# Advanced Data Structures (COP 5536) Spring 2017

Programming Project

Name: Avirup Chakraborty
UFID: 5291-4909

## Introduction

The purpose of this project is to choose and implement the suitable data structure for generating Huffman codes which is then used for implementing the encoder and decoder programs. This report describes the overall program flow with a special emphasis on the decoding algorithm and its complexity. It also includes a comparative study for evaluating performances of 3 data structures for generating Huffman tree.

## Program Structure and Function Prototypes

The program structure is divided into 4 sections:

**1. Frequency Table Generation:**

This step reads the input and generates a frequency table of all the distinct symbols present in the input file. A separate class **BuildFreqTable** has been used which consists of the function **private void generate_freq_table().** This function uses a linked hash map of definition **public LinkedHashMap<String, Integer> freq_map** to calculate and store the frequency table in the same order as in the input file. The function performs input type check and checks whether the input is an integer and within range 0 to 999999. It throws a **NumberFormatException** in case of these violations. It includes another function **LinkedHashMap<String, Integer> get_map()** which returns the linked hash map to the main function.

**2. Constructing the heap and Huffman Tree Creation:**

After the comparative study between 3 data structures viz. **Binary Heap, 4-way Heap and Pairwise Heap**, it was found that that **4-way Heap** was giving best performance. The **class min4wayheap** is used to build the heap and it consists of the following important functions:

a) **public void insert(TreeNode item)** : This function inserts elements to the 4 way heap by checking the existing values with the current inserted item and swaps the values with its parent to set the root node as the minimum element of the whole tree.

b) **public void buildHeap()** : This function builds the entire 4-way heap by calling another function **private void minHeapify(int i)** which takes each element and builds the 4-way tree by comparing with 4 nodes as left, mid1, mid2 and right nodes and inserts the Leaf node using the constructor **public TerminalTreeNode(String s, int val)** recursively.

c) **public TreeNode extractMin()** : This function extracts the node with a minimum value that is used to later to build internal nodes using the class **public InternalTreeNode(TreeNode item1,TreeNode item2)** in the Huffman Tree.

The Huffman tree is being generated by extracting 2 min values from the heap using the function **extractMin()** and then added together to create an internal node using the class constructor **public InternalTreeNode(TreeNode item1,TreeNode item2).** The TerminalTreeNode consists of the original symbols and are created by constructor **public TerminalTreeNode(String s, int val)**. Both the **InternalTreeNode and TerminalTreeNode** extends the abstract class of **public abstract class TreeNode.**

## 3. Encoding Algorithm:

After the Huffman Tree is built in the previous step using 4-way heap, the symbol code table file is generated by traversing through the entire Huffman Tree. The tree traversal is done using the function **void Traversal(TreeNode root,StringBuilder s)** which traverses from the root to left and right and appends 0 for the left child code and 1 for the right child code for a symbol. This function generates the contents of the **code_table.txt** in the following form:

```
2 000
999999 001
0 01
2245 10
446 110
34 111
```

The input vector is read and the symbols are looked up with the contents of the **code_table.txt file** and a string is generated named **encoded_string** which is a string of 0s and 1s. After writing multiple of 8 bits, the **encoded_string** is deleted till that point in order to avoid heap space issues. This string is then converted to binary values by performing bitwise operations and stored in the file **encoded.bin** as **01100100110010110101111101111101110010000.** Each 0 and 1 is 1 bit space, so the size of the file would be **(number of bits)/8** bytes.

**4. Decoding Algorithm:**

The decoder reads each bit of the encoded bit sequence and looks for the code values in the **code_table.txt file** and retrieves its corresponding symbols. The details of the algorithm are as follows:

- **Decoding Tree Generation:**

  a) The code table is read from the file **code_table.txt** and for each line the 2 values of code and symbol is stored in 2 separate variables.
  b) A binary tree is generated based on each bit of the code for each symbol by creating a left child for bit 0 and creating a right child for bit 1 till the code is read which marks the leaf of the tree. The symbol is fed into the leaf node of the binary tree. A separate class has been **public class freq_elem** has been used to generate the code table tree which has getters and setters for left and right child for the code table tree.

- **Decoding Tree Traversal:**

  a) After the decoding symbol tree is built, the encoded string is read byte wise from an array and a bitwise operation is done in order to extract the matching element.
  b) The decoded symbol table is traversed for the encoded bits till it finds a leaf node when it returns the original symbol that is stored in that path of the symbol tree.
  c) After each matched symbol is returned from the symbol tree, the encoded string is deleted till that point in order to avoid heap space issues.
  d) The final output file is stored in **decoded.txt** which is matched with the input file to match the results.

**Complexity of Decoder Algorithm:**

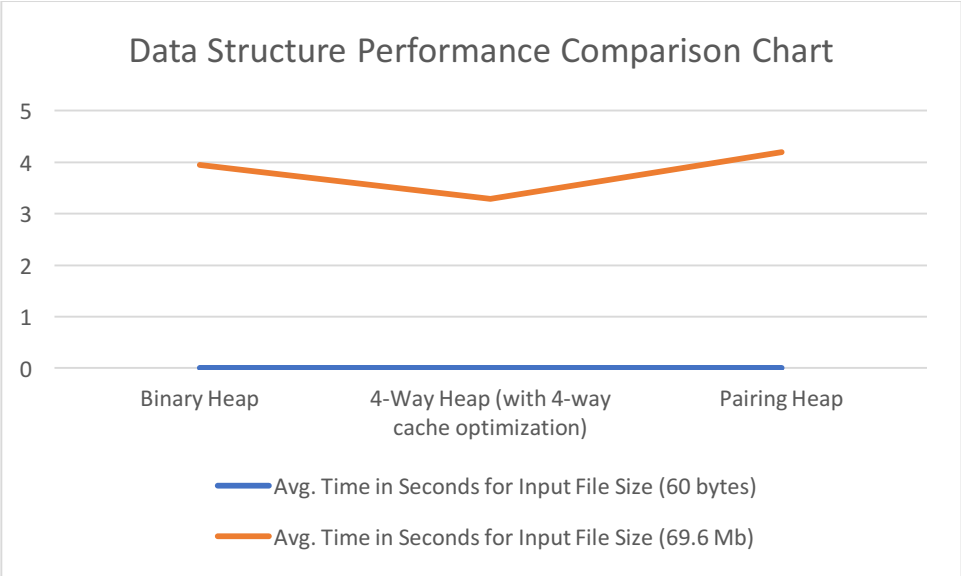
- **Decoder Tree Creation: O(n * length of largest code of a particular symbol)** where n is the number of distinct symbols in **code_table.txt**
- **Decoder Tree Search: O(n)** where n is the number of bits in **encoded.bin** file.

## Performance Analysis and Results

After executing the algorithm with the 3 types of heap, there were varied results for each type of heap and 4-Way Heap gave the best results. All the testing was done in **storm.cise.ufl.edu** server.

**Comparative Study of 3 data structures for Huffman tree generation:**

| Type of Data Structure | Avg. Time in Seconds for Input File Size (60 bytes) | Avg. Time in Seconds for Input File Size (69.6 Mb) |
|---|---|---|
| Binary Heap | 0.0012 | 3.946 |
| 4-Way Heap (with 4-way cache optimization) | 0.0008 | 3.288 |
| Pairing Heap | 0.0019 | 4.196 |




**4-way Cache Optimization Performance of 4-way heap:**

Initially, there was no 4-way cache optimization being done which led to higher tree generation time. But the 4-way cache optimization improved performance since the slots contained values for 4 elements at a time. It was a window of 4 slots sliding at the same time over the heap. Initially, 3 empty slots were filled in the frequency table vector in order to generate the 4 slots at the 1$^{st}$ iteration.

| Type of Data Structure | Time in Seconds for Input File Size (60 bytes) | Time in Seconds for Input File Size (69.6 Mb) |
|---|---|---|
| 4-Way Heap (without cache optimization) | 0.0011 | 3.831 |
| 4-Way Heap (with 4-way cache optimization) | 0.0008 | 3.288 |