Advanced Data Structures
COP5536 Spring 2017
Programming Project Report
Name: Debarshi Mitra
UFID: 3381-3136

Introduction:

Huffman Coding is lossless data compression algorithm. There are primarily two major parts of the Huffman Code generation. They are building the Huffman tree from the input given characters and traverse the Huffman tree and assign codes to the characters. In this programming project, code is written to generate Huffman codes. At first three data structures, namely, binary heap, 4-way cache optimized heap and pairing heap are used to generate the Huffman trees and run time performances are compared. The data structure having the best run time is used to generate a code table. Thereafter, the code is passed through the encoder to encode the input and generate its code. The decoder creates the decode table and the decoded message. The decoded message is checked with the original input and both must be the exactly same.

Function Prototypes and structure of program:

1. **Encoder:** The encoder takes the input file as the input and generates a frequency table. From the frequency table, the Huffman tree is built and code table is generated. From code table, the two output files of encoder are generated which are encoded.bin and code_table.txt.

1.1. Frequency Table:

```
public class Frequency_Table {
        private int entry;  // each entry in the input text file
        private int frequency; // frequency of distinct entries in the input text file
        private Frequency_Table left_child; // left child of the current object in the Huffman tree
        private Frequency_Table right_child; // right child of the current object in the Huffman tree
    }

public Frequency_Table(int entry, int frequency){ //this constructs Frequency_Table object
                this.entry=entry;
                this.frequency=frequency;
                this.left_child=null;
                this.right_child=null;

public Frequency_Table get_left_child() // get value of the left child of the current node

public void set_left_child(Frequency_Table left_child) // set value of the left child of the current node

public Frequency_Table get_right_child() // get value of the right child of the current node

public void set_right_child(Frequency_Table right_child) // set value of the right child of the current node

public int get_entry() // get value of the current node

public void set_entry(int entry) // set value of the current node

public int get_frequency() // get value of the frequency of an entry

public void set_frequency(int frequency) // set value of the frequency of an entry
```

1.2. Huffman Tree using Heap:

```
public class Four_Way_Heap { // this class is used to generate the Huffman tree using 4-way heap
        private ArrayList<Frequency_Table> v_list; //Arraylist is used
```

```
public Four_Way_Heap(ArrayList<Frequency_Table> items) {
    this.v_list = items;
    build_heap();
```

public void insert // Inserts the element into the heap

public void build_heap // develops the 4-way heap

public Frequency_Table extractMin() // returns the minimum entry in the current table

private void min_heapify(int i) // does the 4-way comparison of the heap and inserts the minimum entry.

private int p(int i) // returns the parent

private int l(int i) // returns the left child

private int r(int i) // returns the right child

private void swap(int i,int p) // swaps the value of the parent with the children and puts the smallest value as parent.

1.3. Encode Data from Code Table:

public class encoder –

The file is taken from the command line as an input. When a new number is found, an object is created as entry, when an already retrieved number is found, the frequency is incremented. After that **ArrayList<Frequency_Table> freq_table_vector1** is used to build the Huffman tree by running the **extractMin()** method twice. After that, the **build_code_table** method is called which does inorder traversal and labels each node. If it's a left child, label appends 0 and if right child, label appends 1. The corresponding labels are pushed in a HashMap. The HashMap is read and the code_table.txt file is generated. The initial input file and the **code_table.txt** file are combined to generate the **encoded.bin** binary file.

2. Decoding Algorithm and its Complexity:

public class decoder –

The decoder will take encoded.bin binary file and code_table.txt as input. The decoder builds a decode tree and generates decoded.txt. The decoded.txt and the initial uncompressed input file would be identical.

2.1 Build Decode Tree:

In the code_table.txt, the binary strings are read and each time a 0 occurs it is pushed towards the left of the current node and 1's are pushed towards the right of the current node as internal nodes till the end of the string is reached.

2.2 Create decoded.txt:

The encoded.bin is parsed byte by byte. In the decode tree created above, the tree is traversed from top. When there is a 0, the traversal is done towards left and when there is a 1, the traversal is done towards right. When there is a leaf node, the entry field of the node is written to the decoded.txt file. The current node is updated to the root node.

2.3 Decode tree generation algorithm:

1. Read the code corresponding to the different input values from the code_table.txt file.
2. From every line in code_table.txt, create two variables to store the input value and the encoded code.
3. For all the code values, read each character of the string.
4. Create a root node in the binary tree and after every operation, return the last visited or created node. For every '0', create or visit the left child, and for every '1', create right child for the last visited or appended node.

5. For every character in the code string, insert the node as a child of the last visited or created node. If the node is already present as a child node, then visit it and return that node. Otherwise create the node as child node and return the newly created node.
6. If the current character is the last character of the code string for input value, then store that input value in the node corresponding to the last character.
7. For rest of the code value strings in the code_table.txt file, start from the root node and repeat step 5 and 6.

Time Complexity – O(n*length of largest code corresponding to input value in the code_table.txt file); where n is the number of distinct input symbols present in the code_table.txt file

2.4 Decoded data generation algorithm

1. Read each byte from the encoded.bin file and store it in a byte array.
2. Append all the bits of a byte in a stringbuilder object by performing bitwise operation.
3. Now, start from the root and traverse the binary decode tree by reading each character of the stringbuilder.
4. If a leaf node is encountered, get the input value corresponding to the node and print the value in the decoded.txt file in a new line, delete the substring till index of the stringbuilder and start from the root node again.
5. If the leaf node is not encountered, then read the next byte, and append all the bits as character in the stringbuilder object, start traversing the binary tree from root.

Time Complexity:

O(n) where n is the number of bits in the encoded.bin file.

Performance Analysis – Results and Explanation:

Comparative study of the performance of the different data structures:

Data Structure	**Average time in millisecond for sample_input_large.txt file**
Binary Heap	**4155.2**
4-way cache optimized heap	**3786.5**
Pairing Heap	**23307.7**

From the table above, it can be confirmed that the 4-way heap performs better than the pairing heap and the binary heap. The node in a 4-way heap can have up to 4 children and inserting a new element will need less traversals than the other 2 heaps present. The 4-way heap algorithm use cache optimization technique. All the children are accessed in cache optimized manner and it helps improving the performance of the algorithm.