Assignment 6
Huffman Coding

By Jiancheng (Jason) Xiong
CSE 13S Spring 2023

Due June 8th 2023, 11:59 PM

1 Purpose

The purpose of this assignment is to create a program that takes an input data file and compresses it using the Huffman algorithm. We will use a program huff that computes the huffman code of an input file and encodes an input data file. We will also have the capability to decode the file using a huffman decoder, which has been provided.

2 Using the Program

In order to use the program, compile and run the file huff.c. In order to run the program, run ./huff and specify the commands in which you want to run. The available options are;

-i (inputfile), specifying the input image on which the program will run on.

-o (outputfile), specifying the output file in which the image will be stored

-h; displays a help message detailing program usage.

3 Program Design

The main program will be in the file huff.c, while each of the separate data structures will be in;

- io.a for the reading and writing of various uint sizes, includes functions that open and close files unbuffered, along with header file io.h
- node.c for the creation and manipulation of the node ADT; includes functions that create, delete, and display nodes.
- pq.c for the creation and manipulation of the priority queue ADT; includes functions that create, delete, and modify entries of a priority queue.
- Makefile for the compilation and joining of the separate files

4 Data Structures

There are various implemented data structures, being;

The bitwriter struct is used to facilitate the writing of files. It stores a buffer it uses for storing text to be written, a uint8_t byte, as well as a uint8_t that saves the bit position.

The Node struct is used to save the properties of a node, such as the symbol of the node, the weight of the node, as well as the current code that represents the node, as well as the current length of the code. It also stores a pointer to its left and right children nodes.

The ListElement struct used to create a linked list ADT, storing a pointer to its node as well as a pointer to the next node in the list.

The PriorityQueue ADT that represents the actual queue. We use a second struct to represent the queue so we can always have a pointer to the queue.

The Code ADT that saves the code in the huffman tree that represents the current node, as well as an uint8_t that represents the length of the code.

5 Algorithms/Pseudocode

For Bitwriter Functions:

Bit_write_open(filename)
    Calloc some memory for the bitwrite structure
    Create a buffer using write open
    Set the underlying stream in the bitwrite structure to the buffer made using write open
    Return a pointer to the new bitwrite structure

Bit_write_close(filename)
    If there is data left in the buffer: flush bytes using write_uint8 until it is empty
    write_close() the underlying stream buffer
    Free the bitwriter memory

Set its pointer to null

Bit_write_bit(buffer, x)
Set the bit_position pointer to 0
Leftshift bit by byte if necessary
Increment bit position

Bit_write_uint8(buffer, x)
Call bit_write_bit 8 times, starting from the right bit

Bit_write_uint16(buffer, x)
Call bit_write_bit 16 times, starting from the right bit

Bit_write_uint32(buffer, x)
Call bit_write_bit 32 times, starting from the right bit

For Node Functions:

Node_create(symbol, weight)
Calloc a node
Set node->symbol to symbol
Set node->weight to weight
Set node->left and node->right to NULL
Return a pointer to the node

Node_free(node)
Free the node
Set the pointer to the node to NULL

For Priority Queue Functions:

pq_create(void)
Calloc some memory for the priority queue structure
Return a pointer to the priority queue object

pq_free(priority_queue)

free(priority_queue)
Set the pointer to the priority_queue to NULL

pq_is_empty(priority_queue)
Check if the queue's list field is NULL, if so, return true otherwise return false

pq_size_is_1(priority_queue)
Check the first element in the linked list field
If the first element-> is not NULL (ie there is another element in the linked list), return false, otherwise return true

enqueue(priority_queue, tree)
Allocate memory for a new Listelement e, and set e->tree to tree
Check the current queue's state;
If the queue is empty, set q->list to e
If the weight of the new tree is less than the weight of the head, add it to the beginning of the queue by setting e->next to q->list and setting q->list to e
If the weight of the new tree is greater than the weight of the head, start at the head of the tree, and traverse the linked list until you find an element with a weight greater than the weight of the new tree, then add it to the priority queue before that element

dequeue(priority_queue, tree)
Save the first element from priority_queue into tree, update the pointers, and free the first node

pq_less_than(node1, node2)
If node1->weight is less than node2->weight return true
If node1->weight is greater than node2->weight return false
Otherwise return if node1->symbol < node2->symbol

For Huffman Encoding

fill_histogram(inbuf, histogram)
Initialize filesize to 0

Iterate through the inbuf and add the read bytes to the histogram, incrementing filesize while doing so
Add 0x00 and 0xff to histogram as a hack
Return filesize

Create_tree(histogram, num_leaves)
Create a priority queue pq
Initialize num_leaves to 0
Create a node for every symbol that appears (has a frequency > 0), setting its symbol and adding it to the priority queue

While the size of the pq is greater than 1:
Dequeue a node into left
Dequeue a node into right
Make a parent node with the weights of both children added together
Set parent node->left to left
Set parent node->right to right

Enqueue the parent node

Return the remaining node

Fill_code_table(code_table, node, code, code_length)
For each child of a node (if it exists)
Recursively call this function on that child with a code_length + 1
Else:
Set the code and code_length of node in the code_table

huff_compress_file(outbuf, inbuf, filesize, num_leaves, code_tree, code_table)
Write uint8_t ‘H’
Write uint8_t ‘C’
Write uint32_t filesize
Write uint16_t num_leaves
Huff_write_tree(outbuf, code_tree)
For every byte b from inbuff
Look up the code and code_length
For i from 0 to code_length - 1

Write the rightmost bit
Code >>= 1

6 Function Descriptions

Bit_write_open

Takes in const char *filename
Returns a BitWriter struct
Opens a file, creates and returns a pointer to a BitWriters structure

Bit_write_close

Takes in a Bitwriter **pbuf
Returns nothing
Flushes data in the byte buffer, frees the buffer, and sets pointer to NULL

Bit_write_bit

Takes in a Bitwriter *buf and uint8_t x
Returns nothing
Write a single bit using values in the BitWriter struct pointed to by buf, if there are 8 bits in the buffer

Bit_write_uint8, 16, and 32

takes in a Bitwriter *buf and uint8/16/32_t x
Returns nothing
Writes 8/16/32 bits using bit_write_bit

Node_create

Takes in a uint8_t symbol and double weight
Returns a Node
Creates a node and returns a pointer to it

Node_free

Takes in a Node **node
Returns nothing
Frees the *node and sets it to NULL

Pq_create

Takes in nothing
Returns a PriorityQueue
Allocates a PriorityQueue object and returns a pointer to it

Pq_free
Takes in a PriorityQueue **q
Returns nothing
Frees q and sets pointer to null

Pq_is_empty
Takes in a PriorityQueue *q
Returns a bool
Checks if pq is empty

Pq_size_is _1
Takes in a PriorityQueue *q
Returns a bool
Checks if pq has only one element

Enqueue
Takes in a PriorityQueue *q and a Node *tree
Returns nothing
Adds the tree to the priority queue in the correct position

Dequeue
Takes in a PriorityQueue **q and a Node **tree
Returns bool
Removes the first element, rearranges the pointers, and returns true

Pq_less_than
Takes in two nodes Node *n1 and Node *n2
Returns bool
Returns if n1 is < n2, if they're the same weight the ascii of their symbol is used as a tiebreaker

Fill_histogram
Takes in a Buffer *inbuf and a double *histogram
Returns a uint64_t

Create_tree

Takes in a double *histogram and a uint16_t *num_leaves

Returns a Code

Creates and returns a pointer to a new huffman tree

Fill_code_table

Takes in Code *code_table, Node *node, uint64_t code, uint8_t code_length

Returns nothing

Fills a code table for the huffman code

Huff_compress_file

Takes in BitWriter *outbuf, Buffer *inbuf, uint32_t filesize, uint16_t num_leaves, Node *code_tree, Code *code_table

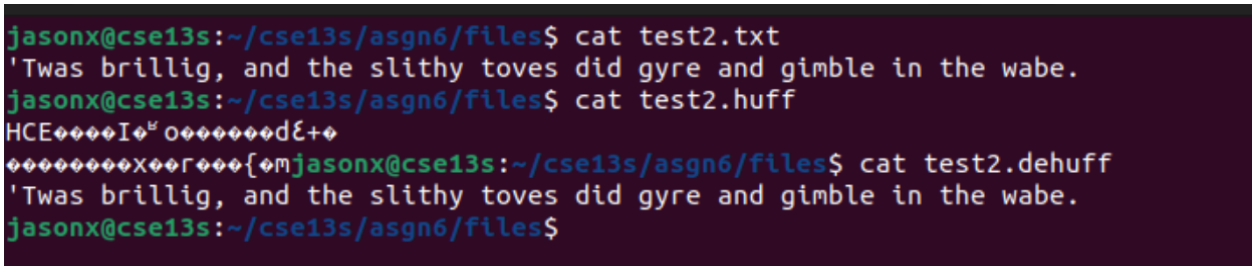
Returns nothing

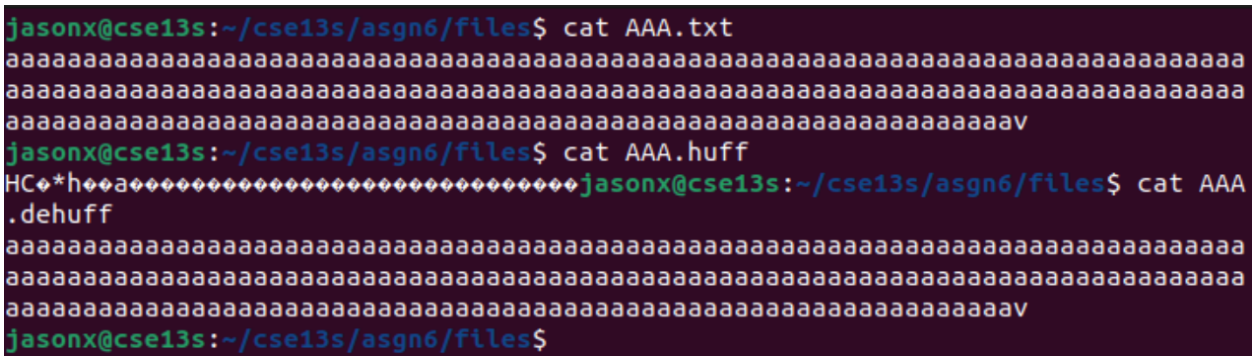
Compresses a file using huffman encoding

7 Results

While programming this assignment, I learned about the ways in which we can use compression in order to decrease the sizes of files we send to others. I have learned a variety of different ways in which files can be compressed, such as run-length encoding, presence bits encoding, and other encoding methods. This assignment taught me the specifics of Huffman encoding, such as the creation of the code table, how codes are formed, as well as other important aspects of the Huffman encoding process.

Pictures of the program running:





8 References

"Getopt() Function in C to Parse Command Line Arguments." *GeeksforGeeks*, 10 Sept. 2018, www.geeksforgeeks.org/getopt-function-in-c-to-parse-command-line-arguments/.

Open(2) - Linux Manual Page, 27 Aug. 2021, man7.org/linux/man-pages/man2/open.2.html.

"BMP File Format." Wikipedia, 16 May 2023, en.wikipedia.org/wiki/BMP_file_format.

"Huffman Code." *Brilliant Math & Science Wiki*, brilliant.org/wiki/huffman-encoding/. Accessed 6 June 2023.