Part 1: Huffman Coding.

Files:
HuffmanCoding.java
BinHeap.java
FourHeap.java
PairingHeap.java

Structure of Program:
The HuffmanCoding class reads the input text file which contains the data to be encoded. To read the input file I have used the BufferedReader class. As I read the input I am building the frequency table which keeps a count of the number of times a value is occurring. To build the frequency table I have made use of HashMap<Key,Value> with key being the data item and value being its frequency.

Now to create the Huffman tree, I will pass the frequency table as an argument to three different methods each for binary heap, 4ary heap and pairing heap. These methods will shift the control to respective classes which have the necessary implementation.

These 3 methods are called 10 times to get the time taken for their execution. I have used System.currentTimeMillis() to get the current system time. This method is called before and after execution and the duration is calculated by taking the difference between the two times. This time is then divided by 10 to get the average time taken by each operation.

Function Prototypes:

1. HuffmanCoding.java

`public static void main(String[] args)`
• Read the input file passed as command line argument. • Creates frequency table • Measures timing of different heap implementations and prints to console.
`public static void build_using_binheap(Map<String,Integer> freq_table)`
• Receives the frequency table as input. • Creates object of BinHeap Class and builds Huffman tree by calling its build_using_binheap method. • Receives code table from the BinHeap Class method.
`public static void build_using_4heap(Map<String,Integer> freq_table)`
• Receives the frequency table as input. • Creates object of FourHeap Class and builds Huffman tree by calling its build_using_4heap method. • Receives code table from the FourHeap Class method.
`public static void build_using_pairingheap(Map<String,Integer> freq_table)`
• Receives the frequency table as input.

• Creates object of PairingHeap Class and builds Huffman tree by calling its build_using_pairingheap method. • Receives code table from the PairingHeap Class method.

2. BinaryHeap.java

`public class BinHeap`
`class Node`
• Node presents a structure which has the following elements in it. ➢ String data ➢ int freq ➢ Node left ➢ Node right
`public int parent(int i)`
• Returns the parent of the ith element in the heap
`public int KthChild(int i,int k)`
• Returns the Kth child of the ith element in the heap
`public Map<String,String> buildTree(Map<String,Integer> freq_table)`
• Takes frequency table as input and returns the code table
`public void insert(Node internal)`
• Inserts a Node in the heap
`public void minHeapify(int p)`
• This function corrects the violations in the heap which may be caused due to remove min or initializing operation.
`public Node removeMin()`
• Returns the Node with the minimum frequency
`public void gen_code_table(Node head,StringBuilder sb)`
• Takes the root node of the Huffman tree and generates a code table. • The StringBuilder is used to append the code as we traverse the tree and on reaching the leaf its value is added to the code table

3. FourHeap.java

`public class FourHeap`
`class Node`
• Node presents a structure which has the following elements in it. ➢ String data ➢ int freq ➢ Node left

➢ Node right
public int parent(**int** i)
• Returns the parent of the ith element in the heap
public int KthChild(**int** i,**int** k)
• Returns the Kth child of the ith element in the heap
public Map<String,String> buildTree(Map<String,Integer> freq_table)
• Takes frequency table as input and returns the code table
public void insert(Node internal)
• Inserts a Node in the heap
public void minHeapify(**int** p)
• This function corrects the violations in the heap which may be caused due to remove min or initializing operation.
public Node removeMin()
• Returns the Node with the minimum frequency
public void gen_code_table(Node head,StringBuilder sb)
• Takes the root node of the Huffman tree and generates a code table. • The StringBuilder is used to append the code as we traverse the tree and on reaching the leaf its value is added to the code table

4. PairingHeap

class Node
• Node presents a structure which has the following elements in it. ➢ String data ➢ **int** freq ➢ Node left ➢ Node right
class HeapNode
• HeapNode presents a structure which has the following elements in it/ ➢ Node element ➢ HeapNode child ➢ HeapNode leftSib ➢ HeapNode rightSib;
public class PairingHeap
public Map<String,String> buildTree(Map<String,Integer> freq_table)
• Takes frequency table as input and returns the code table
public void gen_code_table(Node head,StringBuilder sb)
• Takes the root node of the Huffman tree and generates a code table. • The StringBuilder is used to append the code as we traverse the tree and on reaching the leaf its value is added to the code table
public Node removeMin()
• Returns the Node with the minimum frequency

`public void insert(Node temp)`
• Inserts a Node in the heap.
`public HeapNode meld(HeapNode first,HeapNode second)`
• This method is called by the insert method and the combine method. • It receives the two nodes who must be melded together. • It compares their frequency values and makes the bigger frequency element the child of the smaller frequency.
`public HeapNode combine(HeapNode temp)`
• This function is called by the removeMin(). It takes the leftmost child of the removed element and goes about combining its siblings.
`public void print(HeapNode root)`
• Additional function written for debugging purposes. • It prints the Pairing Heap elements.

Performance Analysis

The Huffman tree is built and the code table is returned using the three heaps. The 3 methods to build Huffman trees with each heap are called 10 times to get the time taken for their execution. I have used System.currentTimeMillis() to get the current system time. This method is called before and after execution and the duration is calculated by taking the difference between the two times. This time is then divided by 10 to get the average time taken by each operation.

HEAP	Time Taken
Binary Heap	1452 milliseconds
4Way Heap	964 milliseconds
Pairing Heap	2681 milliseconds

These are the timings for the different heap implementations. The timings shown are for the case when the program is tested against **sample_input_large.txt.** We can see that 4way heap is the fastest. Hence, I will be using the 4-way heap for my project implementation.

```
45          long startTime = System.currentTimeMillis();
46          for(int i=0;i<10;i++)
47              build_using_binheap(freq_table);
48          long endTime = System.currentTimeMillis();
49
50          System.out.println("Binary Heap: "+(endTime - startTime)/10+" milliseconds");
51
52          startTime = System.currentTimeMillis();
53          for(int i=0;i<10;i++)
54              build_using_4heap(freq_table);
55          endTime = System.currentTimeMillis();
56
57          System.out.println("4way Heap: "+(endTime - startTime)/10+" milliseconds");
58
59          startTime = System.currentTimeMillis();
60          for(int i=0;i<10;i++)
61              build_using_pairingheap(freq_table);
62          endTime = System.currentTimeMillis();
63
64          System.out.println("Pairing Heap: "+(endTime - startTime)/10+" milliseconds");
65      }
66
67      public static void build_using_binheap(Map<String,Integer> freq_table)
68      {
```

Console

```
<terminated> HuffmanCoding [Java Application] C:\Program Files\Java\jre1.8.0_101\bin\javaw.exe (Apr 1, 2017, 5:49:08 PM)
Binary Heap: 1452 milliseconds
4way Heap: 964 milliseconds
Pairing Heap: 2681 milliseconds
```

Fig: Screenshot of the portion of the code which determines the timing of the 3 implementations.

For Binary heap and Four-Way heap, I have done the implementation using arrays, however for Pairing Heap I have used tree(reference) implementation. This means for my program, the nodes stored are stored in an arrays for binary heap and four-way implementation, and thus accessing these nodes and storing them takes O(1) time. For Pairing Heap however, I have to traverse through the tree for inserting or removing elements which takes more time. This is the reason why Pairing heap runs slowest for me.

Now, the four-way heap implementation is faster than binary heap because here we encounter lesser number of cache misses. Even though we increase the number of comparisons at each level in case of four-way heap, we are decreasing the number of levels. On top of that we optimize the four-way heap by shifting the starting position by 3 places, Because of this, all the children of a node can be accessed at once. Thus, we will have only one cache miss to access all children of a particular node, which greatly enhances the performance.

Thus, from the above explanation, we can understand the reason for which Four-Way heap gives the best performance followed by binary heap followed by Pairing Heap.

Part 2: Encoding

Files:
encoder.java
FourHeap.java

Structure of Program:

The encoder.java takes an input text file as a command line argument. It then generates a frequency table and then creates a Huffman tree using a 4ary heap. The build_using_4heap method returns the code table. The input file is then read again and for each data item encountered its corresponding code is retrieved from the code table. These codes are appended into a StringBuilder as the file is being read. To read the text file I am using BufferedReader class.

To write to a bin file I am using FileOutputStream and DataOutputStream. Now to generate the encoded.bin I am converting the StringBuilder to a byte array. For this I am using BitSet which contains the information of all the bits which are set to true i.e. set to 1 in our case. The BitSet can be converted to byte array using the toByteArray method of the BitSet. The byte array is now directly written to the bin file using the write method of the DataOutputStream. The code_table.txt file is written using BufferedWriter class.

Function Prototypes:

encoder.java

`public class encoder`
`public static void main(String[] args)`
• The main method takes input file as a command line argument • It generates two files: ➢ encoded.bin ➢ code_table.txt
`public static Map<String,String> build_using_4heap(Map<String,Integer> freq_table)`
• Receives the frequency table as input. • Creates object of FourHeap Class and builds Huffman tree by calling its build_using_4heap method. • Receives code table from the FourHeap Class method. • Returns the code table.

Part 3: Decoding

Files:
decoder.java

Structure of Program

The decoder.java takes two files encoded.bin and code_table.txt as command line arguments. The first step is to read the code_table.txt file and build the code table. Using the code table now the Huffman tree has to be generated. I have used FileInputStream and DataInputStream to read the encoded.bin file. The contents of the file are read into a byte array. This byte array is then converted to a String with the help of the BitSet implementation. This String which contains the code string is then traversed over the Huffman tree and the values encountered are written to a new file decoder.txt.

Function Prototypes:

`class HuffNode`
• `String data` • `HuffNode left` • `HuffNode right`
`public class decoder`
`public static void main(String[] args)`
• Takes encoded.bin and code_table.txt files as input. • Generates decoded.txt file
`public static HuffNode buildTree(String value, String code, HuffNode head, HuffNode prev,int left)`
• This function is called to build the Huffman tree.
`public static String getValue(HuffNode root, int i)`
• This function is used to return the corresponding value for the code encountered in the String containing the entire file code.
`public static void print(HuffNode head)`
• An additional function written for debugging purposes, it prints the Huffman tree.

Decoding Algorithm:

```
52              //build huffman tree
53              Set<String> set=codeTable.keySet();
54              for(String s:set)
55                  {
56                  root=buildTree(s,codeTable.get(s),root,null,-1);
57                  }
58
```

Using the code table I am creating the Huffman tree. The Huffman tree nodes have a data field, a left child and a right child. The root is initialized to null. Now for each entry in the code table I am calling the buildTree method. The buildTree method takes arguments data value and its corresponding code from the code table, the root node, a prev node which is initially null, and a flag left which indicates whether the head node is the left or right child of the prev node.

```
161  public static HuffNode buildTree(String value, String code, HuffNode head, HuffNode prev,int left)
162  {
163      if(code.length()==0)
164          {
165          head=new HuffNode(value);
166          if(left==0)
167              prev.left=head;
168          else if(left==1)
169              prev.right=head;
170          return head;
171          }
172      if(head==null)
173          {
174          head=new HuffNode("$");
175          if(left==0)
176              prev.left=head;
177          else if(left==1)
178              prev.right=head;
179          }
180
181      if(code.charAt(0)=='0')
182          buildTree(value,code.substring(1, code.length()),head.left,head,0);
183      else if(code.charAt(0)=='1')
184          buildTree(value,code.substring(1, code.length()),head.right,head,1);
185
186      return head;
187  }
```

buildTree algorithm:

1. If entire code is seen, its length will be 0. So here we will have to attach a leaf node. I have created a HuffNode whose data field will hold the value. Now I will check the left value. If it is 0, I know that I have come left from the prev node, thus I attach the newly created node to prev.left. Otherwise I will attach it to previous.right.
2. If the first condition is not true, I know that I am at an internal node. If I have not been through this node before, it will be null. Therefore, I will have to create a new internal node here. I create a new HuffNode whose data field will have "$" in it. Similar to step 1 I will check left value and attach the prev to head.
3. After seeing the internal node I have to check the code, it will tell me if I need to go left or right. I check the 0^{th} index and make a recursive call to the left or right child. I will send the substring with one less character each time a make a recursive call.

Complexity

Let N=number of entries in the code table
H=height of the Huffman tree.
M=length of the longest code present in the codetable

We make a call to the buildTree method for all the entries in the code table.
Therefore, we make N calls.

Now to build a leaf node for every entry of the code table we need to traverse from the root to the leaf node. Thus, the buildTree method takes O(H) time to execute.

However, the height of the tree will be the length of the longest code present in the code table i.e. M. Thus, the buildTree method takes O(M) time to execute.

Thus, the total running time of the algorithm will be **O(NM)**.