Name: AMAR NATH VAID
UFID: 3447-1121
Email: amar.vaid@ufl.edu

Compiler used: javac

Platform/OS	***Compiler***	***Test Result***
MacOSX	javac	Test passed
Linux (thunder server)	javac	Test Passed

Steps to execute the project in Linux Environment using g++ with makefile:

1. Copy all the files including the make file in a separate directory
2. Copy the data file (test file in case of user input to this directory too)
3. Run make
4. The project is ready to be executed. Type in *java encoder <input_file_name>*
5. Output files encoded.bin and code_table.txt are generated.
6. Type in *java decoder <encoded_file_name> <code_table_file_name>*

PART 1: HUFFMAN CODING

A 4-way cache optimized heap was used for the Huffman Code Program for subsequent parts of the project. The Huffman Code Program will take a frequency table as input and gives a code table as the output. For a symbol, the tree is traversed starting from the root node and moving down a level for every bit of the symbol. If a node doesn't exist, a new node is added. At the last bit of the symbol, the message field of the node is assigned the message in the symbol.
If n unique symbols exist in code_table.txt., the build function to build the code table undergoes n iterations. And if the symbols have at most l length, the Huffman Tree is built in O(nl).

PART 2: ENCODER

The encoder reads an input file that is to be compressed and generates two output files – the compressed version of the input file(encoded.bin) and the code table(code_table.txt).

PART 3: DECODER

The output of the previous part are used as input to this part which are encoded.bin and code_table.txt. The code table is used by the decoder to construct the decode tree. Then, using this decode tree, a decoded message (decoded.txt) is generated from the encoded message.

Algorithm used for the decoder:

The decoder decodes the encoded file by first creating the decode tree using the code table from the previous part. The encoded.bin file was decoded to retrieve the original message. The binary format of the encoded.bin file makes it simple to store it in an byte array. A loop is used to run over all the byes once will retrieve the data. One bit at a time is taken from every byte using masking and then we go down the tree. On the occurrence of an external node, the message stored it is retrieved and written to decoded.txt with pointer set towards the root. There are chances of using all 8 bits for a byte. In this case, the next byte is taken and the traversal is continued.

If the height of the tree is h, the maximum length of symbol present in the code table is the same as h. Hence, to encounter an external node, at most h comparisons are made for retrieving a symbol. The number of messages (lines) encoded in encoded.bin decide the time complexity of the algorithm.
Hence, T(n) = O(number of messages in the encoded file * height of the tree i.e. h)

CODE DESCRIPTION

There are three source files:

1. **Huffman.java:** This file has various classes, each of which performs a specific function. The class Node.java is made to define the structure of a node in the heap. It has setter and getter methods to set and get its members. **PairNode.java** defines the structure of a paired Node used in Pairing heap. **BinaryHeap.java** is the implementation of a binary Heap with various functions to insert in a heap, find the minimum and remove it etc. Also, there are classes for Pairing heap and 4-way optimized heap. Then is the **Huffman.java** class which takes the input file, makes the Huffman Tree and the frequency table from the input and then calculates the runtime for the 3 data structures. From this time-estimation, it was found that 4-way cache optimized heap was the best and thus, it was used in the encoder and thus the next source file.

2. **encoder.java:** It takes the input file, applies the function **makeFrequencyTable** to make the frequency table and then makes the Huffman tree using the function **huffmanway4Heap.** Code table is generated using the **fillCodeTable** and **encoded.bin** is generated by converting the input data to binary using the code table. It uses the function **getBinary** to convert it.

3. **decoder.java:** It has three main function. **addNode**, **huffmanConstruction** and **main**. It uses encoded.bin and the code_table.txt to decode data. **huffmanConstruction** takes a symbol and traverses the tree. It drops one level for each symbol element and adds a node if it does not exist using the function **addNode**.

Function declarations:

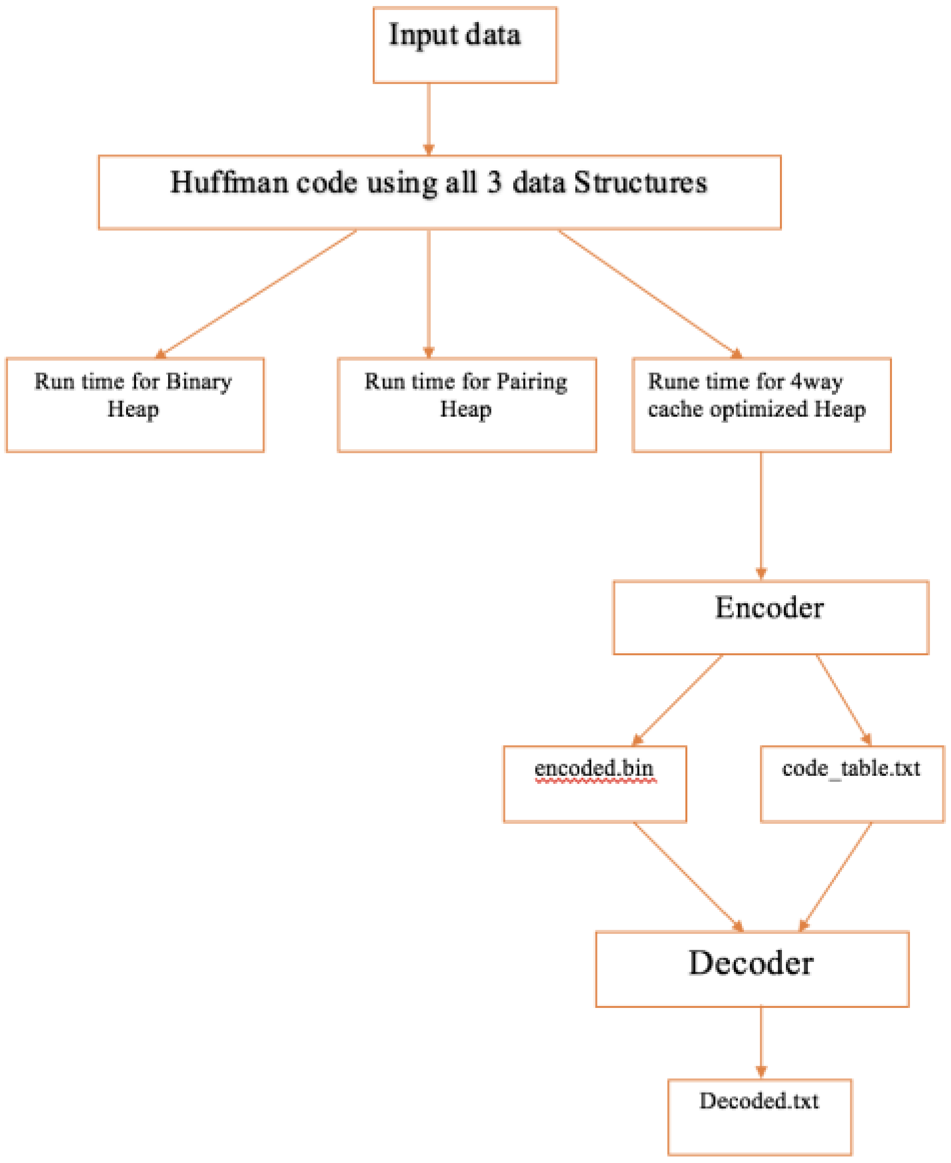
Enoder.java
```
private static void fillCodeTable(Node, String, String[] , BufferedWriter)
private static byte[] getBinary(ArrayList<Integer>, String[])
```

Decoder.java
```
public static void addNode(Node, int ,String )
public static Node huffmanConstruction(String)
```

Huffman.java

```
public static Node huffman4wayHeap(int[] frequencyTable)
public static Node huffmanPairingHeap(int[] frequencyTable)
public static Node huffmanBinaryHeap(int[] frequencyTable)
public static int[] makeFrequencyTable(ArrayList<Integer> inputs)
public static void generateHuffman(Node node, String bits)
```

FLOW OF THE CODE

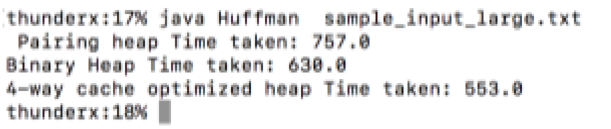

Input data
Huffman code using all 3 data Structures
Run time for Binary Heap
Run time for Pairing Heap
Rune time for 4way cache optimized Heap
Encoder
encoded.bin
code_table.txt
Decoder
Decoded.txt

PERFORMANCE ANALYSIS

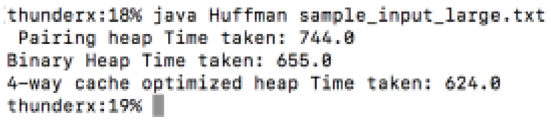
Three priority queues were implemented in the first part to generate Huffman Trees and the run time was measured using the test file sample_input_large.txt .
The priority queues implemented were:

-
- Binary heap
- Pairing heap
- 4-way cache optimized heap

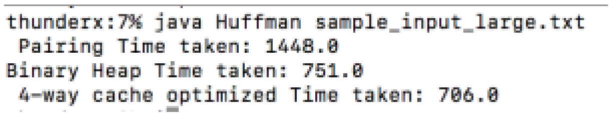
All three abovementioned priority queues were used to generate Huffman Tree. The running time was averaged for 10 runs. The performance measure of the three priority queues has been shown by running a program that displays the runtime for each of them. Also they are depicted using a graphical representation. 5 trials were conducted.



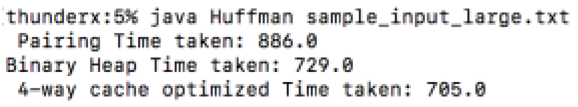
```
thunderx:17% java Huffman  sample_input_large.txt
 Pairing heap Time taken: 757.0
Binary Heap Time taken: 630.0
4-way cache optimized heap Time taken: 553.0
thunderx:18%
```



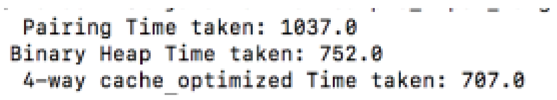
```
thunderx:18% java Huffman sample_input_large.txt
 Pairing heap Time taken: 744.0
Binary Heap Time taken: 655.0
4-way cache optimized heap Time taken: 624.0
thunderx:19%
```



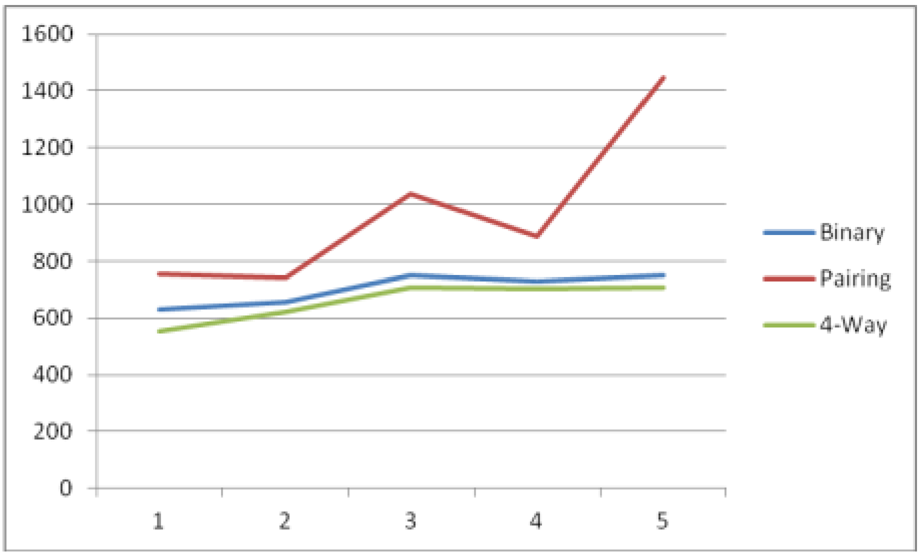
```
thunderx:7% java Huffman sample_input_large.txt
 Pairing Time taken: 1448.0
Binary Heap Time taken: 751.0
 4-way cache optimized Time taken: 706.0
```



```
thunderx:5% java Huffman sample_input_large.txt
 Pairing Time taken: 886.0
Binary Heap Time taken: 729.0
 4-way cache optimized Time taken: 705.0
```



```
 Pairing Time taken: 1037.0
Binary Heap Time taken: 752.0
 4-way cache_optimized Time taken: 707.0
```



Looking at the graph, it is clear that,

- 4-Way Heap has better performance than binary heap. This is because a 4-way heap uses half the comparisons as compared to a binary heap. Pairing Heap gives the worst performance.

Hence it was concluded that the 4-way heap cache optimized heap gave the best performance because of the least runtime. Hence 4-way cache optimized heap was used, it was chosen for the Huffman Code Program for subsequent parts of the project.