COP 5536 Spring 2017 Programming Project Report Huffman Coding

Prepared By:

Harshit Shah

(UFID: 1211-6976)

Content

1. Goals of the project:

Part-1: To determine which among the following priority queue structures gives best performance: Binary Heap, 4-way Cache Optimized heap and Pairing Heap in order to implement Huffman codes

Part-2: To write code for Huffman Encoder and Decoder.

The encoder reads an input file that is to be compressed and generates two output files – the compressed version of the input file and the code table.

The decoder reads two input files - encoded message and code table. The decoder first constructs the decode tree using the code table. Then the decoded message can be generated from the encoded message using the decode tree.

2. Project Structure:

Encoder:

Class Encoder

Functions:

1	**main(String[] args)** The encoding algorithm is primarily divided primarily into four different sub-parts: - Build the frequency table - Building Huffman Tree - Generating code table - Encoding data by replacing each input value by its code
2	**GenerateCodes(BinHeapNode root, String str, HashMap<String, String> charCode)** To generate Huffman Codes from 4-way cache optimized heap eg: A=001
3	**isLeaf(BinHeapNode root)** To check if a node is a leaf node

Class CacOptHeap

Functions:

1	**buildHuffmanTree(HashMap<String, Integer> huff)** Primary method responsible for creating the 4-way heap and building Huffman Tree
2	**createAndBuildMinHeap(HashMap<String, Integer> huff)** Fetch values from the HashMap into an array of heap nodes and then construct the heap
3	**buildMinHeap(BinHeapNode[] minHeap)** To construct 4-way cache optimized heap from the given values in input file
4	**minHeapify(BinHeapNode[] minHeap, int idx, int cntr)**

	To ensure that the tree follows 4-way heap properties
5	**swapMinHeapNode(BinHeapNode[] minHeap, int smallest, int idx)** To swap the minimum element with its parent element
6	**extractMin(BinHeapNode[] minHeap)** To pull out the root element from the 4-way heap and then minHeapify the tree

Class BinHeapNode

Node Structure		
Name	**Type**	**Description**
data	String	Distinct values in the input file
frequency	Integer	Frequency of the values in the input string
left	BinHeapNode	Pointer to the left sibling
right	BinHeapNode	Pointer to the right sibling

Decoder:

Class Decoder

Function:

	main(String[] args)
1	The decode algorithm is divided into four different sub-parts: - Reading the code table - Constructing Huffman tree from code table - Decode .bin file bit by bit - Traverse through Huffman Tree to decode bits from Encoded Binary File

2. Performance Analysis Results:

Among the priority queue structures: Binary Heap, 4-way cache optimized heap and Pairing Heap, **4-way cache optimized heap gives the best performance**.

Average run time in seconds (executed for 10 times) for the above 3 data structures using input data of **sample_input_large.txt** are as follows:

Binary Heap	4-way Cache Optimized Heap	Pairing Heap
0.56	**0.51**	**1.40**

A 4-way cache optimized heap is better than other heaps due to quite a few reasons. Remove-min operations in a 4-way heap do 4 compares per level rather than 2 which happens in a Binary Heap. Number of levels is halved in 4-way heap. Also other operations associated with remove min operation like moving the small element up and loop iterations are halved.

3. Decoding Algorithm and Its Complexity:

Algorithm:

1. Read data and frequency values from code table into HashMap so that they may be later used to generate Huffman Tree

```
//Code Table
BufferedReader br = new BufferedReader(new FileReader(args[1]));
String line;
HashMap<String, String> huff = new HashMap<String,String>();
while ((line = br.readLine()) != null) {
    String tmp[] = line.split(" ");
    huff.put(tmp[0], tmp[1]);
}
br.close();
```

2. Construct Huffman Tree with the help of HashMap created in the earlier step

```
//Construct Huffman Tree from Code Table
BinHeapNode root = new BinHeapNode(null, null);
BinHeapNode curr;
for (Map.Entry<String, String> entry : huff.entrySet()){
    char[] charVal = entry.getValue().toCharArray();
    curr = root;
    for(int i = 0; i < charVal.length; i++){

        if(charVal[i] == '0'){
            if(curr.left != null)
                curr = curr.left;
            else{
                if(i == charVal.length - 1)
                    curr.left = new BinHeapNode(entry.getKey(), null);
                else
                    curr.left = new BinHeapNode(null, null);
                curr = curr.left;
            }
        }
        else{
            if(curr.right != null)
                curr = curr.right;
            else{
                if(i == charVal.length - 1)
                    curr.right = new BinHeapNode(entry.getKey(), null);
                else
                    curr.right = new BinHeapNode(null, null);
                curr = curr.right;
            }
        }

    }
}
```

3. Decode binary file into byte array and convert it to String

```
//Decode Binary File
byte[] bFile = Files.readAllBytes(Paths.get(args[0]));
StringBuilder sb = new StringBuilder();
for (int i=0; i < bFile.length;i++)
{
    sb.append(String.format("%8s", Integer.toBinaryString(bFile[i] & 0xFF)).replace(' ', '0'));
}
String decodedString = sb.toString();
```

4. Starting from the root node in Huffman Tree, traverse the tree character by character from the string until leaf node is reached. Once leaf node is reached, again start traversing from the root node until end of string s reached.

```
//Traverse through Huffman Tree to decode bits from Encoded Binary File
BufferedWriter buff = new BufferedWriter(new FileWriter("decoded.txt"));
int j=0;
while(j<decodedString.length())
{
    BinHeapNode temp = root;
    while(!(temp.left==null && temp.right==null))
    {
        if(decodedString.charAt(j)=='0')
        {
            temp = temp.left;
        }
        else
        {
            temp = temp.right;
        }
        j++;
    }
    buff.write(temp.data);
    buff.newLine();
}
buff.close();
```

Complexity:

As can be seen from the above algorithm, time complexity of decoding algorithm would be **linear in no. of bits**