Advanced Data Structures Project Report Spring 2017

Submitted By,
Sarah Grace Samji
3937-2723

Introduction

The goal of this project is to generate Huffman Code for given data using a priority queue. Three different priority queue implementations, namely- Binary Heap, 4-Way cache Optimized Heap and Pairing Heap have to be implemented to build the Huffman Tree and their performances have to be compared, ultimately choosing the heap with best run time to build the Encoder.

This report serves to address the structure of the program and analysis of the data obtained through execution of the program with various priority queue implementations. The project has been written in Java. A makefile has been provided with the source code. The compilation will generate a binary called **‘encoder and ‘decoder’** which can be run.

Program Structure

The program includes two main classes: encoder.java and decoder.java. Given below is the detailed structure for each. The functions are called in the order presented.

encoder.java

Node class

```
private static class Node implements Comparable<Node>
      {
        private final int key;
        private final int freq;
        private final Node left, right;

        Node(int key, int freq, Node left, Node right) {
            this.key    = key;
            this.freq  = freq;
            this.left  = left;
            this.right = right;

        }

}
```

Defines the node structure representing the Min Heap and Huffman Tree.

Attributes are:

Key: type int to store the Key

Freq: type int to store the frequency of occurrence of the key

Left,Right: Type Node attribute to store the left and right child.

public static void main(String args[])

The main function of encoder class that accepts the input file as command line arguments. It makes calls to the respective functions of the encoder.

public static void frequency_table(String args)

Reads the input file using BufferedReader and FileReader and records the frequency of occurrence of each integer in the freq_table[] array.

public static void priority_queue(int[] freq_table) throws IOException

For each non zero element in the freq_table create a new node and call heapify(). Thus providing the nodes of the min-heap tree for the priority queue.

public static void heapifyup(Node n)

This function is used to min-heapify the current priority queue. A whilie loop runs as long as the cur_node is greater than zero. Here we check the currently inserted node and it's parent. If the current node is greater than it's parent then swap the nodes, else the tree is a min-heap already.

public static void remove(int ind)

This function swaps the min node with the last node in the queue and deletes the last node. It then calls heapifydown() on the root.

public static void heapifydown(Node n)

This is called upon deletion of a node from the queue. The node at the root is continuously checked with it's children an swapped with the child with lesser frequency until a leaf node is reached.

public static void huffman_tree()

If there are more than two nodes in the queue then a Huffman tree can be constructed. This is done by merging the two min nodes and inserting as a new node (with combined frequency of extracted nodes) into the queue.

public static void code_generation(Node n, String code, BufferedWriter fw) throws IOException

This function traverses the huffman tree recursively and writes the generate codeword and key pairs into a file using BufferedWriter into code_table.txt. It also stores the codes as string into a code_table[] array for future access.

public static void encoder_file(String args)

Takes as input the input file and references the code_table[] array to generate the encoded bits of the data. BufferedOutput Stream is used to write binary bit stream into encoded.bin. Binary data is produced by initializing an integer variable bytearray and then inserting the bit's into each bit position.

decoder.java

```
public static class Node
      {
        public int key;
        public final String code;
        public Node left;
            public Node right;

        Node(int key, String code, Node left, Node right) {
            this.key    = key;
            this.code  = code;
            this.left  = left;
            this.right = right;

        }
```

Node structure for the Decoder tree.

public static void main(String args[])

Accepts two filenames as command line arguments. First file is the code_table.txt BufferedReader is used to read the file and the key and code are passed as parameters to insert().

public static void insert(int key, String code)

For each codeword , iterate on each bit and traverse left or right depending on the value to generate the decode tree.

private static void decode(String args)throws IOException

Create decoded.txt by reading encoded.bin using InputStream. Each byte of data read is checked bit by bit and used to traverse the decoder tree. Upon reaching a leaf node the key is written out to the file.

Performance Analysis

Time calculated in milliseconds.

	Binary Heap	**4-Way Cache Optimized**	**Pairing Heap**
62 Bytes	5	3	2
66.4 MB	893	920	1013
664MB	1524	1589	1653

Based on the average run time to construct Huffman tree **Binary heap** showed the best results. However, Binary and pairing were comparable in their run times for different input sizes. Since the implementation of Huffman tree involves more of delete min operations binary heap tends to perform better, however cache optimizations in 4 way heap brought the run time up considerably.

Binary heap

Decoding Algorithm

The decoder takes as input the encoded.bin and code_table.txt file.

Here I have created a Binary Trie to store the decoder tree. Here values are not associated with the node and a key value is determined by its position in the tree. Since every key is associated with a code and the code is binary a Binary Trie is best suited for this application considering it has fewer collisions than Hashmap and also more of simple look-up operations.

Consider a string of length m then the complexity of a **search** operation is **O(m).** This is because every search operation traverses the trie from the root to the desired node which will be a leaf if the search is successful. Hence even in the worst case the maximum time is limited to the length of the string.

Similarly **insert** operations has a complexity of **O(m*n).** A insert can be of a new node or it can already exist. But the process is similar to search. Insertion of first node or node whose code word isn't prefixed by another code you start at the root, create new nodes corresponding to the code till the code word is completed and then the last node's key value is updated with the key. If some other node prefixes a substring of the key's code then you just have to traverse down the tree and create a few additional nodes. In the worst case all n nodes will have to be created fresh hence O(m*n).

ReadMe

IDE:Eclipse (JDK 1.7.0_79)

Two class files: encoder.java decoder.java

Files read: sample_input_small.txt, sample_input_large.txt, encoded.bin, code_table.txt

Files written: encoded.bin, code_table.txt, decoded.txt

Conclusion

From the above experiment's we can see that binary heap and 4-way cache have comparable complexities. However, as size of file goes beyond cache memory the 4-way heap may perform better but for the data file we are limited to binary heap is best suited as binary heap performs better due to large number of remove min operations(4-way has a poor remove min complexity).