PROGRAMMING PROJECT
HUFFMAN ENCODER-DECODER

I. INTRODUCTION:

This report sheds light on the design and analysis of the Huffman encoder and decoder for Software Client-Toggle, who decided to use Huffman coding to send data across to their servers in compressed format. The software specifications of the project are:

Programming Language: Java
IDE: Eclipse
Compile script: makefile ('make clean' command deletes the .bin and .class files)
Run: java encoder <inputfile>
java decoder <encode.bin> <code_table.txt>

The key points that will be covered by the report are the Huffman encoder, decoder and the tree structure along with the analysis.

II. PROGRAM STRUCTURE:

1) Code: encoder.java

This is the implementation of the Huffman encoder in Java. The encoder class will have to read the input file as a command line argument in 'args[0]'. The file will be read line by line and for all the input data, its corresponding frequency will be calculated and a frequency table will by computed internally using a HashMap implementation. Once this is done, we compute a min heap for the data. In this step, a detailed analysis of the heap structure was done using Pairing heap, 4-way cache optimized heap and binary heap (discussed in section III) and the best (i.e. 4-way cache optimized heap) was chosen to proceed further with tree generation.
The following methods are used in this stage:

- **void huffman(freq_map, filename)** takes the frequeny table as the input. It adds all the entries one by one to the 4-way heap which heapifies itself everytime a node is inserted(discussed in section II. 4) Thus the heap is formed.
- **Node constructHeap(HashMap<String, Integer> freq_map)** is then called. This function removes the minimum 2 nodes(structure discussed in section II.3) of the heap and begins creation of the Huffman tree. After pulling out two minimum nodes it adds an internal node having frequency equal to the sum of the frequencies of the other two nodes.
- **void buildCodes(Node root1, String repr,File codeTableFile,BufferedWriter output)** is the next function to be called. This method creates the Huffman code for the corresponding file data by reading the Huffman tree. This function also writes the code_table.txt file which will be further used for decoding purposes.
- **void encode(BufferedReader bufferedReader, BufferedOutputStream buffOpStream)** method then creates the encoded.bin file. This method reads the input file, matches the

data with the code in the code table formed in the previous step. Once that is done, the codes are converted into byte array and eventually written to the encoded.bin file in order to achieve a high compression rate.

The encoded.bin and the code_table.txt files formed in this stage will be transferred over the network to the server which will then use the files for decoding purpose.

2) Code: decoder.java

This is the implementation of the Huffman decoder in Java. The decoder class will have to read 2 input files as a command line arguments in 'args[0]' and 'args[1]' namely – encoded.bin and code_table.txt. The decoder will first read the code_table.txt i.e the code table and form a Huffman tree again that will be used in future for decoding the bin file. The algorithm used for reconstructing the Huffman tree is the logic for **Binary Trie data structure.** The complexity for the look up in the data structure is **O(m)** where m is the length of the lookup string. The complexity of creating a trie is **O(m*l)** where m is the number of words, and l is an average length of the word. We need to perform l lookups on the average for each of the m words in the set. After this step, the decoder will read the 'encoded.bin' file byte by byte and traverse through the Huffman tree to find the corresponding key and print it in the decoded file.

The following methods are used in this stage:

- **void buildTree(String value, String code)** method takes the value and the corresponding code from the code table- line by line as input. The code string is traversed one character at a time to check if it is '0' or '1'. The tree is then generated one node at a time by creating a node in the left if the character is '0' else a node to the right if the character is '1'. The result of this method is a fully formed Huffman tree.
- **void generateOutput(String encodedFilePath)** method reads the 'encoded.bin' file in a byte by byte format, converts the bytes into a bit string and traverses the tree for each bit. Once it hits the leaf nodes the data is then extracted from the tree and written to the output file.

3) Code: Heap.java

Heap.java maintains the implementation of the Node of the Huffman Tree in Java. It acts like an interface for each Node.

```
public class Node {
		String data;
		int freq;
		Node left;
		Node right;
		Node(){};
		Node(String data,int freq){
			this.data = data;
			this.freq = freq;
		}
}
```

4) Code: FourWayHeap.java

FourWayHeap.java extends the class Heap.java to interact with the encoder and decoder. This implementation comes handy while maintaining the sorted order i.e the mean heap of the data and frequency during step 1 of section II.1. The implementation for the same is as discussed below:

```
class FourWayHeap  extends Heap
{
    int heapStart=3;
    private int d;
    private int heapSize;
    private Node[] heap;
    int n=3;
    int capacity=8;
    int numChild=4;
    public void insert(Node x){…}
    public Node delete(int ind){…}
    private void heapifyUp(int childInd){…}
    public void heapifyDown(int ix) {…}
}
```

- **public void insert(Node x){…}** method allows you to insert a new Node in the heap. Every time an insertion happens, the heapifyUp() is called.
- **public Node delete(int ind){…}** method allows you to delete a new Node from the heap. Every time an insertion happens, the heapifyDown() is called.
- **private void heapifyUp(int childInd){…}-** When a node is inserted the heap size is increased and the node is added to the last. Now, the structure needs to be re-sorted to accommodate the new node. This method takes care of the re-sorting towards the start of the heap.
- **public void heapifyDown(int ix)** {…}- When a node is deleted i.e the minimum node is extracted from the root, there may be an imbalance caused as the root doesn't exist anymore. The structure, therefore needs to be re-sorted to choose a new node from among its children. This method takes care of the re-sorting towards the bottom of the heap.

III. ANALYSIS AND RESULT:

As per the understanding of the concepts the following is the theoretical amortized complexity of the given data structures implemented for analysis is $O(\log_d n)$ where d is the degree of children.

However, on the practical implementation and execution of the 3 ways to implement heap i.e. Binary heap, Pairing heap and 4-way cache optimized heap, 4-way cache optimized turned out to be the fastest. The following is the performance if we consider the average time taken after executing 10 back to back iterations of the above-mentioned implementation:

Note : All results shown are values obtained by running the program for Huffman Tree generation on Thunder and the time is in milliseconds.

Input File Size	Binary Heap	4-way Cache optimized Heap	Pairing Heap
62 bytes	4	3	5
69.6MB	504	415	830
766MB	1556	1429	1624

IV. CONCLUSION:

It can be concluded from the experiments done above, 4-way cache optimization heap has a better performance than the other two heaps in this scenario. This can be due to the following reasons:

- The height of the tree is half that of the binary tree
- So, worst case insert will have to move up to half the levels that in binary and pairing heaps
- Although removeMin() has 4 comparisons to do, which is double that of pairing and binary heaps but the levels are less
- So, other operations with respect to removeMin() are also halved
- The siblings are going to fall in the same cache lines internally therefore, the cache misses are also reduced to ~$\log_4 n$
- Therefore, the speed has also drastic change with increasing record size

_____