Assignment 6 - Huffman Coding

Chucheng Xie

CSE 13S – Spring 2023

Purpose

In this assignment, we will write a data compressor, huff, that computes the Huffman code of an input file. We will be provided with a program, dehuff, that decompresses a Huffman Coded file. We'll also be provided with several unit-test programs: bwtest.c, nodetest.c, and pqtest.c.

We will need to create a "bit writer" abstract data type, a binary tree abstract data type, a priority queue abstract data type and compress a data file using Huffman Coding.

Program Design

Pseudocode:

bitwriter.c:

```
struct BitWriter;

BitWriter *bit_write_open(const char *filename) {
   buf = malloc(sizeof(BitWriter));
   buf -> underlying_stream = write_open(filename);
   buf -> byte = 0;
   buf -> bit_position = 0;
   return buf;
}

void bit_write_close(BitWriter **pbuf) {
   BitWriter *buf = *pbuf;
   if (buf -> bit_positon) > 0
      write_uint8(buf -> underlying_stream, buf -> byte)
   write_close(&(buf -> underlying_stream));
   free(buf);
```

```
    *pbuf = NULL;
}

void bit_write_bit(BitWriter *buf, uint8_t bit) {
    if (buf -> bit_position > 7)
        write_uint8(buf -> underlying_stream, buf -> byte);
        buf -> byte = 0x00;
        buf -> bit_position = 0;
    if (bit & 1)
        buf -> byte |= (bit & 1) << buf -> bit_position;
    ++(buf -> bit_position);
}

void bit_write_uint8(BitWriter *buf, uint8_t byte) {
    for (int i = 0; i < 8; i++)
        bit_write_bit(buf, byte & 1);
        byte >>= 1;
}

void bit_write_uint16(BitWriter *buf, uint16_t x) {
    for (int i = 0; i < 16; i++)
        bit_write_bit(buf, x & 1);
        x >> = 1;
}

void bit_write_uint32(BitWriter *buf, uint32_t x) {
    for (int i = 0; i < 32; i++)
        bit_write_bit(buf, x & 1);
        x >>= 1;
}
```

node.c:

```
Node *node_create(uint8_t symbol, double weight) {
    Node *node = malloc(sizeof(Node));
    node -> symbol = symbol;
    node -> weight = weight;
    node -> code = 0;
    node -> code_length = 0;
    node -> left = NULL;
    node -> right = NULL;
    return node;
}

void node_free(Node **node) {
    free(*node);
    *node = NULL;
}
```

pq.c:

```
typedef struct ListElement ListElement;
struct ListElement { Node *tree; ListElement *next; };
struct PriorityQueue { ListElement *list; };
PriorityQueue *pq_create(void) {
    PriorityQueue *new_queue = (PriorityQueue *)calloc(1, sizeof(PriorityQueue));
    return new_queue;
}

void pq_free(PriorityQueue **q) {
    free(*q);
    *q = NULL;
}

bool pq_is_empty(PriorityQueue *q) {
    return q -> list == NULL;
}
```

```
bool pq_size_is_1(PriorityQueue *q) {
  return (q != NULL) && (q -> list != NULL) && (q -> list -> next == NULL);
}

void enqueue(PriorityQueue *q, Node *tree) {
  ListElement *new_element = (ListElement *)malloc(sizeof(ListElement));
  new_element -> tree = tree;
  if (q -> list == NULL) {
    q -> list = new_element;
  } else if (tree -> weight < q -> list -> tree -> weight) {
    new_element -> next = q -> list;
    q -> list = new_element;
  } else {
    ListElement *current = q -> list;
    while (current -> next != NULL && current -> next -> tree -> weight <= tree -> weight)
{
      current = current -> next;
    }
    new_element -> next = current -> next;
    current -> next = new_element;
  }
}

bool dequeue(PriorityQueue *q, Node **tree) {
  if (q == NULL || q -> list == NULL) { return false; }
  ListElement *element = q -> list;
  q -> list = element -> next;
  *tree = element -> tree;
  free(element);
  return true;
}
```

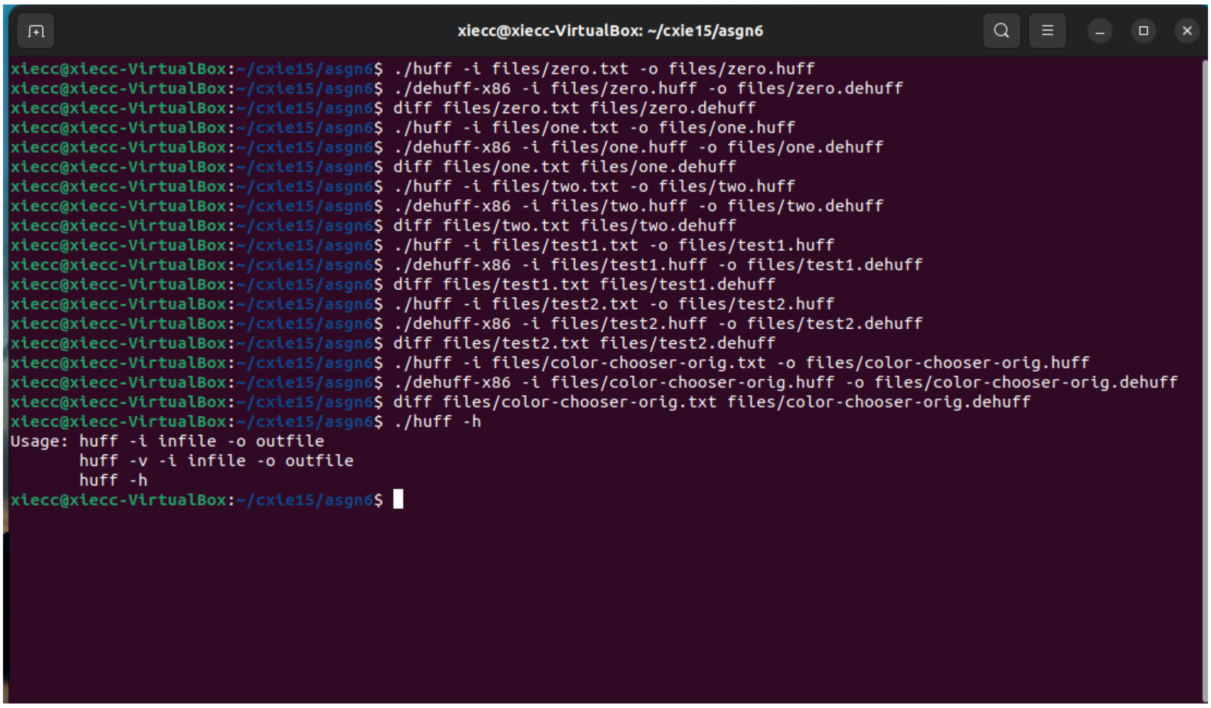
huff.c:

```
typedef struct Code {
    uint64_t code;
    uint8_t code_length;
} Code;

uint64_t fill_histogram(Buffer *inbuf, double *histogram) {
    for (int i = 0; i < 256; i++)
        histogram[i] = 0.0;
    uint64_t filesize = 0;
    uint8_t byte;
    while (read_uint8(inbuf, &byte))
        histogram[byte]++;
        filesize++:
    histogram[0x00]++;
    histogram[0xff]++;
    return filesize;
}

Node *create_tree(double *histogram, uint16_t *num_leaves) {
      1. Create and fill a Priority Queue.
      2. Run the 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
      3. Dequeue the queue's only entry and return it.
}

fill_code_table(Code *code_table, Node *node, uint64_t code, uint8_t code_length) {
    if node is internal
        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;
}

huff_compress_file(outbuf, inbuf, filesize, num_leaves, code_tree, code_table) {
    8 | ‘H’
    8 | ‘C’
    32 | filesize
    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
            1 | code & 1
            code >>= 1
}

huff_write_tree(outbuf, node) {
    if node is an internal node
        huff_write_tree(node -> left)
        huff_write_tree(node -> right)
        1 | 0
    else
        1 | 1
        8 | node -> symbol
}
```

Result

From this assignment, I have written a data compressor, huff, that computes the Huffman code of an input file. I also created a “bit writer” abstract data type, a binary tree abstract data type, a priority queue abstract data type and compressed a data file using Huffman Coding. The files encrypted with the huff program written by myself can finally be decrypted with the dehuff program provided, and there is no difference between the decrypted file and the unencrypted file.

Program simple output:



```
xiecc@xiecc-VirtualBox:~/cxie15/asgn6$ ./huff -i files/zero.txt -o files/zero.huff
xiecc@xiecc-VirtualBox:~/cxie15/asgn6$ ./dehuff-x86 -i files/zero.huff -o files/zero.dehuff
xiecc@xiecc-VirtualBox:~/cxie15/asgn6$ diff files/zero.txt files/zero.dehuff
xiecc@xiecc-VirtualBox:~/cxie15/asgn6$ ./huff -i files/one.txt -o files/one.huff
xiecc@xiecc-VirtualBox:~/cxie15/asgn6$ ./dehuff-x86 -i files/one.huff -o files/one.dehuff
xiecc@xiecc-VirtualBox:~/cxie15/asgn6$ diff files/one.txt files/one.dehuff
xiecc@xiecc-VirtualBox:~/cxie15/asgn6$ ./huff -i files/two.txt -o files/two.huff
xiecc@xiecc-VirtualBox:~/cxie15/asgn6$ ./dehuff-x86 -i files/two.huff -o files/two.dehuff
xiecc@xiecc-VirtualBox:~/cxie15/asgn6$ diff files/two.txt files/two.dehuff
xiecc@xiecc-VirtualBox:~/cxie15/asgn6$ ./huff -i files/test1.txt -o files/test1.huff
xiecc@xiecc-VirtualBox:~/cxie15/asgn6$ ./dehuff-x86 -i files/test1.huff -o files/test1.dehuff
xiecc@xiecc-VirtualBox:~/cxie15/asgn6$ diff files/test1.txt files/test1.dehuff
xiecc@xiecc-VirtualBox:~/cxie15/asgn6$ ./huff -i files/test2.txt -o files/test2.huff
xiecc@xiecc-VirtualBox:~/cxie15/asgn6$ ./dehuff-x86 -i files/test2.huff -o files/test2.dehuff
xiecc@xiecc-VirtualBox:~/cxie15/asgn6$ diff files/test2.txt files/test2.dehuff
xiecc@xiecc-VirtualBox:~/cxie15/asgn6$ ./huff -i files/color-chooser-orig.txt -o files/color-chooser-orig.huff
xiecc@xiecc-VirtualBox:~/cxie15/asgn6$ ./dehuff-x86 -i files/color-chooser-orig.huff -o files/color-chooser-orig.dehuff
xiecc@xiecc-VirtualBox:~/cxie15/asgn6$ diff files/color-chooser-orig.txt files/color-chooser-orig.dehuff
xiecc@xiecc-VirtualBox:~/cxie15/asgn6$ ./huff -h
Usage: huff -i infile -o outfile
       huff -v -i infile -o outfile
       huff -h
xiecc@xiecc-VirtualBox:~/cxie15/asgn6$ 
```