heap. Therefore 4-way cache optimized heap is used to implement Huffman encoding and decoding.
We can confirm this by the fact that, as d=4 the number of layers in heap goes down by a factor of 2 and therefore the comparisons per level go up by factor of two, which in turn gives us better performance owing to cache effects and overall lower height tree. Hence it outperforms the binary tree. When we do a decrease-key (and in turn insertions), we change the priority of a node in the tree, then repeatedly swap it up with its parent until it either hits the root of the tree or its priority ends up becoming smaller than the priority of its parent. Therefore worst case is given by height of heap. Since the number of nodes in each layer of a d-ary heap grows exponentially by a factor of d at each step, the height of a d-ary heap = O(log n / log d). This implies that increase in value of d, the height of the d-ary heap will decrease, so decrease-keys and insertions will take less time.

Conclusion:

The Huffman Encoder and Decoder has been successfully implemented using 4-way cache optimized heap. The Encoder correctly encodes the data into encoded.bin .Decoder uses encoded.bin and code_table.txt to decode the file and save it to decode.txt . Thus we can compress huge amount of data, send it , and decode it on another side.

Reference:

- ADS Lecture Slides by Dr. Sartaj Sahni.
- http://www.geeksforgeeks.org/
- http://www.sanfoundry.com/
- http://www.ccs.neu.edu/home/vkp/2510-sp11/Labs/Lab11/lab11.pdf
- www.stackoverflow.com