COP 5536

ADVANCED DATASTRUCTURES

HUFFMAN ENCODER

AND

DECODER

PROJECT REPORT-SPRING2017

Submitted by:
Sweta Thapliyal
UFID:3543-6779
sthapliyal@ufl.edu

Problem Statement:

Design and implement a data compression algorithm using Huffman codes for a video streaming site to send its data to a server. Design and implement both Huffman encoder and decoder for the same. The encoder will be on the client side. The encoder will compress the data file from client side and the same file will be generated by the decoder as the output.

Introduction:

for developing the encoder firstly, we are going to create the Huffman codes of all the words present in the input file. For that we analyzed three types of priority queue i.e. binary min heap, four-way cache optimized min heap and paining heap. Here the minimum time I achieved is with the four way cache optimized heap. Using the frequency and this priority queue we are building an optimized code table. This code table is use to encode the data.

On the other side the decoder use the same code table and encoded binary file to decode the data. In decoder we are building a trie from the code table and the looking up to it using the binary encoded file.

Compiler used/language:

Java Programming Language Compiler or javac. The program may need to be run with increased heap space, i.e., JVM arguments –Xmx8g for faster run.

Application Structure:

1. PriorityQueue.java

This class is an abstract class which is the parent of Binary heap, Fourway heap and pairing heap. It has the common methods of all the classes which we can use to interface it with the Huffman tree class.
Methods: the three methods here are abstract that are implemented in the Fourway heap class.
public PriorityQueue() // constructor
public abstract Node removeMin() throws Exception; // method to remove the min element
public abstract PriorityQueue create(FrequencyTable f);// creation of priority queue using the frequency table
public abstract void insert(Node node); //insertion of a node in a queue
public int size()// returns the size of the priority queue
public void printHeap()//prints all the element present in the heap

Attribute: The priority queue has only one attribute that is the array list of type Nodes. Which are the Huffman nodes.

```
ArrayList<Node> heap;
```

2. FourWayHeap.java

This class is only the implementation for the priority Queue class.

Methods:

public FourWayHeap() // constructor
public int parent(int child) //finds the parent index
public void swap(int parent, int child) // swap two nodes
private void heapifyUp(int child) // heapifys up when adding new node
private void heapify(int index) // heapify down
public Node removeMin() throws Exception // removes the min node
public PriorityQueue create(FrequencyTable f) // creates the heap using frequency table
public void insert(Node node)//insert a new node
public void printHeap()//prints the value of the heap

3. Node.java

This class implements the node structure of Huffman tree nodes. Three attributes here are pairing heap specific.

Methods:

public Node(int data, int frequency) //constructor

Attributes:

int data; //stores the string value
int frequency; //stores the frequency
Node left; // left child of the Huffman tree
Node right;//right child of the Huffman tree
Node leftChild; // left child in pairing heap
Node nextSibling; //sibling in the pairing heap
Node prev;// previous node in pairing heap

4. FrequencyTable.java

This class stores the frequency of all the words in the file.

Attributes:

public int[] fArray; //on every index the frequency for that index is stored

Methods:

public void populateFrequencyTable(File input) //reads the input from the file and populate in the fArray

5. HuffmanTree.java

This class is basically the Huffman tree class which has the node and generates the code table by creating the Huffman tree.

Attributes:

Node root; //stores the root of huffman tree

HashMap<Integer, String> codeTable; //stores the temporary code table between the generation

Methods:
public void createHuffmanTree(FrequencyTable t) // this fuction takes a frequency table and build a Huffman tree using Fourway heap
public Map<Integer, String> generateCode(Node root) //this method takes input as the root and generates the code table
public void writeCodeTable() // this method writes the generated code table on file

6. Encoder.java

This class is the main encoding class of this program. The main program of this class takes input as the input file and calls the encode method. Here we are calling the functions the store frequencies and then creating the Huffman tree and generating the code table and writing the binary file.
Methods:
public void encode(File input, HuffmanTree tree)//encode method takes the input file and the Huffman tree as the argument and is basically the driver for all the code in huffman tree. The binary file is written here

7. decoder.java

This class stores the logic of the decoding. It reads the input file as encoded.bin and the code table and generates the output file as decoded.txt. we are using here the Binary TRIE to generate the original file

<u>Algorithm for decoding:</u>

Ofor decoding we are using binary tries. We read the code table and start inserting each word into the binary trie according to the trie property. That is if code has 0 the we will go to the left node and if the code has 1 we will go to the right node. And if the node doesn't exist for that particular path we create the new nodes.
For look up we directly iterate through our trie and print the leaf node,

Complexity:
For every entry the worst case is we will have to insert every node. So the worst case time for one word is the length of the word. Let the length be l. for the m number of inputs in the code table the average time will be l*m. so the worst case complexity is O(ml). but the amortized cost will be lower since we can't get worst case for every string.

For looking up suppose we have n words in encoded bin then we can read all of the file in a single go. We keep reading bit by bit and moving through the trie and then when we hit the leaf node we print it. So the time complexity is O(n*number of average bits per word).

So the total decoding complexity is O(maxmimumof(ml,nl)).

Attributes:

static Node *root*=**null**; //root of the trie
static Node *next*=**null**: //next node of the current node

Methods:

private static void decode(String args[]) // takes encoded bin and code table as input and regenerates the code table from file. It calls the function to create the trie and generate the output
private static void createTrie(**int** value, String code): // for each code we insert a node in trie
private static void generateOutput(String encodedFilePath); //it takes the binary file path, reads it and convert it to the decoded file by looking up into the trie.

Result:

Input file size	**Binary heap**	**4-way cache optimized heap**	**Pairing heap**
62 bytes	12	10	8
69.6mb	934	650	1030
577mb	1017	766	1556

the time mentioned is in milliseconds to create a Huffman tree. These are the average of 10 iterations. So from the practical approach the 4 way cache optimized heap is fastest.

Analysis:

The fourway cache optimized heap outperforms every other heap used for the priority queue. This is possible because of the following reasons:

1. the height of the fourway heap is less as compared to the binary heap because the log base of log n in binary is 2 and for four way is 4.
2. Every operation is directly related to the height. So, since the height is less complexity of add and remove operations is reduced.
3. Unlike pairing heap, we don’t have to meld the trees while performing the delete.
4. Since the 4-way heap is implemented in such a way that all the children are in same memory heap, the runtime is drastically reduced.