

# Assignment 6 – Huffman Coding

Rahul Amudhasagaran

CSE 13S – Spring 2023

## Purpose

The purpose of this program is to use Huffman Coding to compress a data file while being extremely memory efficient.

## How to Use the Program

To use this program, the user must run the program with an input and output file selected. The user can also select the help option for more directions on how to use the program.

```
Options:  
-i: sets the file to read from FILE  
-o: sets the file to write to as FILE  
-h: help message
```

Running the help function will exit the program. Otherwise, the user should specify which input and output file they will select.

These descriptions were given by the Lab Document<sup>1</sup>.

## Program Design

There are 4 major parts to this program: the bit writer that acts as a buffer, the binary tree that stores the values, the priority queue that fills up the binary tree, and the main file that runs all the functions.

## Data Structures

There are three main data structures used in this program. One of the data structures is a bit writer that uses a byte long buffer to store and write information from a data file. This is what is used to extract and export information from and to files. The next data structure is the priority queue. This priority queue takes the information extracted from the bit writer and places it in a comprehensible and useful order. The last data structure is the binary tree. This tree is where the actual encoding and decoding of the data is done.

## Algorithms

This program does not use any algorithms as of yet.

---

<sup>1</sup>Assignment 6 Huffman Coding by Kerry Veenstra edited by Ben Grant[1]

---

## Function Descriptions

There are many functions used in this program. I will sort them out based on which section they are part of.

### Bit Writer Functions

```
BitWriter *bit_write_open (const char *filename)
Inputs: name of the file
Outputs: new Bit Writer
Purpose: This function enables the write option for the bit writer.

void bit_write_close (BitWriter **pbuf)
Inputs: Pointer to the pointer of the Bit Writer
Outputs: None
Purpose: This function closes the write option for the bit writer and frees the buffer.

void bit_write_bit (BitWriter *buf, uint8_t x)
Inputs: Pointer to the bit writer, 8 bit unsigned integer value
Outputs: None
Purpose: This function writes one bit to the buffer.

Psuedo Code:
if (buf -> bit_position > 7)
    write_uint8 (buf -> underlying_stream, buf -> byte)
    buf -> byte = 0
    buf -> bit_position = 0
if (x & 1) buf -> byte |= (x & 1) << buf -> bit_position
++buf -> bit_position

void bit_write_uint8 (BitWriter *buf, uint8_t x)
Inputs: Pointer to the bit writer, 8 bit unsigned integer value
Outputs: None
Purpose: This function writes one byte to the buffer.

void bit_write_uint16 (BitWriter *buf, uint16_t x)
Inputs: Pointer to the bit writer, 16 bit unsigned integer value
Outputs: None
Purpose: This function writes two bytes to the buffer.

void bit_write_uint32 (BitWriter *buf, uint32_t x)
Inputs: Pointer to the bit writer, 32 bit unsigned integer value
Outputs: None
Purpose: This function writes four bytes to the buffer.
```

### Binary Tree Functions

```
Node *node_create (uint8_t symbol, double weight)
Inputs: 8 bit unsigned integer value, weight of node
Outputs: New Node
Purpose: This function creates a node.

void node_free (Node **node)
Inputs: Pointer to the pointer of the node
Outputs: None
Purpose: This function frees the node.
```

```

void node_print_tree (Node *tree, char ch, int indentation)
Inputs: Node, character value, indentation
Outputs: None
Purpose: This function prints the binary tree.

Psuedo Code:
    if (tree == NULL) return
    node_print_tree (tree -> right, '/', indentation + 3)
    printf ("%cweight = %.0f", indentation + 1, ch, tree -> weight)
    if (tree -> left == NULL && tree -> right == NULL)
        if (' ' <= tree -> symbol && tree -> symbol <= '~')
            printf ("", symbol = '%c', tree -> symbol)
        else
            printf ("", symbol = 0x%02x", tree -> symbol)
    printf ("\n")
    node_print_tree (tree -> left, '\\', indentation + 3)

```

### Priority Queue Functions

```

PriorityQueue *pq_create (void)
Inputs: None
Outputs: new Priority Queue
Purpose: This function creates a new priority queue.

void pq_free (PriorityQueue **q)
Inputs: Pointer to the pointer of the Priority Queue
Outputs: None
Purpose: This function frees the Priority Queue.

bool pq_is_empty (PriorityQueue *q)
Inputs: Pointer to the Priority Queue
Outputs: if Priority Queue is empty
Purpose: This function checks if the Priority Queue is empty.

bool pq_size_is_1 (PriorityQueue *q)
Inputs: Pointer to the Priority Queue
Outputs: if Priority Queue is size 1
Purpose: This function checks if the Priority Queue has only one value.

void enqueue (PriorityQueue *q, Node *tree)
Inputs: Pointer to the Priority Queue, Pointer to the binary tree
Outputs: None
Purpose: This function enqueues an item from the binary tree to the Priority Queue
and sorts based on increasing order of priority.

Psuedo Code:
    ListElement *e = (ListElement *) calloc (1, sizeof (ListElement))
    e -> tree = tree
    if (pq_is_empty (q))
        q -> list = e
    else if (pq_less_than (e -> tree, q -> list -> tree))
        e -> next = q -> list
        q -> list = e

```

```

else
    ListElement *i
    for (i = q -> list; i -> next != NULL; i = i -> next)
        if (pq_less_than (e -> tree, i -> next -> tree))
            e -> next = i -> next
            i -> next = e
            return
    i -> next = e

```

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

Inputs: Pointer to the Priority Queue, Pointer to the pointer of the binary tree

Outputs: Check if the Priority Queue is empty

Purpose: This function dequeues an item from the Priority Queue.

Psuedo Code:

```

if (pq_is_empty (q)) return false
ListElement *e
e = q -> list
q -> list = q -> list -> next
*tree = e -> tree
free (e)
return true

```

void pq\_print (PriorityQueue \*q)

Inputs: Pointer to the Priority Queue

Outputs: None

Purpose: This function prints the Priority Queue.

bool pq\_less\_than (Node \*n1, Node \*n2)

Inputs: Two Pointers to two nodes

Outputs: Check if Node 1 is less than Node 2

Purpose: This function checks if one node's value is less than the other.

#### Huffman Coding Functions (main)

uint64\_t fill\_histogram (Buffer \*inbuf, double \*histogram)

Inputs: Pointer to the buffer, pointer to an array of doubles

Outputs: total size of the file

Purpose: This function fills a histogram of frequencies of values of the Huffman Code.

Psuedo Code:

```

for (uint16_t i = 0; i < 256; ++i)
    histogram [i] = 0
uint8_t byte
uint64_t filesize = 0
while (read_uint8 (inbuf, &byte))
    ++histogram [byte]
    ++filesize;
if (filesize == 0)
    ++histogram[0x00]
    ++histogram[0xff]
return filesize

```

Node \*create\_tree (double \*histogram, uint16\_t \*num\_leaves)

---

Inputs: array of doubles, number of leaf nodes in the tree

Outputs: pointer to a new Huffman Tree

Purpose: This function constructs a new Huffman Tree.

Pseudo Code:

```
*num_leaves = 0
PriorityQueue *pq = pq_create ()
for (uint16_t i = 0; i < 256; ++i)
    if (histogram [i])
        Node *n = node_create (i, histogram [i])
        ++(*num_leaves)
        enqueue (pq, n)
Node *left
Node *right
while (!pq_size_is_1 (pq))
    dequeue (pq, &left)
    dequeue (pq, &right)
    Node *p = node_create (0, left -> weight + right -> weight)
    p -> left = left
    p -> right = right
    enqueue (pq, p)
dequeue (pq, &left)
pq_free (&pq)
return left
```

void fill\_code\_table (Code \*code\_table, Node \*node, uint64\_t code, uint8\_t code\_length)

Inputs: Pointer to a code table, pointer to the binary tree, value, length of traversal.

Outputs: None

Purpose: This function traverses the Huffman tree and assigns values to each node.

Pseudo Code:

```
if (node -> left)
    fill_code_table (code_table, node -> left, code, code_length + 1)
    code |= 1 << code_length
    fill_code_table (code_table, node -> right, code, code_length + 1)
else
    code_table [node -> symbol].code = code
    code_table [node -> symbol].code_length = code_length
```

void huff\_compress\_file (BitWriter \*outbuf, Buffer \*inbuf, uint32\_t filesize, uint16\_t num\_leaves, Node \*code\_tree, Code \*code\_table)

Inputs: input and output buffer, file size, number of leaves, Huffman Tree, code table

Outputs: None

Purpose: This function writes a Huffman Coded File.

Pseudo Code:

```
bit_write_uint8 (outbuf, 'H')
bit_write_uint8 (outbuf, 'C')
bit_write_uint32 (outbuf, filesize)
bit_write_uint16 (outbuf, num_leaves)
huff_write_tree (outbuf, code_tree)
uint8_t byte
uint64_t code
uint8_t code_length
```

```

while (read_uint8 (inbuf, &byte))
    code = code_table [byte].code
    code_length = code_table [byte].code_length
    for (uint8_t i = 0; i < code_length; ++i)
        bit_write_bit (outbuf, code & 1)
        code >>= 1

void huff_write_tree (BitWriter *outbuf, Node *code_tree)
Inputs: output buffer, Huffman Tree
Outputs: None
Purpose: This function writes the Huffman tree to the Huffman Coded File.

Psuedo Code:
    if (node -> left)
        huff_write_tree (outbuf, node -> left)
        huff_write_tree (outbuf, node -> right)
        bit_write_bit (outbuf, 0)
    else
        bit_write_bit (outbuf, 1)
        bit_write_uint8 (outbuf, node -> symbol)

void free_tree (Node **code_tree)
Inputs: pointer to the Pointer of the code tree
Outputs: None
Purpose: This function frees the Huffman tree.

Psuedo Code:
    if ((*code_tree) -> left)
        free_tree (&((*code_tree) -> left))
        free_tree (&((*code_tree) -> right))
    node_free (code_tree)

int main (int argc, char **argv)
Inputs: argument count and arguments
Outputs: exit code
Purpose: This function runs all the other functions.

```

## Results

This code functioned perfectly. There were no errors while running, and the code simulated Huffman Coding. There are some areas that could be more efficient. An example would be when I free the tree. I realized that instead of DFS, I could use reverse recursion instead because if there is a child node, there will always be only one right child node. This way it would be much faster. The results are shown in Figures 1 and 2.

## Error Handling

There was some error handling that was used in this program. I caught when the program tried to accept an invalid file. This program caught this and handled it properly. The print help function is also working as intended, as shown in Figure 3.

---

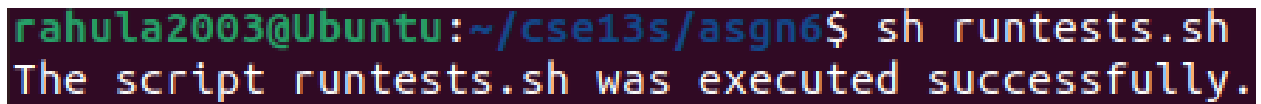
A terminal window with a dark background and light-colored text. The prompt is 'rahula2003@Ubuntu:~/cse13s/asn6\$'. The command 'sh runtests.sh' has been entered, and the output is 'The script runtests.sh was executed successfully.'

Figure 1: Screenshot of bash script running successfully

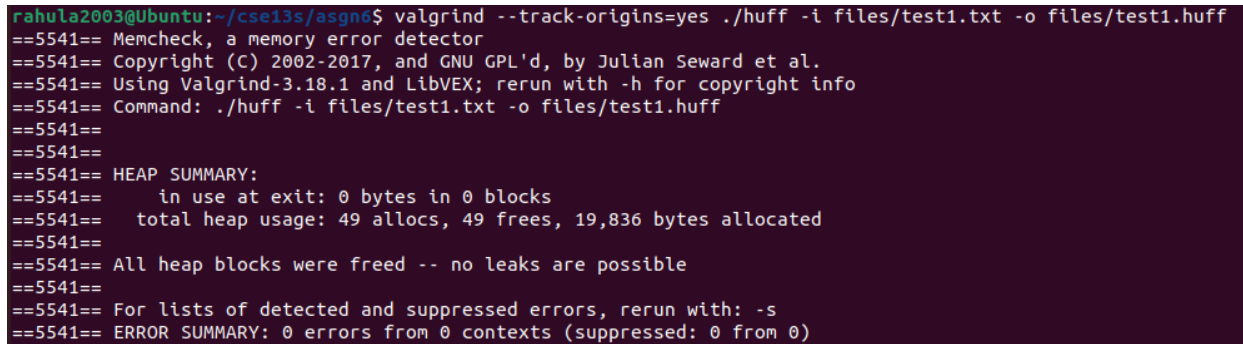
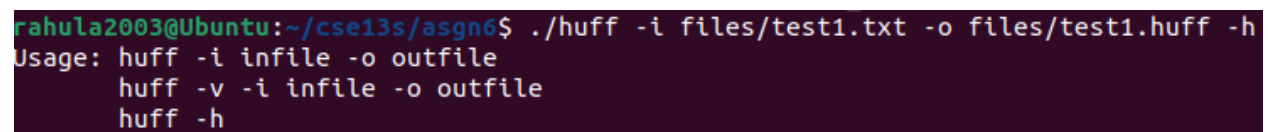
A terminal window showing the output of a valgrind command. The command is 'valgrind --track-origins=yes ./huff -i files/test1.txt -o files/test1.huff'. The output includes copyright information for Memcheck, a summary of heap usage (49 allocs, 49 frees, 19,836 bytes allocated), and a confirmation that all heap blocks were freed with no leaks possible. The error summary shows 0 errors from 0 contexts.

Figure 2: Screenshot of valgrind showing no memory leaks

## References

- [1] Kerry Veenstra, Jess Srinivas. Assignment 6 huffman coding. <https://git.ucsc.edu/cse13s/spring2023/resources/-/blob/master/asn6/asn6.pdf>, 2023. [Online; accessed 2-June-2023].

---

A terminal window with a dark purple background. The prompt is 'rahula2003@Ubuntu:~/cse13s/asgn6\$'. The command entered is './huff -i files/test1.txt -o files/test1.huff -h'. The output shows the usage of the huff program: 'Usage: huff -i infile -o outfile', 'huff -v -i infile -o outfile', and 'huff -h'.

```
rahula2003@Ubuntu:~/cse13s/asgn6$ ./huff -i files/test1.txt -o files/test1.huff -h
Usage: huff -i infile -o outfile
       huff -v -i infile -o outfile
       huff -h
```

Figure 3: Screenshot of help function