Huffman Compression &Decompression

Algorithms assignment 1

NAME : Esraa Hesham ID: 4371

NAME : Mai Hossam ID: 4387

NAME : Reem Mahmoud ID: 4525

Huffman Compression &Decompression

1)description of the Implementation

A menu is printed with all possible options for the user to choose the desired operation

---- Menu ----
-- [1] to compress
-- [2] to decompress
-- [3] to compress folder
-- [4] exit

Compressing:

Creating the Huffman tree:

For each character in the file

Frequency [character]++

For each character possible

If frequency [character]>0

Create Huffman tree with charater and frequency

Add Huffman tree to a priority queue.

While priority queue size >1

A= priority queue dequeue

B= priority queue dequeue

C=a frequency + b frequency

Create Huffman tree with child trees a and b and frequency c

Final Huffman tree = priority queue dequeue.

Converting the Huffman tree to a map:

Add top Huffman tree to a tree list

Values list = list containing one empty item.

Map = empty array.

Position= 0.

While position <tree list size.

If tree list [position]is a leaf

Key = tree list[position]character

Value = values list[position]

Map add (key, value)

Else

Initial value = values list [position]

Left = initial value + 0.

Right = initial value + 1.

Add left tree to tree list.

Add right tree to tree list

Position ++

Then return map

Converting input into a compressed output:

For each character in the input

Find its new value using the map

Write the value to the output.

- The path of Both input file and output file that will contain the compressed text are given by the user.

- **handleEncodingNewText**(): calls the following functions to encode the input file and takes the choice weather to encode a File or a Folder.

- **calculateCharIntervals(nodes):** It takes the priority queue as input and Iterates the Text of the input file and starts setting each character's frequency then adding it to the Frequency HashMap with value of char and frequency as key.

- **buildTree**(**nodes**) : Extracts the 2 Nodes with the lowest frequency from the priority queue and creats a new node with a constructor that sets the new vlue with the sum of the old nodes.

- **generateCodes(nodes.peek(), "")** : It starts generating the New codes Recursively , starting from the Root of the tree which is exactly the remaining node in the priority queue, sets the value "0" to the left branch and "1" to the right branch.

- **printCodes()** : It just prints the new generated codes in the header of the output compressed file.

- **encodeText()** : It Iterates the input text and converts each byte to its binary equivalent and it's then casted to char and added to the encoded string, if it's not devisable by 8 zeros are concatenated to guarantee correct conversion.

Decompressing:

- **handleDecodingNewText**() : calls the following functions to decode the encoded file and prints the header containing the codes used and outputs the decoded text to the file that the user had already provided it's path.

- **decodeText(encodedBinary)** : It Iterates the string that is about to be decoded bit by bit and search in the Hashmap lookupFile to find it's equivalent character.

2) Data structures used for implementing the encoding techniques:

static PriorityQueue<Node> nodes = new PriorityQueue<>((o1, o2) -> (o1.value < o2.value) ? -1 : 1);
A priority queue to create a huffman tree,
When creating the huffman tree, The priority queue is used to sort out where each character should appear on the tree. The first two items are removed from the priority queue and made children to a new huffman tree with the frequency set to the sum of it's children frequencies.

static TreeMap<Character, String> codes = new TreeMap<>();
To store the new generated codes.

static TreeMap <Character,Integer>frequency=new TreeMap <>();
To store each character and it's frequency.

static TreeMap<Character, String> codes = new TreeMap<>();
To store each character and it’s code.

3)The algorithms used and its complexity

Compression

- Using Huffman Compression, the time taken to compress a file is affected by the **file size, and the number of different bytes in the file.**
- Firstly the file is traversed and the frequency of each byte is recorded. This is done char by char ,so the initial reading of the file and collecting of frequencies takes linear time, dependent on the file size.
- time taken to create the huffman tree is linearly dependent on the number of different characters in the file.
- each bit in the file is converted to a string of characters.
- performing a breadth first search on the huffman tree, while keeping track of several lists. The time taken to complete the breadth first search depends on the number of nodes in the huffman tree.
- as the file gets larger, the time becomes more dependent on the file size, and the number of different bytes becomes insignificant.

Time to run huffman compression over a file:
= O(n + b)
n = file size
b = number of different bytes in the file

Decompression

- we insert the codes into LookupFile hashMap

- we Iterate the string that is about to be decoded bit by bit and search in the Hashmap lookupFile to find it's equivalent character.

- the decompression runs in linear time, depending on the amount of data to decompress.

Time to run huffman decompression over a file:
= O(b + n)
n = compressed data size
b = number of different bytes in the decompressed file

4) the header format of the compressed file.

It contains the generated codes and their Ascii values in this format
Ascii : Generated Code.