

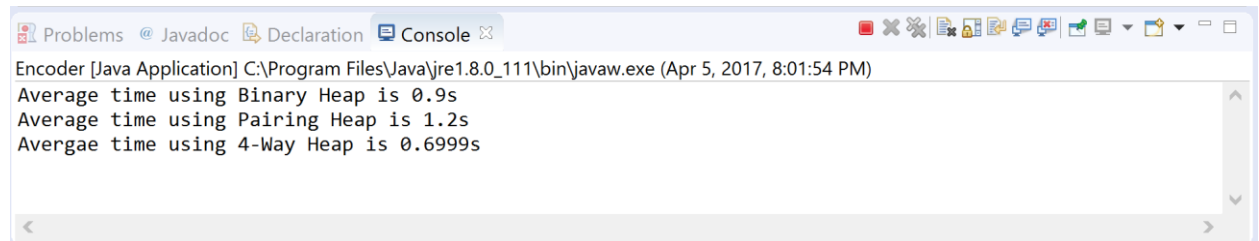
# Advanced Data Structures (COP5536) SPRING 2017

## Programming Project Report

Ashvini M Patel  
UFID: 47949297

## PART 1: Performance Evaluation Results

Average time taken for Huffman tree creation for **10 iterations** with the **sample\_input\_large.txt** file provided were as follows:



```

Encoder [Java Application] C:\Program Files\Java\jre1.8.0_111\bin\javaw.exe (Apr 5, 2017, 8:01:54 PM)
Average time using Binary Heap is 0.9s
Average time using Pairing Heap is 1.2s
Average time using 4-Way Heap is 0.6999s
  
```

Compared to using Binary heap, a cache optimized 4-way heap reduced the number of cache misses while fetching the children of a given node. Also, the height of the tree is smaller for a 4-way heap.

Compared to using pairing heap, the 4-way heap gave better results because not many pointer changes are involved for an extract min operation.

As the implementation of **4-way heap** gave the fastest performance amongst the three data structures, the Huffman Encoder and Decoder were built using 4-way heap.

## Part 2: Function Prototypes and Program Structure

Class: **FreqTable**

- public HashMap<String,Long> **GenFreqTable**(File f)  
The FreqTable class generates the frequency table for the given input text file in the form of a HashMap.

Class: **FourWayHeap**

This class implements the 4-way heap data structure.

- public static FreqDatum **removeMin()** : this function extracts the node with the least frequency and returns it
- public static void **heapify** (int position) : this function constructs the heap in such manner that the element with the minimum frequency is at the root.
- public FreqDatum **buildTree**(HashMap<String,Long> freqtable) : this function takes the frequency table in the form of a HashMap and builds a Huffman tree.

### Class: **FreqDatum**

The FreqDatum class denotes the cumulative frequency node, generated during the creation of the Huffman Tree. Members of this class include the data, it's frequency (set to -1 for non-leaf nodes), left child & right child.

### Class : **CodeTable**

- public HashMap<Integer,StringBuilder> **GenCodeTable** (FreqDatum fd): The CodeTable class generates the code table for the given frequency table in the form of a HashMap. The HashMap contains codes for the given data . The function that generates this HashMap is GenCodeTable.

### Class: **HeapTraverse**

- public void **Traverser**(FreqDatum Root,HashMap<Integer, StringBuilder> hMap, ArrayList<String> arr) : The HeapTraverse class defines the function Traverser which traverses the Huffman Tree to build the codes for each data value in the input file. This function is called recursively, and it begins by processing the root of the tree and branching to the left and right child (if they exist).

### Class: **Encoder**

The Encoder class is responsible for reading the input file and generating the Frequency Table. Then it calls the CodeTable class object to generate **codetable.txt** which contains the code table. Then it uses the generated code table to write **encoded.bin** file.

### Class: **DecoderNode**

This class forms the data structure used by the Decoder class. It's members include the data, it's corresponding code and it's left and right children.

### Class: **Decoder**

- void **decodeTree**(ArrayList<DecoderNode> DecoderNode): This function constructs the decode tree using codetable.txt with nodes specified by the DecoderNode class.
- void **decoder**(String BINFILE): This function takes the encoded.bin file generated by the Encoder class and creates the decoded.txt file containing the output of the decoding process.

## **PART 3: Decoding Algorithm and its Complexity**

First the decoder tree is built using the code table generated by the encoder. The tree is designed such that the leaves correspond to a data element. The non-leaf nodes of the tree determine the path to the “data node”. The path is created as such: For each bit in the codeword for an element if a ‘0’ is encountered a left node is created and if a ‘1’ is encountered a right node is created. If either of the nodes are already created then the algorithm just traverses down the tree and checks the next bit. This is done repeatedly until each bit of the codeword is considered.

Once the decoder tree is built, the encoded.bin file content is used to traverse the tree from the root.

For each ‘0’ bit the algorithm traverses to the left child and for each ‘1’ bit the algorithm traverses to the right child. If the algorithm reaches a leaf node, the traversal starts again from the root.

### **Complexity**

The total number of traversals depend on the number of bits in the encoded.bin file as well as the sum of the number of bit in the codes of the codetable. Hence we can say that the complexity of traversing is dependent on those two factors.