Huffman Code Text File Compressor

Contents

Purpose

The purpose of this report is to identify and defend key structural decisions made in the code, and to explain the code in the source files.

Note to consider

In order to fully understand the working of each line in this code, read the comments given in the source files.

Running the Program

1. **Input:**

   You will be asked to input a file name. The name must be valid, and the file must be present in the same folder as the source files.

   a. If an incorrect name is entered, an error message will prompt you to enter the file name again.
   b. If the file name is valid, a confirmation message will flash on screen.
   c. You must press 'c' to continue to the next part.

2. **Choice:**

   You will be shown a set of 3 options to choose from. Simply type in one of the numbers representing the options:

   1. Basic Huffman Code
   2. Optimized Huffman Code
   3. Exit

   a. If an incorrect number is entered, an error message will prompt you to enter the option number again.

3. **Output:**

   a. If option 1 was selected, the basic Huffman code will be shown.
   b. If option 2 was selected, the optimized Huffman code will be shown.
   c. If the above 2 options were selected, the *Average Bit Rate, Compression Ratio,* and *Original File Length* will be displayed.

d. If option 3 was selected, you will exit the program (An exit message will be displayed).

4. Repetition:

Steps **2** to **3** will be repeated until option 3 is selected to exit the program.

Structures Used

1. Linked List: (For storing the generated Huffman code)

A linked list was used to store the Huffman code being generated as the tree was traversed. Although a dynamic array could have been used instead, I opted against it. The reason being it is that when generating the Huffman code, the length of the code for each character is unknown. If an array were used, the size would have to be doubled whenever the max limit would be reached for the array. This would mean transferring the contents of the array to a new array of double the size. This not only wastes memory space, but the time complexity also greatly suffers.

With a linked list (head and tail pointers are used along with a length variable), elements can be inserted and deleted as the tree is traversed without having to go through the whole list. Simply using the head and tail pointers, we can insert and remove from either side which makes this much more efficient.

2. Tree:

Initial Plan:

At first, I decide to use a BST to take input from the file since it is easily able to handle duplicates which would be the basis of the character's frequency calculation. Then this BST would be converted into a basic Huffman tree and, finally, the basic tree would be optimized.

However, the BST to Huffman tree conversion seemed unnecessary and would just add more time to the program execution since the conversion from BST to Huffman would be an extra function.

Revised Plan:

It was decided to take input from the file directly into a basic Huffman tree. An algorithm was designed so that the frequency could also be calculated, and duplicate characters were not added. This was possible since a basic Huffman tree has no order, therefore, the nodes could be placed anywhere in the tree (in no specific order).

The structure of the tree which was implemented is how it is defined in the project question statement. The functions used to form this structure will be explained further down this report.

3. Heap: (for optimizing the tree)

A Min Heap was used to act as a priority queue. With its functionality, it was highly suitable for optimizing the tree.

Heap was made of an array. In a binary heap, the structure is that of an almost complete binary tree. This is why an array implementation was suitable. An array element can be accessed using an index which makes it easy to traverse. Since the tree is almost complete, there are no gaps between nodes. Therefore, there are no gaps of wasted space in the array either.

Using this Min Heap structure, the tree was optimized through heap sort. A heap can maintain order and fullness of a tree. Using these properties, arranging the nodes in order proved to be easy to implement.

Task 1

In order to clearly highlight the difference between a basic and an optimal Huffman code, it decided to implement the basic tree as well.

In this task, the file is read line by line and characters are extracted from each line and inserted into the Huffman structured tree. Following this, the tree is traversed, and a table is displayed:

Table output diagram:

Character	Frequency	Huffman Code
Char example 1	2	0001
Char example 2	3	00001

[Note: The table for task 2 will be in the exact same format as the one above.]

Task 2:

The output of this task is in the same format as task 1. The optimized tree uses the task 1 tree. However, a min heap was used to optimize the tree. In task 2, the nodes of the tree from task 1 are copied (deep copy) into the heap. This is done so that we can still use the task 1 tree to display the basic Huffman tree to the user.

After this, we optimize the tree using heap sort. The Huffman code generated in task 2 is much smaller than the one generated in task 1. The character with the most frequency has the least lengthy code.

Structures and their functions

[Note: Setters, Getters, Constructors and Destructors will not be discussed as they are self-explanatory]

1. Linked List

void insert(char);

This function simply inserts a character (0 or 1) into the list.

void delAtEnd();

This function deletes the node at the end of the list (last input).

void display(ListNode*);

This function displays the whole list in a single line to represent the generated Huffman code.

2. Tree

TreeNode* createNode(char, int);

This function creates a new node and assigns the character and frequency we pass as arguments. The node's children will be NULL It is a sole node with no connection.

bool find(TreeNode*, char);

This function searches through the whole tree to find a node which matches the character passed into it. If found, then the function returns true. Otherwise it returns false.

void insertFromFile(char);

This function is used to insert characters which are passed to it. The main use here is to use a structure which can reject duplicate nodes and count the frequency of each character as it is inserted

void Inorder(TreeNode*);

This function is designed **only for testing**. It prints data of the tree nodes in an inorder recursive pattern.

void PrintCode(TreeNode*, LinkedList*, double*);

This function prints the character along with its frequency and Huffman code as it traverses the tree in a preorder recursive pattern. The output will be the table form discussed earlier in the report. This function also multiplies the frequency of a character with the length of its Huffman code and adds the result into the ABR (Average bit rate) variable.

3. Min Heap

void insert(TreeNode*);

This function creates a deep copy of the node passed to it. This copy is then Inserted into the heap at the end of it in order to maintain the almost complete binary tree property. The function then proceeds to swap nodes' positions after comparing their frequencies. This is done to maintain the heap order.

void heapify(int);

The root is passed initially. This function swaps the parent node with the smallest child if the child's frequency is smaller than the parent's frequency. This function is used in the heapsort() function to maintain heap order when a node is removed.

void heapSort();

This function keeps removing nodes from the heap until no elements remain. The remaining array is fully sorted in descending order.

void formTree(TreeNode*);

This function inserts nodes at the end of the heap after sorting is complete. It is used to insert parent nodes when forming the optimized Huffman tree.

void display();

This function is **only for testing** functionality and errors in the heap.

Functions Outside of Structures

void storeChar(Tree*, string, int*);

This function reads the passed string character by character and inserts each character into the tree. The file length is also calculated here.

void readFile(Tree*, string, int*);

This function reads the file line by line until the file ends. It calls storeChar() function to store the extracted string into the tree. It also accounts for '\n' character to be inserted.

void heapStore(TreeNode*, MinHeap*);

This function traverses the tree in a preorder recursive pattern and stores the node into the heap by calling heap->insert() function of the min heap structure discussed previously.

void Optimal_Huffman(MinHeap*,Tree*, Tree*);

This function calls a combination of other functions in order to optimize the tree of task 1.

void displayTitle();

This function prints a title for the user to see.

void continue0();

This function holds the continue functionality. This means that user has to enter 'c' to continue to the next screen. It is mainly used before a screen clearing operation such as system('CLS").