CSE13s Fall 2021
Assignment 5:- Huffman Coding
By: Ruhin Gharai

The purpose of the program

The purpose of this assignment is to make a Huffman encoder and a Huffman decoder. This encoder will read in an input file, find the Huffman encoding of its contents, and use the encoding to compress the file. This decoder will read in a compressed input file and decompress it, expanding it back to its original, uncompressed size. There are multiple files that will help to implement these tasks of the encoder and the decoder.

Nodes

The first file is a node.c:- Huffman trees are composed of nodes, with each node containing a pointer to its left child, a pointer to its right child, a symbol, and the frequency of that symbol. The node's frequency is only needed for the encoder.

```
struct Node {
        Node * left ; // Pointer to left child .
        Node * right ; // Pointer to the right child .
        uint8_t symbol ; // Node 's symbol .
        uint64_t frequency ; // Frequency of symbol .
 };
```

Node *node_create(uint8_t symbol, uint64_t frequency)

The constructor for a node. Sets the node's symbol as a symbol and its frequency as frequency.

```
Node *n = (Node *) malloc(sizeof(Node))
Set n->symbol and n->frequency
Return n
}
```

void node_delete(Node **n)

The destructor for a node. Make sure to set the pointer to NULL after freeing the memory for a node.

free(n)
n = NULL
Returnl

Node *node_join(Node *left, Node *right)

Joins a left child node and right child node, returning a pointer to a created parent node. The parent node's left child will be left and its right child will be right. The parent node's symbol will be '$' and its frequency the sum of its left child's frequency and its right child's frequency.

Node *n = (Node *) malloc(sizeof(Node));
Set symbol to $
Set frequency to the sum of left and the right frequency
Set left and right to left node and right node
Return n

void node_print(Node *n)

A debug function to verify that your nodes are created and joined correctly.

Print node symbol and frequency

Priority Queues

The second file is pq. c:- the encoder will make use of a priority queue of nodes. A priority queue functions as a regular queue but assigns each of its elements a priority, such that elements with a high priority are dequeued before elements with a low priority

PriorityQueue *pq_create(uint32_t capacity)

The constructor for a priority queue. The priority queue's maximum capacity is specified by capacity.

PriorityQueue *pq = (PriorityQueue *) malloc(sizeof(PriorityQueue));
Set size to 0 and capacity to capacity
Malloc capacity*sizeof(Node) to items
Return pq

void pq_delete(PriorityQueue **q)

The destructor for a priority queue. Make sure to set the pointer to NULL after freeing the memory for a priority queue.

free memory in q
q = NULL
Return;

bool pq_empty(PriorityQueue *q)

Returns true if the priority queue is empty and false otherwise.

return q->size == 0

bool pq_full(PriorityQueue *q)

Returns true if the priority queue is full and false otherwise.

return q->size == q->capacity

uint32_t pq_size(PriorityQueue *q)

Returns the number of items currently in the priority queue.

return q->size

bool enqueue(PriorityQueue *q, Node *n)

Enqueues a node into the priority queue. Returns false if the priority queue is full prior to enqueuing the node and true otherwise to indicate the successful enqueuing of the node.

```
If pq_full return;
Items[size] = n
Size++
pq_heap_up()
Return true
```

bool dequeue(PriorityQueue *q, Node **n)

Dequeues a node from the priority queue, passing it back through the double pointer n. The node dequeued should have the highest priority over all the nodes in the priority queue. Returns false if the priority queue is empty prior to dequeuing a node and true otherwise to indicate the successful dequeuing of a node.

```
If pq_empty return false
*n = q->items[0]
Items[0] = last item
Heap_down
Return true
```

void pq_print(PriorityQueue *q)

A debug function to print a priority queue. This function will be significantly easier to implement if your en queue () function always ensures a total ordering over all nodes in the priority queue. Enqueuing nodes in an insertion-sort-like fashion will provide such an ordering. Implementing your priority queue as a heap, however, will only provide a partial ordering, and thus will require more work in printing to assure you that your priority queue functions as expected (you will be displaying a tree).

```
for (uint64_t i = (q->capacity - q->top); i < (q->capacity); ++i)
    if (q->items[i] != NULL)
            node_print((q->items[i]))
```

Codes

The third file is code.c:- After constructing a Huffman tree, you will need to maintain a stack of bits while traversing the tree in order to create a code for each symbol. I created a new ADT, a Code, that represents a stack of bits.The logic for setting a bit, clearing a bit, and getting a bit implemented here will be used later on for your bit vectors.

```
typedef struct {
        uint32_t top;
        uint8_t bits [ MAX_CODE_SIZE ];
    } Code ;
```

Code code_init(void)

You will simply create a new Code on the stack, setting top to 0, and zeroing out the array of bits, bits. The initialized Code is then returned.

uint32_t code_size(Code *c)

Returns the size of the Code, which is exactly the number of bits pushed onto the Code.

return c->top

bool code_empty(Code *c)

Returns true if the Code is empty and false otherwise.

return c->top == 0

bool code_full(Code *c)

Returns true if the Code is empty and false otherwise. The maximum length of a code in bits is 256, which we have defined using the macro ALPHABET. Why 256? Because there are exactly 256 ASCII characters (including the extended ASCII).

```
if (c->top == MAX_CODE_SIZE) {
return true;
}
return false;
}
```

bool code_set_bit(Code *c, uint32_t i)

Sets the bit at index i in the Code, setting it to 1. If i is out of range, return false. Otherwise, return true to indicate success.

```
if (i < ALPHABET) {
        c->bits[i / 8] = (c->bits[i / 8]) | (1 << (i % 8))
        return true
    }
    return false
}
```

bool code_clr_bit(Code *c, uint32_t i)

Clears the bit at index i in the Code, clearing it to 0. If i is out of range, return false. Otherwise, return true to indicate success.

```
if (i < ALPHABET) {
        c->bits[i / 8] = (c->bits[i / 8]) & ~(1 << (i % 8))
        return true
    }
    return false
}
```

bool code_get_bit(Code *c, uint32_t i)

Gets the bit at index i in the Code. If i is out of range, or if bit i is equal to 0, return false. Return true if and only if bit i is equal to 1.

```
if (((c->bits[i / 8] >> (i % 8)) & 1) == 1) {
        return true
    }
    return false
}
```

bool code_push_bit(Code *c, uint8_t bit)

Pushes a bit onto the Code. The value of the bit to push is given by bit. Returns false if the Code is full prior to pushing a bit and true otherwise to indicate the successful pushing of a bit.

```
if (code_full(c) == false)
        if (bit == 1)
            code_set_bit(c, c->top)
            ++c->top
            return true
        if (bit == 0)
            code_clr_bit(c, c->top)
            ++c->top
            return true
```

bool code_pop_bit(Code *c, uint8_t *bit)

Pops a bit off the Code. The value of the popped bit is passed back with the pointer bit. Returns false if the Code is empty prior to popping a bit and true otherwise to indicate the successful popping of a bit.

```
if (code_empty(c) == false) {
        if (code_get_bit(c, c->top - 1) == true) {
            --c->top
            *bit = 1
            code_clr_bit(c, c->top)
```

```
        return true
    }
    if (code_get_bit(c, c->top - 1) == false) {
        --c->top
        *bit = 0
        code_clr_bit(c, c->top);
        return true;
```

void code_print(Code *c)

A debug function to help you verify whether or not bits are pushed onto and popped off a Code correctly.

```
uint32_t cSize = code_size(c);
   for (uint32_t i = 0; i < cSize; ++i) {
      printf("%d  \n", c->bits[i]);
   }
}
```

I/O

The fourth file is I/O.c :- Low-level system(version) calls (syscalls) such as read(), write(), open() and close(). Functions defined by the following I/O module will be used by both the encoder and decoder. two extern variables defined in io.h: bytes_read and bytes_written. These are here for the purposes of collecting statistics.

int read_bytes(int infile, uint8_t *buf, int nbytes)

write a wrapper function to loop calls to read() until we have either read all the bytes that were specified (nbytes) into the byte buffer buf, or there are no more bytes to read. The number of bytes that were read from the input file descriptor, infile, is returned.

Total bytes read = 0
While total bytes read is less than nbytes
        Total += Syscall read function(infile, buf, nbytes-total)

Bytes_read += total
Return total

int write_bytes(int outfile, uint8_t *buf, int nbytes)

This function is very much the same as read_bytes(), except that it is for looping calls to write(). As you may imagine, write() is not guaranteed to write out all the specified bytes (nbytes), and so we must loop until we have either written out all the bytes specified from the byte buffer buf, or no bytes were written. The number of bytes written out to the output file descriptor, outfile, is returned.

Total bytes written = 0
While total bytes written is less than nbytes
    Total += Syscall write function(infile, buf, nbytes-total)

Bytes_written += total
Return total

bool read_bit(int infile, uint8_t *bit)

You will maintain a static buffer of bytes and an index into the buffer that tracks which bit to return through the pointer bit. The buffer will store the BLOCK number of bytes, where BLOCK is yet another macro defined in defines. h. This function returns false if there are no more bits that can be read and true if there are still bits to read. It may help to treat the buffer as a bit vector.

If bit_index == 0 read block bytes with read_bytes
    Bit = get_bit(buf, bit_index)
        Increment bit index and mod by block *8
If bit_index > end of buffer
    Return false else return true

void write_code(int outfile, Code *c)

The bits will be buffered starting from the 0 the bit in c. When the buffer of BLOCK bytes is filled with bits, write the contents of the buffer to outfile.

Loop through the code bit vector
If bit then set bit else clear bit
Increment bit_index
If buffer full write the bytes and reset bit index

void flush_codes(int outfile)

The sole purpose of this function is to write out any leftover, buffered bits. Make sure that any extra bits in the last byte are zeroed before flushing the codes.

```
 if (bitIndex != 0) {
       uint64_t nbytes = byte_length(bitIndex)
       write_bytes(outfile, buffer, nbytes)
       for (uint64_t i = 0; i < (bitIndex / 8) ++i) {
           buffer[i] = 0
       }
       bitIndex = 0
   }
}
```

Stacks

The fifth file is stack.c:- will need to use a stack in your decoder to reconstruct a Huffman tree. The interface of the stack should be familiar from assignment 4. The difference is that the stack this time around will store nodes.

```
struct Stack {
          uint32_t top;
          uint32_t capacity ;
          Node ** items ;
};
```

Stack *stack_create(uint32_t capacity)

The constructor for a stack. The maximum number of nodes the stack can hold is specified by capacity.

Malloc for stack
Set top to 0 and capacity to capacity, malloc capacity * size of Node
Return s
stack_delete() {
Free space in stack
}

void stack_delete(Stack **s)

The destructor for a stack. Remember to set the pointer to NULL after you free the memory allocated by the stack.

```
if (*s && (*s)->items) {
        for (uint32_t i = 0; i < (*s)->top; ++i) {
            node_delete(&(*s)->items[i])
        }
        free((*s)->items)
        free(*s)
        *s = NULL
    }
}
```

bool stack_empty(Stack *s)

Returns true if the stack is empty and false otherwise. bool stack_full(Stack *s) Returns true if the stack is full and false otherwise.

return if top is equal to 0

uint32_t stack_size(Stack *s)

Returns the number of nodes in the stack.

return value of top

bool stack_push(Stack *s, Node *n)

Pushes a node onto the stack. Returns false if the stack is full prior to pushing the node and true otherwise to indicate the successful pushing of a node.

bool stack_pop(Stack *s, Node **n)

Pops a node off the stack, passing it back through the double-pointer n. Returns false if the stack is empty prior to popping a node and true otherwise to indicate the success successful popping of a node.

void stack_print(Stack *s)

A debug function to print the contents of a stack.

```
for (uint64_t i = 0; i < (s->top); ++i)
        node_print(s->items[i])
```

A Huffman Coding Module

The sixth file is Huffman.c:- An interface for a Huffman coding module that you will need to implement will be given in Huffman. h.

Node *build_tree(uint64_t hist[static ALPHABET])

Constructs a Huffman tree is given a computed histogram. The histogram will have ALPHABET indices, one index for each possible symbol. Returns the root node of the constructed tree. The use of static array indices in parameter declarations is a C99 addition. In this case, it informs the compiler that the histogram hist should have at least an ALPHABET number of indices.

```
    Node *n;
    Node *left;
    Node *right;
```

```
Node *joined = NULL
PriorityQueue *pq = pq_create(ALPHABET)
```

Node n, left, right, joined
Priority queue PQ
Loop through the alphabet and if hist of alphabet is not 0 create a node and set frequency to hist[i]
While the size of pq > 1
Dequeue left and right and the enqueue the joined ned
Dequeue the root node
Delete pq
Return node

void build_codes(Node *root, Code table[static ALPHABET])

Populates a code table, building the code for each symbol in the Huffman tree. The constructed codes are copied to the code table, table, which has ALPHABET indices, one index for each possible symbol.

```
if(root) {
        If we are at a leaf node
                Set table[root->symbol] to c
Else
        Push 0 onto code
        Recursive call with left
        Pop bit

Push 1 onto code
Build code with right node
Code pop bit
```

void dump_tree(int outfile, Node *root)

Conducts a post-order traversal of the Huffman tree rooted at root, writing it to outfile. This should write an ‘L’ followed by the byte of the symbol for each leaf and an ‘I’ for interior nodes. You should not write a symbol for an interior node.

```
if (root) {
      dump_tree(outfile, root->left)
```

```
    dump_tree(outfile, root->right)
}
if (!(root->left) && !(root->right)) {
    write_bytes(outfile, 0, 'L')
    write_bytes(outfile, 0, root->symbol)
}
    write_bytes(outfile, 0, 'I')
}
```

Node *rebuild_tree(uint16_t nbytes, uint8_t tree_dump[static nbytes])

Reconstructs a Huffman tree given its post-order tree dump stored in the array tree_dump. The length in bytes of tree_dump is given by nbytes. Returns the root node of the reconstructed tree.

Node for right left and joined
Stack of size alphabet
Loop through nbytes
If it's a leaf node
    Increment the counter and create node with tree dump of i
    Push onto stack
Else
    Pop right and left and push the joined ned
Return the root node (Last node in the stack)

void delete_tree(Node **root)

The destructor for a Huffman tree. This will require a post-order traversal of the tree to free all the nodes. Remember to set the pointer to NULL after you are finished freeing all the allocated memory

```
if (*root)
    delete_tree(&(*root)->left)
    delete_tree(&(*root)->right)
    node_delete(root)

  return
```

The Encoder

```
post_order_travers(Node *root, uint8_t *buf, uint32_t i) {
        If root recursively call with left and right
                If leaf node
                        Buf[i++] = ‘L’ buf[i++] = root->symbol
                Else:
                        Arr[i++] = ‘I’

Int main() {
        Int br; bytes_read
        Header h
        Statbuf
        Uint8_t treemdump[MAX_TREE_SIZE]
        Uint8_t buf[BLOCK]
        Unique_symbols = 0
        Dump_index = 0
        Infile = 0 stdin
        Outfie = 1 stdout
        Code table[ALPHABET]
        Tempfiledesc
        Uint64_t hist

        Increment histogram of 0 and 255
        Switch case for command-line arguments using get opt
        If the file is not seekable open a temp file and read to the temp file
        Fstat the infile and save to statbuf
        Change permission for outfile to match

While (br = read_bytes(infile, buf, BLOCK) > 0) {
Hist[buf][i]++;

}

Loop through the alphabet
If hist[i] > 0 increment unique symbols
```

Root = build_tree(hist)
build_codes(root, table)

H.magic = MAGIC
H.permissions = instatbuf.st_mode
H.tree_size = 3 * unique_symbols -1
H.file_size = instatbuf.st_size
Write the header using write bytes. Cast the header to a uint8
post_order_traverse(root,dump &dump_index

Lseek the file back to 0
while ((br = read_bytes(infile, buf, BLOCK)) > 0)
For each byte (iterate to br)
write_code(outfile, &table[buf[i]);

Flush_codes

If verbose print the stats

Unlink the tempfile if we made one
Free up any space used

The Decoder

Bw - 0
Header h
Node root_node
Node node

Struct stat Instabug
Uint8_t buf [BLOCK]
Uint8_t Bit
Infile = 0
Outfile = 1
Opt = 0
Verbose = false
Switch case to handle the command line arguments
Read_bytes(infile, (uint8_t *) &h, sizeof(Header));

If magic does not match print error and exit

Get stats of the infile
fchmod(outfile, h.permissions)
Dump[h.tree_size];
read_bytes(infile, dup, h.tree_size);
Node = root_node
While (bw <h.file_size && read_bit(infile &bit) {
    If bit is 1 node = right. else node = left
        If leaf node {
            Buf[buf_index++] = node->symbol
            Bw++
            Node = root_node

            If the buffer is full write_buffer to file and reset buf_index

write_bytes(outfile, buf, buf_index)

If (verbose) {
    Print the stats
}
Free up any extra space so there are no memory leaks

Design process:-

Most of the files were simple to do using the explanation from section help and TA pseudo-code except io.c, code.c, encode and decode. Io.c was really long and hard to code cause a lot of things I didn't really know how to go about and required a lot of debugging and when going through the pipeline it just showed checked so I couldn't tell if there is an error unless it was syntax error but after going to many sections I think I did it correctly. My code.c push and pop were not working cause I had errors in my set and ctrl which were quite small but was hard to notice until later. And encode I am still struggling in it and decode only got half of it to work on the pipeline