# ADS Project Report

**Mayank Gupta**
**4899-8256**

## Task

The encoder file will create a frequency table from the input table and call Pairing Heap and generate a codetable which will be used to generate a .bin code and .txt codetable. This can be used by the decoder to generate a decoded file which will be identical to the original file.

**Structures used-**

1) **Huffman_Node(String d,int f, Huffman_Node l, Huffman_Node r)**
   This is used to store the data based on the frequency in the Huffman tree.
2) **HeapNode(Huffman_Node n,HeapNode c,HeapNode l,HeapNode r)**
   This HeapNode will store a Huffman Node and along with that keep track of the leftmost child which is c and the left and right sibling of the node.
   The leftmost child can be used to therefore visit other siblings by going right until we encounter a null.
3) **BinTreeNode(String data, left,right)**
   This is a Binary Tree that will store in the leaf nodes the value of the elements using the code table. '1' means right '0' means left. So for code of "01" means root.left.right.data would give us the value of that element.

# Function Prototype

## BinHeap/FourHeap

BinHeap and Fourway Heap have identical structure with the only difference being that four way heap will generate four children in the find_smallest() function.

1. **public Map<String,String> buildHuffTree(Map<String,Integer> freq_table)**

   Take the input freq table and generates the codetable from that.
   This is the only public function that is accessed by Huffman Performance.

2. **private void heap_array(int set_size, Map<String,Integer> freq_table)**

   It takes the input freq_table and set_size. This is used to generate the heap array.

3. **private void minHeapify(int p)**

   Minheapify takes the index of the parent and checks if any child is smaller than the parent, if yes then it replaces the child with the parent and travels with the parent till the process is terminated.

4. **private int find_smallest(int p)**

   It is used to find the smallest among p and it's children.

5. **private void huffman_insertion()**

Huffman_insertion will keep on removing the smallest elements, summing them up and reinserting into the heap until we are left with only the root node.

6. **private void insert(Huffman_Node SumNode,int freqsum)**
   It takes the input of the Huffman Node which we created by adding the 2 min elements and their combined frequency.
   It is performed such that an element is inserted at the bottom and is moved bottom up till we satisfy the Min Heap Properties.

7. **private void deleteMin()**

   deleteMin() will delete the first element of the heap array and replace it with the rightmost child i.e. the last element of the heap array. After this it will perform MinHeapify to satisfy Min Heap properties.

8. **private void char_encoding(Huffman_Node head,Huffman_Node left,Huffman_Node right,StringBuffer sb)**

   This is used to generate the values for huffman codes of an element. If we take the left path the code increase by a “0” and if we take the right path the code increases by “1”, this continues recursively until we enter all the values in the codetable.
   This takes the input a Huffman_Node and it’s left and right child, If the left and right child are null means that we have reached the leaf node.

## Pairing Heap

Unlike BinHeap or Fourway Heap, we cannot store the elements directly in form of an array so we make use of a HeapNode which stores the left and right sibling and acts similar to a linked list.

1. **private void insert_array(Map<String,Integer> freq_table)**

   In Insert array the elements are inserted to an initially root==null Heap

2. **private void insert(Huffman_Node temp)**

   In Insert we meld the input node temp with the root, if root==null the input node becomes the new root.

3. **private HeapNode meld(HeapNode first,HeapNode second)**

   Whoever's frequency is lesser becomes the parent after melding. If the frequency is same, the first is given priority and is the parent.

4. **private HeapNode firstchildtosecond(HeapNode child, HeapNode parent)**
   This will take two HeapNodes and turn the first node into the child of the second node.

5. **private void deleteMin()**

   Now, deleteMin() is again different from BinHeap/4 way heap, this is because in Pairing Heap whenever deleteMin() is performed, the children get added to the root list and then melding is performed from left to right. Due to this there was a need of a First in First Out Structure and the Dynamic ArrayList seemed to serve the purpose of maintaining a rootlist.

6. **private void huffman_insertion()**
   This is the same function as before.

7. **private void char_encoding(Huffman_Node head,Huffman_Node left,Huffman_Node right,StringBuffer sb)**

   This is the same function as before.

## Encoder

1. **public static void mapping(Map<String,String> codeTable)**
   Mapping will write the output codetable to a txt file.

2. **public static String encode(String s,Map<String,String> codeTable)**

   Here,encode will use the input file and codetable to replace data with the code and that will use createFile to generate a .bin file.

3. **public static void createFile(String s)**
   This will create a .bin file from the given string of bit values.

## Decoder

1. **public static Map<String,String> getCodeTable(String codetable)**

   This will generate the codetable by reading the file.

2. **public static String readEncode(String encoded)**

   This will read the encoded bin file and return it in form of a string.

3. **public static BinTreeNode huffmantree(String value, String code, BinTreeNode head)**
   This will generate the Binary Huffman tree and return it's root.

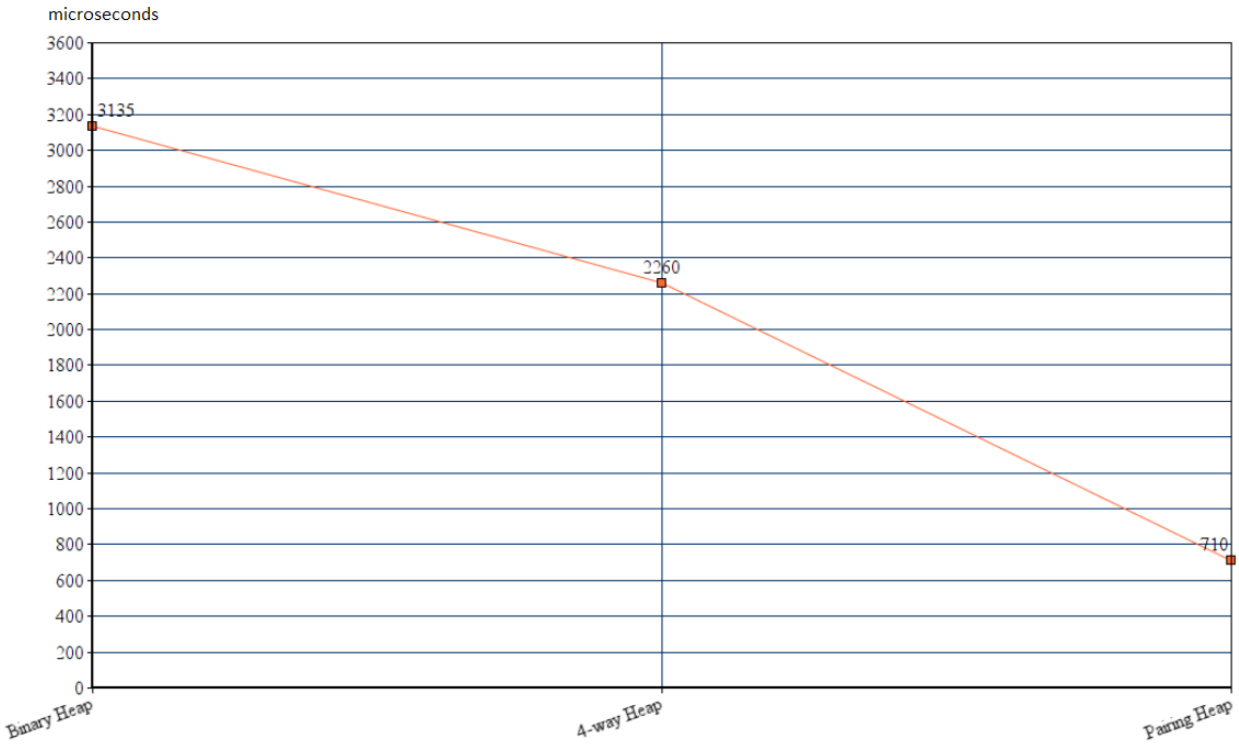
4. **private void output(BinTreeNode root, String bin)**

   This will take the above root and string in binary form and then write to a decoded.txt file by traversing the tree.

# Performance Analysis

This was the performance run for the 3 heaps on **sample_test** which was the smaller data set. This was conducted using the HuffmanPerformance class.

Time using Binary Heap (microseconds):3135
Time using 4-way Heap (microseconds):2260
**Time using Pairing Heap (microseconds):710**

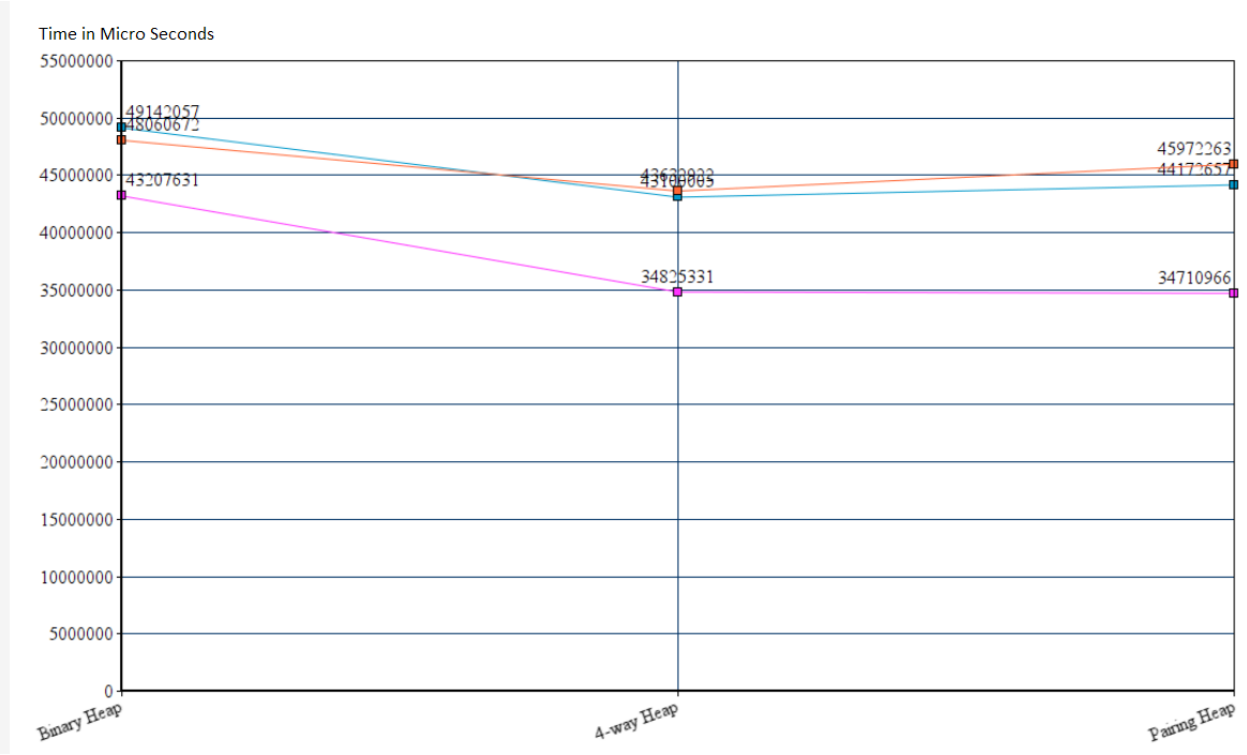



This was the performance run for the 3 heaps on **sample_input_large**.

Time using Binary Heap (microseconds):48060672
**Time using 4-way Heap (microseconds):43632922**
Time using Pairing Heap (microseconds):45972263

Time using Binary Heap (microseconds):43207631
Time using 4-way Heap (microseconds):34825331
**Time using Pairing Heap (microseconds):34710966**

Time using Binary Heap (microseconds):49142057
**Time using 4-way Heap (microseconds):43100005**
Time using Pairing Heap (microseconds):44172657




In this case, we can see that for sample input large, The 4-way Heap and Pairing Heap seem to be evenly matched which 4-way heap seems to retain a slight edge. But given that sample_input_large is on one side of the spectrum and is an extreme case, I also gave weightage to the input_test which was the smaller testbench. There the Pairing Heap seems to have performed more than 3 times better than a 4-way Heap. Thus for an average scenario , a Pairing Heap seems to give better performance than a 4-way heap and that is the one we use in the encoding decoding portion.

## Decoding Tree- Binary Tree with only leaf nodes having relevant data.

For Decoding, I used a standard binary tree. The only difference here was that the values were only stored at the leaf nodes of the tree. The rest were all

“InnerNode”.If the code said ‘0’ then it moves to the left child, if it says ‘1’ it will move to the right child. Here, one thing to note is that Huffman Trees have values only at the Outer Leaf Nodes. What this means is that if a node has left and right both equal to null, then it means we have reached a data node. This also means that once a code has been encountered, no other variable will have a code starting with this same string.