ADVANCED DATA STRUCTURES (COP 5536) SPRING 2017

PROGRAMMING PROJECT REPORT

HUFFMAN ENCODING AND DECODING

SUBMITTED BY:

Name: Dhiraj V. Borade

UFID: 45958142

UF Email ID: dhirajborade@ufl.edu

STEPS TO RUN THE PROGRAM

The project has been written in java and can be compiled using standard JDK 1.8.0_60, with javac compiler.

Unzip the submitted folder. The folder contains the project files (.java files) and the Makefile. The input files must be placed in the project directory before running the program or input file path to be specified in <input_file_name>.

Output file i.e. decoder.txt is created in the project directory.

To compile in the Unix environment, run the following command after changing the directory to the project directory.

1. **"cd"** to the current directory of the Source Files.
2. Perform the following the operations:
    a. $ make clean
    b. $ make
    c. $ java encoder <input_file_name>
    d. $ java decoder <encoded.bin> <code_table.txt>

Operation Description:

$ make clean: All the previously generated object files and executables are cleaned up.

$ make: All the .java files in the source folder are compiled and .class files are generated

$ java encoder <input_file_name>: Program for Huffman Encoder is executed with <input_file_name> as the input arguments. File <input_file_name> is compressed using Huffman Encoding into encoded.bin and code_table.txt is generated.

$ java decoder <encoded.bin> <code_table.txt>: Output files from encoder are provided as inputs to decoder and the original file is decoded as decoded.txt

PROBLEM DESCRIPTION

This project requires the implementation of Huffman Encoder and Decoder to reduce the size of the data that is to be transferred to server.

1. **Huffman Coding:**

   This part includes generating Huffman Codes, which consist of generating frequency table, which is used to generate Huffman tree and using this Huffman tree, code table is generated. Code table consist of all the elements (no duplicates) from input file and the corresponding Huffman code generated for the element.
   Huffman tree is constructed using one of the priority queue structures viz., Binary Heap, 4-Way Cache Optimized Heap and Pairing Heap.
   Performance Analysis is performed for those priority queue structures and the best priority queue structure to be used for Huffman code generation.
   Huffman code generation takes frequency table as input and outputs code table.

2. **Huffman Encoder:**

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

3. **Huffman Decoder:**

   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.

The following structures are used for implementation:

1. Binary Heap, 4 Way Cache Optimized Heap and Pairing Heap for generating Huffman Codes.
2. TreeMap for Frequency table: Key => Element and Value => Frequency.
3. HashMap for Code Table: Key => Element and Value => Huffman Code.

HUFFMAN CODING

Huffman coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters. The most frequent character gets the smallest code and the least frequent character gets the largest code. The variable-length codes assigned to input characters are Prefix Codes, means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bit stream.

There are mainly two major parts in Huffman Coding:

1. Build a Huffman Tree from input characters.
2. Traverse the Huffman Tree and assign codes to characters.

Steps to build Huffman Tree:

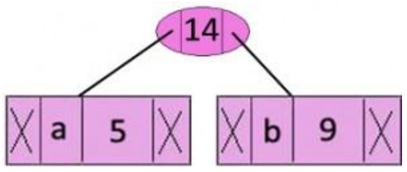
Input is TreeMap of unique characters as TreeMap key along with their frequency as TreeMap value of occurrences and output is Huffman Tree.

1. Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root).
2. Extract two nodes with the minimum frequency from the min heap.
3. Create a new internal node with frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.
4. Repeat Step 2 and Step 3 until the heap contains only one node. The remaining node is the root node and the tree is complete.

   Let us understand the algorithm with an example:

Character	**Frequency**
a	5
b	9
c	12
d	13
e	16
f	45

Step 1. Build a min heap that contains 6 nodes where each node represents root of a tree with single node.



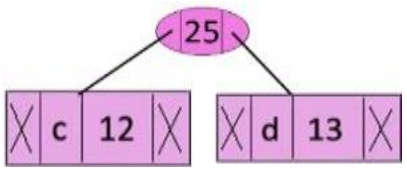
Step 2 Extract two minimum frequency nodes from min heap. Add a new internal node with frequency 5 + 9 = 14.

Now min heap contains 5 nodes where 4 nodes are roots of trees with single element each, and one heap node is root of tree with 3 elements

Character	Frequency
c	12
d	13
Internal Node	14
e	16
f	45

Step 3: Extract two minimum frequency nodes from heap. Add a new internal node with frequency 12 + 13 = 25.

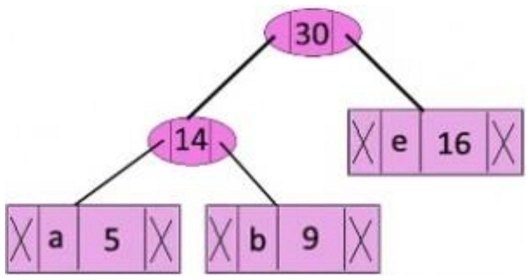
Now min heap contains 4 nodes where 2 nodes are roots of trees with single element each, and two heap nodes are root of tree with more than one nodes.



Character	Frequency
Internal Node	14
e	16
Internal Node	25
f	45

Step 4: Extract two minimum frequency nodes. Add a new internal node with frequency 14 + 16 = 30.

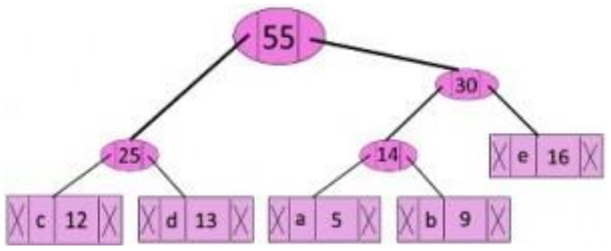
Now min heap contains 3 nodes.



Character	Frequency
Internal Node	25
Internal Node	30
f	45

Step 5: Extract two minimum frequency nodes. Add a new internal node with frequency 25 + 30 = 55.

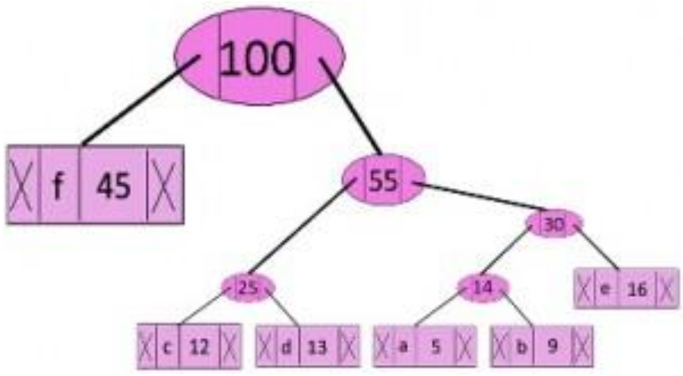
Now min heap contains 2 nodes.



Character	Frequency
f	45
Internal Node	55

Step 6: Extract two minimum frequency nodes. Add a new internal node with frequency 45 + 55 = 100.

Now min heap contains only one node.

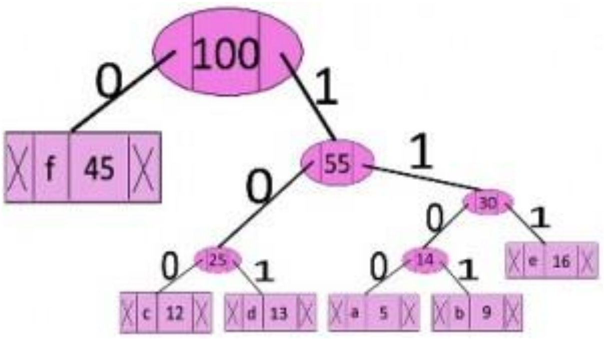


Character	Frequency
Internal Node	100

Since the heap contains only one node, the algorithm stops here.

Steps to print codes from Huffman Tree:

Traverse the tree formed starting from the root. Maintain an auxiliary array. While moving to the left child, write 0 to the array. While moving to the right child, write 1 to the array. Print the array when a leaf node is encountered.



The codes are as follows:

Character	Huffman Code
f	0
c	100
d	101
a	1100
b	1101
e	111

Time complexity: O(nlogn) where n is the number of unique characters. If there are n nodes, deleteMin() is called 2*(n – 1) times. deleteMin() takes O(logn) time as it calls heapifyDown(). So, overall complexity is O(nlogn).

PERFORMANCE ANALYSIS OF PRIORITY QUEUE STRUCTURE

Huffman tree is constructed using Priority Queue Structures. Priority Queue Structures available are Binary Heap, Four Way Cache Optimized Heap and Pairing Heap. Performance analysis is performed for those three structures.

1. Binary Heap:

Every Binary tree with **N** external nodes defines a code set for **N** messages. Start with a collection of external nodes in a set **S**, each one of the given weights. Each external node defines a different tree. We use the greedy approach to select two trees with minimum weight from a set **S**, we combine them by making them children of a new root node and place the node back in the set **S**, the weight of the new tree is the sum of the weights of the individual trees. Repeat until one tree is left.

Huffman_Tree_Binary_Heap() takes 521 mS to execute (for 1 iteration)
Huffman_Tree_Binary_Heap() takes 3432 mS to execute (for 10 iterations)
Huffman_Tree_Binary_Heap() takes 42240 mS to execute (for 100 iterations)

2. Four Way Cache Optimized Heap:

N node tree whose degree is 4. Number of nodes in breadth first manner with root being numbered 1. For a node tree with degree **d**

$$\boldsymbol{Parent(i) = \frac{ceil(i-1)}{d}}$$

Children are d*(i-1) + 2, …., min {d*i+1, n}

When we use a node tree with degree 4 and root being numbered 1, the siblings are in 2 cache lines when mapped into cache aligned array. We optimize it by shifting 4-heap by 2 slots which results, siblings being in the same cache line.

Huffman_Tree_FourWay_Heap() takes 484 mS to execute (for 1 iteration)
Huffman_Tree_FourWay_Heap() takes 2804 mS to execute (for 10 iterations)
Huffman_Tree_FourWay_Heap() takes 36840 mS to execute (for 100 iterations)

3. **Pairing Heap:**

Node Structure:

- Child: Pointer to first node of children list.
- Left and Right Sibling:
    - Used for doubly linked list (not circular) of siblings.
    - Left pointer of first node is to parent.
    - X is first node in list iff x.left.child = x.
- Data
- Compare-Link Operation:
    - Compare roots.
    - Tree with largest root becomes leftmost child

Initially each node is a tree, we perform the Compare-Link operation and form a tree. Then we pull out the root i.e. the minimum element. We perform 2-Pass Scheme with the child nodes, and remove the next min element and combine both the elements and the combined result node is added back to the lists of trees.

Huffman_Tree_Pairing_Heap() takes 796 mS to execute (for 1 iteration)
Huffman_Tree_Pairing_Heap() takes 4962 mS to execute (for 10 iterations)
Huffman_Tree_Pairing_Heap() takes 49624 mS to execute (for 100 iterations)

Conclusion: Thus on the basis of the above performance analysis, Four Way Cache Optimized Heap is better for Huffman Tree generation and thus was used for Huffman Encoding and Decoding.

FUNCTION PROTOTYPES AND STRUCTURE OF THE PROGRAM

The project has a total of 10 classes encoder.java, decoder.java, DecoderNode.java, Huffman.java, HuffmanBinaryNode.java, HuffmanFourWayNode.java, HuffmanPairingNode.java, BinaryHeap.java, FourWayHeap.java and PairingHeap.java with encoder.java and decoder.java are the main functions.

1. **encoder.java:**

   This is the main function for Huffman Encoding. It takes <input_file_name> as input arguments and outputs its compressed version as encoded.bin and also generates code_table.txt.

2. **decoder.java:**

   This is the main function for Huffman Decoding. It takes <encoded_file_name> <code_table_file_name> as its input arguments and generates the decoded file as decoded.txt.

3. **DecoderNode.java:**

   This class describes the structure of the node to be inserted in the Decoder Tree.
   The class contains the following variables and pointers:
   Pointers:
   a. leftChild: Pointer to the left child of the node.
   b. rightChild: Pointer to the right child of the node.

   Variables:
   a. key: Element.
   b. code: Huffman code for the element.

   We have a constructor in this class that initializes the pointers to the current node.

4. **Huffman.java:**

   This class is the central working part for the entire project. The function prototypes of Huffman class are given below:

   a. public void generateWordCount(String inputArgs):

      Function to generate the frequency table.

   b. public void Huffman_Tree_Binary_Heap():

      Function to generate Huffman Codes using Binary Heap.

   c. public void Huffman_Tree_FourWay_Heap():
      Function to generate Huffman Codes using Four Way Heap.

d. public void Huffman_Tree_Pairing_Heap():

   Function to generate Huffman Codes using Pairing Heap.

e. public void generateCodeHuffmanBinaryNode (HuffmanBinaryNode huffmanBinaryNode, FileWriter writer):

   Function to write Huffman Codes generated using Binary Heap to code_table.txt

f. public void generateCodeHuffmanFourWayNode (HuffmanFourWayNode huffmanFourWayNode, FileWriter writer):

   Function to write Huffman Codes generated using Four Way Heap to code_table.txt

g. public void generateCodeHuffmanPairingNode (HuffmanPairingNode huffmanPairingNode, FileWriter writer):

   Function to write Huffman Codes generated using Pairing Heap to code_table.txt

h. public long getFileSize(String filename):

   Function to calculate the file size in Kilobytes.

5. **HuffmanBinaryNode.java:**

This class describes the structure of the node to be inserted in the Binary Heap.
The class contains the following variables and pointers:
Pointers:

a. leftChild: Pointer to the left child of the node.
b. rightChild: Pointer to the right child of the node.

Variables:

a. key: Element.
b. frequency: Huffman code for the element.
c. isLeaf: To check whether the node is leaf or not.
d. incomingEdge: To check whether the node is a leftChild (0) or rightChild (1).

We have a constructor in this class that initializes the pointers to the current node.

6. **HuffmanFourWayNode.java:**

This class describes the structure of the node to be inserted in the 4 Way Heap.
The class contains the following variables and pointers:
Pointers:

a. leftChild: Pointer to the left child of the node.
b. rightChild: Pointer to the right child of the node.

Variables:

a. key: Element.
b. frequency: Huffman code for the element.
c. isLeaf: To check whether the node is leaf or not.
d. incomingEdge: To check whether the node is a leftChild (0) or rightChild (1).

We have a constructor in this class that initializes the pointers to the current node.

7. **HuffmanPairingNode.java:**

This class describes the structure of the node to be inserted in the Pairing Heap. The class contains the following variables and pointers:
Pointers:

a. leftChild: Pointer to the left child of the node.
b. rightChild: Pointer to the right child of the node.
c. nextSibling: Pointer to the next sibling of the node.

Variables:

a. key: Element.
b. frequency: Huffman code for the element.
c. isLeaf: To check whether the node is leaf or not.
d. incomingEdge: To check whether the node is a leftChild (0) or rightChild (1).

We have a constructor in this class that initializes the pointers to the current node.

8. **BinaryHeap.java:**

This class generates Minimum Binary Heap with the following functions:

```
public void insert(HuffmanBinaryNode huffmanBinaryNode)

public HuffmanBinaryNode deleteMin()

private void heapifyUp(int childInd)

private void heapifyDown(int ind)
```

We have a constructor in this class that initializes the Binary Heap with 0 nodes.

9. **FourWayHeap.java:**

This class generates Minimum Four Way Heap with the following functions:

```
public void insert(HuffmanFourWayNode huffmanFourWayNode)

public HuffmanFourWayNode deleteMin()

private void heapifyUp(int childInd)
```

private void heapifyDown(int ind)

We have a constructor in this class that initializes the Four Way Heap Cache Optimized Heap with 3 nodes having values of -1.

10. PairingHeap.java:

This class generates Minimum Pairing Heap with the following functions:

public HuffmanPairingNode insert(HuffmanPairingNode huffmanPairingNode)

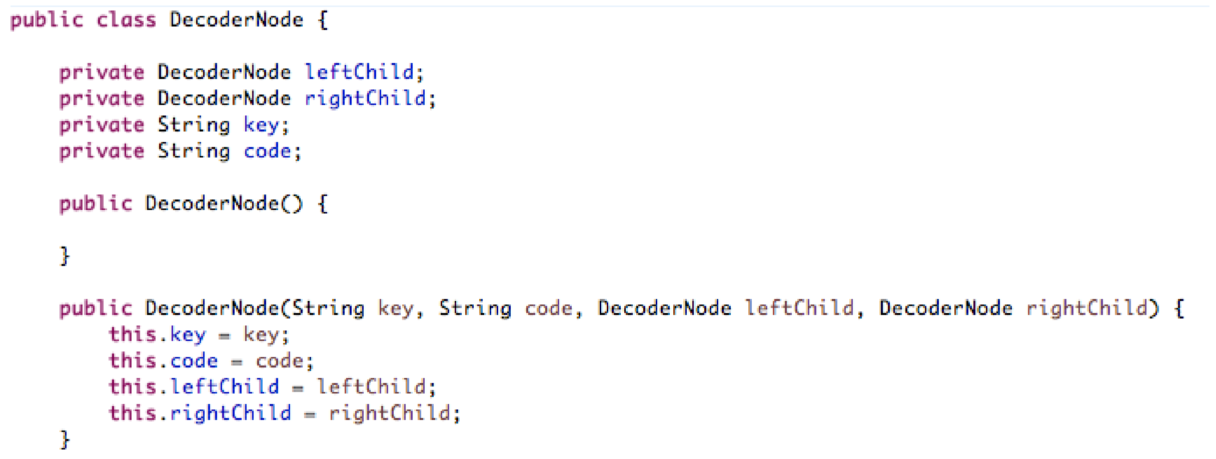
public HuffmanPairingNode deleteMin()

private static HuffmanPairingNode compareAndLink(HuffmanPairingNode first, HuffmanPairingNode second)

private HuffmanPairingNode combinesiblings(HuffmanPairingNode firstSibling)

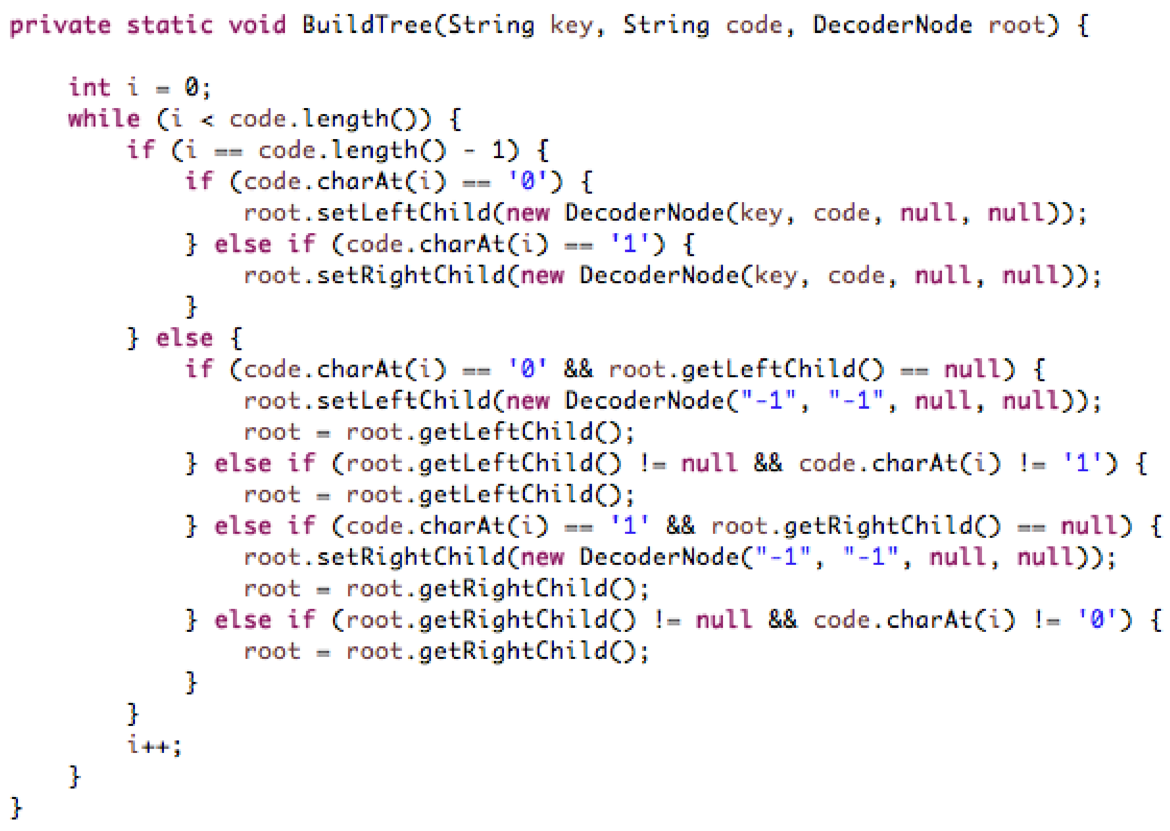
DECODING ALGORITHM

In order to implement the decoding from the compressed encoded.bin, Decoder tree is created with each node having the following structure:



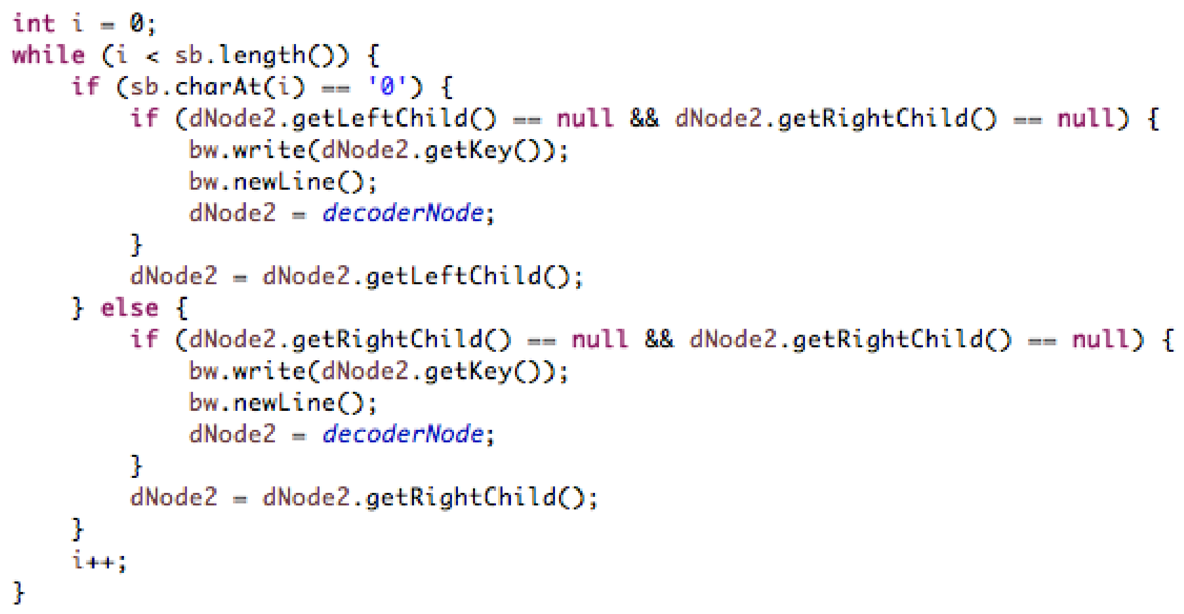
```java
public class DecoderNode {

    private DecoderNode leftChild;
    private DecoderNode rightChild;
    private String key;
    private String code;

    public DecoderNode() {

    }

    public DecoderNode(String key, String code, DecoderNode leftChild, DecoderNode rightChild) {
        this.key = key;
        this.code = code;
        this.leftChild = leftChild;
        this.rightChild = rightChild;
    }
```

In order to create the Decode tree, the given code table is read line by line. An empty root node is created then we check if the left child is null and code bit is 0 then we attach a left child to it else the program traverses left. If bit value is 1 we create a right child of it, if it is null else program traverses right, if this is the last bit of respective code value and we assign the key associated with it to the respective node. The structure is as follows:



```java
private static void BuildTree(String key, String code, DecoderNode root) {

    int i = 0;
    while (i < code.length()) {
        if (i == code.length() - 1) {
            if (code.charAt(i) == '0') {
                root.setLeftChild(new DecoderNode(key, code, null, null));
            } else if (code.charAt(i) == '1') {
                root.setRightChild(new DecoderNode(key, code, null, null));
            }
        } else {
            if (code.charAt(i) == '0' && root.getLeftChild() == null) {
                root.setLeftChild(new DecoderNode("-1", "-1", null, null));
                root = root.getLeftChild();
            } else if (root.getLeftChild() != null && code.charAt(i) != '1') {
                root = root.getLeftChild();
            } else if (code.charAt(i) == '1' && root.getRightChild() == null) {
                root.setRightChild(new DecoderNode("-1", "-1", null, null));
                root = root.getRightChild();
            } else if (root.getRightChild() != null && code.charAt(i) != '0') {
                root = root.getRightChild();
            }
        }
        i++;
    }
}
```

Once the code tree is generated we read the stream and traverse the tree. When the program reaches the leaf node the associated key value is written into decode.txt. The function associated with this is as follows:



```java
int i = 0;
while (i < sb.length()) {
    if (sb.charAt(i) == '0') {
        if (dNode2.getLeftChild() == null && dNode2.getRightChild() == null) {
            bw.write(dNode2.getKey());
            bw.newLine();
            dNode2 = decoderNode;
        }
        dNode2 = dNode2.getLeftChild();
    } else {
        if (dNode2.getRightChild() == null && dNode2.getRightChild() == null) {
            bw.write(dNode2.getKey());
            bw.newLine();
            dNode2 = decoderNode;
        }
        dNode2 = dNode2.getRightChild();
    }
    i++;
}
```

Complexity of the Decoder Algorithm:

Tree is traversed from root to leaf to find the element associated with the Huffman Code.
All the bits from encoded file are used to find the element present in the leaf of the Decoder Tree.
Therefore,

N => Number of Encoded Bits

Worst Case Complexity = O(N)

CONCLUSION:

- The decoded output file and the original input file has the same content and same size, this concludes that the Huffman Encoding and Decoding was implemented successfully.
- Also, reduction in size of encoded file from that of original input file was more than 60%.
- As per the algorithm and structure of the code implemented, Four Way Cache Optimized Heap is the fastest priority queue structure out of the three implemented.