Assignment 1 – Pass the Pigs

Your Name

CSE 13S – Spring 2023

Purpose

The purpose of the huff program is to compress data files using Huffman Coding. Huffman Coding is a technique that reduces the size of the files by representing frequently occurring bytes with fewer bits and less common bytes with more bits. By implementing this program, we can significantly reduce the storage space required for data files, leading to more efficient data management. The huff program provides a reliable and straightforward solution for compressing files, contributing to improved storage efficiency and optimized file transfer.

How to Use the Program

To use the program, you can follow these steps:

1. First, ensure that you have the necessary test files in the ”files” directory. We have provided several test files for you to use.

2. To compress a file, run the `"huff"` program using the command-line options. For example, you can compress the `"zero.txt"` file and save the compressed output as `"zero.huff"` by running the following command:

```
$ ./huff -i files/zero.txt -o files/zero.huff
```

3. Similarly, you can compress other files by replacing ”zero.txt” and ”zero.huff” with the appropriate file names.

4. If you want to decompress a compressed file back to its original form, you can use the `"dehuff-arm"` or `"dehuff-x86"` program. For example, to decompress the ”zero.huff” file and save the decompressed output as `"zero.dehuff"`, run the following command:

```
$ ./dehuff-arm -i files/zero.huff -o files/zero.dehuff
```

5. Repeat the above step for other compressed files, replacing the file names accordingly.

6. To verify the accuracy of the compression and decompression process, you can compare the original text file with the decompressed file using the ”diff” command. For example, to compare the ”zero.txt” file with the `"zero.dehuff"` file, run the following command:

```
$ diff files/zero.txt files/zero.dehuff
```

If there are any differences between the two files, the ”diff” command will report them.

7. You can perform the above comparisons for all the files by repeating the ”diff” command with the appropriate file names.

8. Alternatively, to automate the compression, decompression, and comparison process, you can use the provided shell script called `"runtests.sh"`. This script looks for files with the `".txt"` extension in the ”files” directory and performs the compression, decompression, and comparison automatically. To run the script, use the following command:

```
$ ./runtests.sh
```

Note that you can modify the `"runtests.sh"` script to see and customize its functionality.

9. Lastly, if you need help or want to learn about the available command-line options, you can use the "-h" option. It will display a help message on the terminal. For example:

```
$ ./huff -h
```

The help message will provide information on how to use the program and the available command-line options.

Program Design

To maintain the program effectively, it is essential to understand its organization, including the main data structures and algorithms used. Here is an overview of the main data structures and algorithms in the program:

0.1 Bit Writer:

- Reuses Unbuffered File I/O functions for reading and writing uint8_t, uint16_t, and uint32_t data types.
- Implements additional functions for writing files bit-by-bit using write_open(), write_uint8(), and write_close().

0.2 Node:

- Represents a node in a binary tree used for Huffman Coding.
- Contains fields such as ‘symbol‘, ‘weight‘, ‘code‘, ‘code_length‘, ‘left‘, and ‘right‘.
- Provides functions to create and free nodes and print trees.
- Nodes’ fields can be accessed directly using the C element-selection-through-pointer (-¿) operator.

0.3 Priority Queue:

- Implements a Priority Queue abstract data type.
- Stores pointers to trees, where each tree node has a weight.
- Uses a linked list to represent the Priority Queue.
- Consists of two structs: ‘ListElement‘ and ‘PriorityQueue‘.
- **‘ListElement‘** represents a node in the linked list, containing a pointer to a tree and a next field.
- **‘PriorityQueue‘** represents the queue itself and points to the separate linked list with a list field.

0.4 Huffman Coding:

- Involves five steps for encoding files using the Huffman algorithm.
- Step 1: Create a histogram of the input file’s bytes/symbols.
- Step 2: Create a code tree from the histogram.
- Step 3: Fill a 256-entry code table for each byte value.
- Step 4: Close and reopen the input file in preparation for step 5.
- Step 5: Create a Huffman Coded output file from the input file.

Key functions:

- **‘fill_histogram()‘**: Updates a histogram array with the number of each unique byte value in the input file.
- **‘create_tree()‘**: Creates a Huffman Tree from the histogram and returns a pointer to it.
- **‘fill_code_table()‘**: Fills a code table for each leaf node’s symbol in the Huffman Tree.
- **‘huff_compress_file()‘**: Writes a Huffman Coded file using the provided BitWriter, input buffer, file size, code tree, and code table.

Data Structures

0.5 Node:

Structure: typedef struct Node Node
Fields:

- uint8_t symbol: Represents the symbol associated with the node.
- double weight: Represents the weight of the node.
- uint64_t code: Represents the Huffman code assigned to the node.
- uint8_t code_length: Represents the length of the Huffman code.
- Node *left: Pointer to the left child node.
- Node *right: Pointer to the right child node.

0.6 Priority Queue:

Structure: typedef struct PriorityQueue PriorityQueue
Fields:

- ListElement *list: Pointer to the linked list that represents the priority queue.

0.7 ListElement

Structure: typedef struct ListElement ListElement
Fields:

- Node *tree: Pointer to the tree node stored in the list element.
- ListElement *next: Pointer to the next list element in the linked list.
- Code:

Algorithms

Want to show some pseudocode? Use the framed verbatim text shown above.

```
huff_compress_file(outbuf, inbuf, filesize, num_leaves, code_tree, code_table)
    write 8 'H'
    write 8 'C'
    write 32 filesize
    write 16 num_leaves
    huff_write_tree(outbuf, code_tree)
    for every byte b from inbuf
```

```
        code = code_table[b].code
        code_length = code_table[b].code_length
        for i = 0 to code_length - 1
            /* write the rightmost bit of code */
            1 code & 1
            /* prepare to write the next bit */
            code >>= 1
```

Function Descriptions

0.7.1 BitWriter *bit_write_open(const char *filename)

- Inputs: filename (const char*)
- Outputs: Pointer to a BitWriter struct (BitWriter*)
- Purpose: This function opens the specified file for writing and returns a pointer to a BitWriter struct. If there is an error during the file opening process, it returns NULL.

0.7.2 void bit_write_close(BitWriter **pbuf)

- Inputs: Pointer to a BitWriter struct (BitWriter**)
- Outputs: None
- Purpose: This function flushes any remaining data in the byte buffer, closes the underlying stream, frees the BitWriter object, and sets the pointer to NULL.

0.7.3 void bit_write_bit(BitWriter *buf, uint8_t x)

- Inputs: BitWriter struct (BitWriter*), bit value (uint8_t)
- Outputs: None
- Purpose: This function writes a single bit, x, to the BitWriter struct. It collects 8 bits into the buffer byte before writing the entire buffer to the underlying stream using write_uint8().

0.7.4 void bit_write_uint8(BitWriter *buf, uint8_t x)

- Inputs: BitWriter struct (BitWriter*), 8-bit value (uint8_t)
- Outputs: None
- Purpose: This function writes the 8 bits of the uint8_t value x by calling bit_write_bit() 8 times, starting with the least significant bit (LSB).

0.7.5 void bit_write_uint16(BitWriter *buf, uint16_t x)

- Inputs: BitWriter struct (BitWriter*), 16-bit value (uint16_t)
- Outputs: None
- Purpose: This function writes the 16 bits of the uint16_t value x by calling bit_write_bit() 16 times, starting with the least significant bit (LSB).

0.7.6 void bit_write_uint32(BitWriter *buf, uint32_t x)

- Inputs: BitWriter struct (BitWriter*), 32-bit value (uint32_t)
- Outputs: None
- Purpose: This function writes the 32 bits of the uint32_t value x by calling bit_write_bit() 32 times, starting with the least significant bit (LSB).

0.7.7 Node *node_create(uint8_t symbol, double weight)

- Inputs: symbol (uint8_t), weight (double)
- Outputs: Pointer to the newly created Node (Node*)
- Purpose: This function creates a new Node and sets its symbol and weight fields. It returns a pointer to the newly created Node.

0.7.8 void node_free(Node **node)

- Inputs: Pointer to a Node pointer (Node**)
- Outputs: None
- Purpose: This function frees the memory occupied by the Node pointed to by *node and sets it to NULL.

0.7.9 void node_print_tree(Node *tree, char ch, int indentation)

- Inputs: Root Node of the tree (Node*), character to represent the root (char), indentation level for printing (int)
- Outputs: None
- Purpose: This function is used for diagnostics and debugging. It prints a sideways view of the binary tree, with lines connecting the nodes. It helps visualize the structure of the tree. The root of the tree is represented by the character ch. The indentation parameter determines the level of indentation for printing the nodes.

0.7.10 PriorityQueue *pq_create(void)

- Inputs: None
- Outputs: Pointer to the newly created PriorityQueue (PriorityQueue*)
- Purpose: This function allocates a PriorityQueue object using calloc() and returns a pointer to it. The list data field is automatically initialized to NULL.

0.7.11 void pq_free(PriorityQueue **q)

- Inputs: Pointer to a PriorityQueue pointer (PriorityQueue**)
- Outputs: None
- Purpose: This function frees the memory occupied by the PriorityQueue pointed to by *q and sets it to NULL.

0.7.12 bool pq_is_empty(PriorityQueue *q)

- Inputs: PriorityQueue (PriorityQueue*)
- Outputs: Boolean value indicating whether the queue is empty (true if empty, false otherwise)
- Purpose: This function checks if the PriorityQueue is empty by checking if the list field is NULL. If the list field is NULL, it indicates an empty queue.

0.7.13 bool pq_size_is_1(PriorityQueue *q)

- Inputs: PriorityQueue (PriorityQueue*)
- Outputs: Boolean value indicating whether the queue contains a single element (true if contains a single element, false otherwise)
- Purpose: This function determines if the PriorityQueue contains a single element. In the Huffman Coding algorithm, it is used to check if the queue has reached the termination condition of having a single value.

0.7.14 void enqueue(PriorityQueue *q, Node *tree)

- Inputs: PriorityQueue (PriorityQueue*), Node to be inserted (Node*)
- Outputs: None
- Purpose: This function inserts a tree into the priority queue, ensuring that the tree with the lowest weight is placed at the head of the queue.

0.7.15 bool dequeue(PriorityQueue *q, Node **tree)

- Inputs: PriorityQueue (PriorityQueue*), Pointer to a Node pointer (Node**)
- Outputs: Boolean value indicating the success of dequeuing (true if successful, false if the queue is empty)
- Purpose: This function removes the queue element with the lowest weight, sets *tree to point to it, frees the memory occupied by the element, and returns true. If the queue is empty, it returns false.

0.7.16 void pq_print(PriorityQueue *q)

- Inputs: PriorityQueue (PriorityQueue*)
- Outputs: None
- Purpose: This diagnostic function prints the trees of the queue in a specified format.

Pseudocode:

```
Assert that q is not NULL.
Set e to q->list.
Set position to 1.
While e is not NULL:
  If position is 1, print a separator line.
  Else, print a different separator line.
  Call node_print_tree(e->tree, '<', 2).
  Set e to e->next.
Print the final separator line.
```

0.7.17 bool pq_less_than(Node *n1, Node *n2)

- Inputs: Two Node pointers (Node*, Node*)
- Outputs: Boolean value indicating whether n1 is less than n2 based on weight and symbol tie-breaker (true if n1 is less than n2, false otherwise)
- Purpose: This function is used in the enqueue operation to compare the weights of two Node objects. It includes a tie-breaker based on the symbol value to ensure deterministic program operation.

0.7.18 uint64_t fill_histogram(Buffer *inbuf, double *histogram)

- Inputs: Input buffer (Buffer*), Histogram array (double*)
- Outputs: Total size of the input file (uint64_t)
- Purpose: Updates the histogram array with the number of occurrences of each unique byte value in the input file and returns the total size of the file.

0.7.19 Node *create_tree(double *histogram, uint16_t *num_leaves)

- Inputs: Histogram array (double*), Pointer to store the number of leaf nodes (uint16_t*)
- Outputs: Pointer to the new Huffman Tree (Node*)
- Purpose: Creates a new Huffman Tree based on the histogram array and returns a pointer to it. Also returns the number of leaf nodes in the tree.

0.7.20 void fill_code_table(Code *code_table, Node *node, uint64_t code, uint8_t code_length)

- Inputs: Code Table array (Code*), Node of the Huffman Tree (Node*), Huffman code for the node (uint64_t), Length of the Huffman code (uint8_t)
- Outputs: None
- Purpose: Recursively traverses the Huffman Tree and fills in the Code Table for each leaf node's symbol.

Pseudocode:

0.7.21 void huff_compress_file(BitWriter *outbuf, Buffer *inbuf, uint32_t filesize, uint16_t num_leaves, Node *code_tree, Code *code_table)

- Inputs: Output buffer (BitWriter*), Input buffer (Buffer*), File size (uint32_t), Number of leaves (uint16_t), Code Tree root (Node*), Code Table (Code*)
- Outputs: None
- Purpose: Writes a Huffman Coded file using the provided parameters.

0.7.22 void huff_write_tree(BitWriter *outbuf, Node *node)

- Inputs: Output buffer (BitWriter*), Node of the Huffman Tree (Node*)
- Outputs: None
- Purpose: Recursively writes the code tree to the output buffer.

Results

The result should be a file that is compressed and then decompress. The expectation is that these files will math exactly the original/