

Assignment 6 Report

DRAFT

By Jason Zhang
Prof. Veenstra
CSE 13S

Due June 2, 2023

Purpose

This program performs Huffman Coding by using unbuffered file-I/O functions to read and write bytes files into and out of C data structures. These data structures include abstract data types like `bit writer`, `binary tree`, `priority queue`. The program will then evaluate the occurrence of bytes and switch more common bytes to be represented in fewer bits and compensate by making less common bytes to be represented in more bits. These functionalities will be accessed through a series of files including `bitwriter.c`, `huff.c`, `node.c`, `pq.c`, of which the harness takes in command line options and performs the desired operations.

How to Run

Run the program from command line using a Makefile:

```
make
```

The program can then be ran directly:

```
./huff
```

Or with an input txt file:

```
./huff < input.txt > output.txt
```

Possible command line options for `sorting.c` include:

```
./huff -i -o -h
```

Of which,

- i : Sets the file to read from (input file). Requires a filename as an argument.
- o : Sets the file to write to (output file). Requires a filename as an argument.
- h : Prints a help message to `stdout`.

Program Design

The source code for the Huffman Coding program is found in 4 main files. Of which, `pq.c`, `node.c`, contain the data structures used. `bitwriter.c` is mainly composed of functions that help read, write bytes from other files using unbuffered file I/O functions. The `huff.c` file

contains a `main()` function that the program runs with. In addition, the program uses existing libraries from C such as `stdio.h` and `unistd.h`.

Data Structures

The program utilizes a `BitWriter` struct that acts as a buffer for file I/O.

```
/* put in bitwriter.h */
typedef struct BitWriter BitWriter;
```

```
/* put in bitwriter.c */
#include "io.h"
struct BitWriter {
    Buffer *underlying_stream;
    uint8_t byte;
    uint8_t bit_position;
};
```

The program also uses a customary `Node` structure to store data, as well as `uint64_t` for code and `uint8_t` for associated parameters. Additionally, a priority queue data structure is made to help with Huffman Coding. Pseudocode as follows:

```
/* put everything in node.h */
typedef struct Node Node;
```

```
struct Node {
    uint8_t symbol;
    double weight;
    uint64_t code;
    uint8_t code_length;
    Node *left;
    Node *right;
};
```

```
/* put in pq.h */
typedef struct PriorityQueue PriorityQueue;
```

```
/* put in pq.c */
typedef struct ListElement ListElement;
```

```
struct ListElement {
    Node *tree;
    ListElement *next;
};
```

```
struct PriorityQueue {
    ListElement *list;
};
```

Algorithms

The huff program mainly utilizes these following steps:

1. Create and fill a priority queue.
2. Run the Huffman Coding algorithm
3. Dequeue the queue's only entry and return it.

Function Descriptions

Each data structure algorithm performs steps below:

- **BitWriter functions**
- **BitWriter *bit_write_open(const char *filename);**
 - input: string filename
 - output: BitWriter *
 - **def bit_write_open():**
 - Allocate a BitWriter object, BitWriter *buf.
 - Create Buffer *underlying_stream using write_open().
 - buf->underlying_stream = underlying_stream
 - Return a pointer to the new BitWriter object, buf.
 - If unable to perform any of the prior steps, report an error and end the program.
- **void bit_write_close(BitWriter **pbuf);**
 - input: BitWriter ** pbuf
 - output: void
 - **def bit_write_close():**
 - if bit_position > 0:
 - write byte to underlying_stream using write_uint8()
 - write_close() underlying_stream
 - free *pbuf
 - *pbuf = NULL
- **void bit_write_bit(BitWriter *buf, uint8_t x);**
 - input: BitWriter *buf, uint8_t x
 - output: void
 - **def bit_write_bit():**
 - if bit_position > 7:
 - write byte to underlying_stream using write_uint8()

- ```

 clear byte to 0x00
 clear bit_position to 0
 if x & 1 then byte |= (x & 1) << bit_position
 ++bit_position;

```
- **void bit\_write\_uint8(BitWriter \*buf, uint8\_t x);**
    - input: BitWriter \* buf, uint8\_t x
    - output: void
    - ```
def bit_write_uint8():
    for i = 0 to 7:
        write bit i of x using bit_write_bit()
```
 - **void bit_write_uint16(BitWriter *buf, uint16_t x);**
 - input: BitWriter * buf, uint16_t x
 - output: void
 - ```
def bit_write_uint16():
 for i = 0 to 15:
 write bit i of x using bit_write_bit()
```
  - **void bit\_write\_uint32(BitWriter \*buf, uint32\_t x);**
    - input: BitWriter \* buf, uint32\_t x
    - output: void
    - ```
def bit_write_uint32():
    for i = 0 to 31:
        write bit i of x using bit_write_bit()
```
 - **Node Functions**
 - **Node *node_create(uint8_t symbol, double weight);**
 - input: uint8_t symbol, double weight
 - output: Node *
 - ```
def node_create():
 allocate memory for Node
 set symbol
 set weight
 return pointer
```
  - **void node\_free(Node \*\*node);**
    - input: Node \*\*node
    - output: void
    - ```
def node_free()
    free *node
    set pointer to NULL
```
 - **void node_print_tree(Node *tree, char ch, int indentation);**
 - input: Node *tree, char ch, int indentation

- output: void
- `def node_print_tree()`
 - `if (tree == NULL) return;`
 - `node_print_tree(tree->right, '/', indentation + 3);`
 - `print(weight, indentation + 1);`
 - `if (tree->left == NULL and tree->right == NULL):`
 - `if (' ' <= tree->symbol and tree->symbol <= '~'):`
 - `print(symbol)`
 - `else:`
 - `print(tree->symbol)`
 - `print("\n");`
 - `node_print_tree(tree->left, '\\', indentation + 3)`

- **Priority Queue**

- **PriorityQueue *pq_create(void);**

- input: void
- output: PriorityQueue *
- `def pq_create():`
 - `calloc() memory for PriorityQueue object`
 - `return pointer`

- **void pq_free(PriorityQueue **q);**

- input: PriorityQueue **q
- output: void
- `def pa_free():`
 - `free(*q)`
 - `*q = NULL`

- **bool pq_is_empty(PriorityQueue *q);**

- input: PriorityQueue **q
- output: bool
- `def pq_is_empty():`
 - `check if NULL in queue's list field`
 - `return true if so, false otherwise`

- **bool pq_size_is_1(PriorityQueue *q);**

- input: PriorityQueue *q
- output: bool
- `def pq_size_is_1():`
 - `check if priority queue had only 1 value`

- **void enqueue(PriorityQueue *q, Node *tree);**

- input: PriorityQueue *q, Node *tree
- output: void

- `def enqueue():`
 - Allocate new ListElement *e
 - set tree = tree
 - if queue is empty:
 - q->list = e
 - if tree->weight < head->weight:
 - insert element at beginning of queue
 - e->next = q->list
 - q->list = e
 - if new element goes after an existing one:
 - find it
- **void dequeue(PriorityQueue *q, Node **tree);**
 - input: PriorityQueue *q, Node **tree
 - output: void
 - `def dequeue():`
 - if queue is empty:
 - return false
 - remove element with lowest weight
 - *e = lowest weight
 - *tree = e->tree
 - free(e)
 - return true
- **void pq_print(PriorityQueue *q);**
 - input: PriorityQueue *q
 - output: void
 - `def pq_print():`
 - prints all trees of priority queue
- **bool pq_less_than(Node *n1, Node *n2)**
 - input: Node *n1, Node *n2
 - output: bool
 - `def pq_less_than():`
 - if (n1->weight < n2->weight) return true;
 - if (n1->weight > n2->weight) return false;
 - return n1->symbol < n2->symbol;
- **Huffman Coding Functions**
- **uint64_t fill_histogram(Buffer *inbuf, double *histogram)**
 - input: Buffer *inbuf, double *histogram
 - output: uint64_t
 - `def fill_histogram():`
 - clear all elements of histogram
 - read bytes from inbuf with read_uint8()

- ```

 ++histogram[byte] for each byte read
 ++filesize

```
- **Node \*create\_tree(double \*histogram, uint16\_t \*num\_leaves)**
    - input: double \*histogram, uint16\_t \*num\_leaves
    - output: node \*
    - def create\_tree():

```

 create and fill priority queue
 run Huffman Coding algorithm:
 while (Priority Queue has more than one entry):
 Dequeue into left
 Dequeue into right
 Create a new node with a weight = left->weight +
 right->weight
 node->left = left
 node->right = right
 Enqueue the new node
 Dequeue and return

```
  - **fill\_code\_table(Code \*code\_table, Node \*node, uint64\_t code, uint8\_t code\_length)**
    - input: double \*histogram, uint16\_t \*num\_leaves
    - output: node \*
    - def fill\_code\_table():

```

 if node is internal
 /* Recursive calls left and right. */
 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
 /* Leaf node: store the Huffman Code. */
 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)**
    - input: double \*histogram, uint16\_t \*num\_leaves
    - output: node \*
    - def huff\_compress\_file():

```

 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

```

## Error Handling

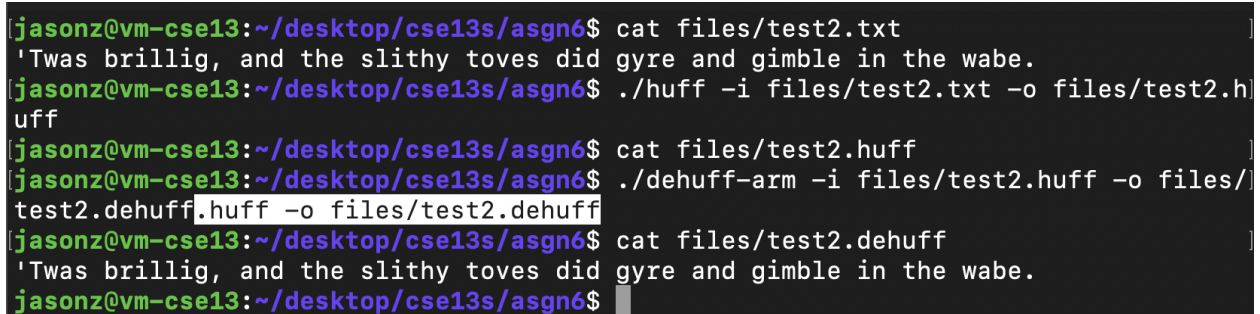
`getopt()` includes error handling where in the event that the user's input option does not exist, an error message will be displayed. Functions for the custom data structures also contain error checking to make sure pointers passed in aren't NULL. Destructive functions for data structures involving heap memory allocation are thoroughly tested through valgrind to ensure no memory leaks.

## Result

Program works as intended. Valgrind: All heap blocks were freed -- no leaks possible. No false positives reported by scan-build. No known bugs or errors.

What I learned:

- Huffman Coding: Gained a better understanding of Huffman Coding as well as how to determine the most common bytes in a file through histograms.
- Bit Writer: How to use abstract data type `BitWriter` to handle writing individual bits to a file and write compressed data in bit form.
- Priority Queue: How to use abstract data type `Priority Queue` to efficiently select symbols with the highest frequency during the Huffman Coding process through enqueue and dequeue.



```

[jasonz@vm-cse13:~/desktop/cse13s/asgn6$ cat files/test2.txt
'Twas brillig, and the slithy toves did gyre and gimble in the wabe.
[jasonz@vm-cse13:~/desktop/cse13s/asgn6$./huff -i files/test2.txt -o files/test2.huff
[jasonz@vm-cse13:~/desktop/cse13s/asgn6$ cat files/test2.huff
[jasonz@vm-cse13:~/desktop/cse13s/asgn6$./dehuff-arm -i files/test2.huff -o files/test2.dehuff
[jasonz@vm-cse13:~/desktop/cse13s/asgn6$ cat files/test2.dehuff
'Twas brillig, and the slithy toves did gyre and gimble in the wabe.
[jasonz@vm-cse13:~/desktop/cse13s/asgn6$

```

Figure 1. Screenshot of huffman encoded and decoded message.