Huffman’s Coding’

Summary: Huffman’s coding is an algorithm where you can compress files to a smaller size. We will be creating our own version to encode and decode our own files with this algorithm with the help of trees, priority queues, stacks, and code tables.

Node.c:

StructNode:

- Left: Left children of the node
- Right: right child of the node
- Symbol: symbol
- Frequency: number of times it exists i guess

Node *node_create(uint8_t symbol, uint64_t frequency):

- Constructor to create a node structure

void node_delete(Node **n):

- Frees the node and sets the pointer to NULL

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

- Creates a parent with the two nodes as the children
- The symbol will be ‘$’ and the frequency is the sum of the frequency form the two nodes

void node_print(Node *n);

- Prints the symbol and frequency of the node

bool node_cmp(Node *n, Node *m);

- Checks if the frequency of node n is greater than the frequency of node m

void node_print_sym(Node *n);

- Only prints the symbol of the node

PriorityQueue.c:

Struct priorityQueue:

- Tail: uint64_t that indicates the last value of the queue also indicates the size
- Capacity: indicates the size of queue
- Queue: Contains a list of nodes

PriorityQueue *pq_create(uint32_t capacity);

- Initialize and allocate memory for my priority queue

void pq_delete(PriorityQueue **q);

- Delete the allocated memory

bool pq_empty(PriorityQueue *q);

- Check if the priority queue is empty

bool pq_full(PriorityQueue *q);

- Check if the priority queue is full

uint32_t pq_size(PriorityQueue *q);

- Returns how many values are in the queue

bool enqueue(PriorityQueue *q, Node *n);

- Adds a node into the priority queue.
- Sort the queue
- Return true to indicate a successful addition and false if it failed

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

- Return the node with the highest priority and store it in n.
- Sort the queue
- Return true indicating success and false if the removal failed.

void pq_print(PriorityQueue *q);

- Prints the priority q

Code.c

CodeStruct:

- Top of the “code” stack
- Uint8_t bits[MAX_code_SIZE]: fixed array of 32 values(256/8)

uint32_t code_size(Code *c);

- Return the amount of bits pushed onto the CODE
- Use Top to track it

bool code_empty(Code *c);

- Check if Code is empty by seeing if top is 0

bool code_full(Code *c);

- Return true if code is full

bool code_set_bit(Code *c, uint32_t i);

- Using bit shifts to set the bit at index i
- Return true to indicate success and false if “i” is out of range

bool code_clr_bit(Code *c, uint32_t i);

- Clear the bit at index i using bit shifts and bit manipulation
- Return true to indicate success and false if “i” is out range

bool code_get_bit(Code *c, uint32_t i);

- Get the bit at index i
- Return true if the bit is 1 and false if bit is 0

bool code_push_bit(Code *c, uint8_t bit);

- Pushes a bit into Code indicated by the value of the input
- Return true to indicate success and false if Code is full

bool code_pop_bit(Code *c, uint8_t *bit);

- Pops a bit from Code c and puts it in uin8_t * bit
- Returns true if successful and false if Code is empty prior to popping

void code_print(Code *c);

- Print code

Io.c:

Extern vars:

- Bytes_read: count how bytes were read
- Bytes_written: count how many bytes were written

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

- Keep reading blocks of bytes using the read() function until there is no more to read or you have read enough(indicated by nbytes) into *buf.

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

- Keep writing blocks of bytes until there is no more to write or you have written enough into the file.

bool read_bit(int infile, uint8_t *bit);

- Take in a block of bytes and read the bits on at the time
- Return true if there is more bits read and false otherwise

void write_code(int outfile, Code *c);

- Buffer each bit in C into the buffer.
- Then write the contents to the outfile

void flush_codes(int outfile);

- Zero the leftover bits and flush them away

Stack.c:

StructStack:

- Top: top of the stack
- Capacity: Size of the stack
- **items: List of the nodes in the stack(basically is the stack)

Stack *stack_create(uint32_t capacity):

- Create the stack and initialize each variable in the stack.

void stack_delete(Stack **s);

- Delete each node in the stack and free the stack.

bool stack_empty(Stack *s);
- Check if the stack is empty

bool stack_full(Stack *s);
- Check if the stack is full

uint32_t stack_size(Stack *s);
- Returns the size of the stack

bool stack_push(Stack *s, Node *n);
- Add the inputted node to the top of the stack

bool stack_pop(Stack *s, Node **n);
- Take the top of the top of the stack and store it in n.
- Free that node

void stack_print(Stack *s);
- Prints the whole stack

Huffman.c

Node *build_tree(uint64_t hist[static ALPHABET]);
- Construct a huffman tree based on the inputted histogram
- Create a priority queue
- Put every node in the priority queue
- Loop until there is only one node left in the priority queue
- Dequeue two nodes and join them with a node with symbol $ and frequency of the sum of the two nodes
- Put that node in the priority queue
- Return the root of the tree

void build_codes(Node *root, Code table[static ALPHABET]);
- Create a code table which will be the code for each symbol in the created tree
- Use a backtracking algorithm.
- Recursively call the nodes that go left and give them a 0
- Recursively call the nodes that go right and give them a 1
- Once you reach the leaf, store that in the code table

void dump_tree(int outfile, Node *root);

- Do a post order traversal or dfs on the tree and write it to the file. Write “L” after every leaf symbol and “I” after every interior nodes, but no symbol for interior nodes.

Node *rebuild_tree(uint16_t nbytes, uint8_t tree[static nbytes]);
- Rebuild the tree from the post-order tree dump stored in the imputed array.
- Create a stack
- Iterate through the tree array
- If the value is a leaf node, push it in a stack
- If the value is interior, pop the next two values and set those two nodes as the current’s node’s children. Push that node into the stack
- Repeat until there is only one node in the stack
- Return that node(root)

void delete_tree(Node **root);
- Free all nodes and set the pointer for the tree to NULL
- Recursively do this

Encoder.c

- Functionality: Reads an input and gets the huffman encoding in order to compress the file
- Options:
    - H: print the help message
    - -i : set the input file that will be encoded(default = stdin)
    - -o : set the output file that will take in the encoded/compressed file(default = stdout)
    - -v : Prints the compression statistics to stderr. Stats are uncompressed file size, compressed file size, and saved space.
        - Equation for space saved: 100 * (1-(compressed size/uncompressed size))
- How to encode the file:
    - Count the number of occurrences of each unique symbol in the file
    - Construct the huffman tree using the histogram
    - Construct a code table to represent a symbol and the value at the symbol’s code
    - Emit an encoding of the huffman tree to the file
    - Construct a header
        - Set magic to Magic
        - Set permissions and file_size with the help of fstat
        - Set file_size to 3* unique symbols -1
    - Write the header to outfile
    - Write the dumped tree into outfile
    - Write the code to the outfile with the help of write code
    - Call flush_codes to write if there is still codes in the buffer

Decoder.c:

- Functionality: Read in a compressed input file and decompress it.
- Options:
    - -h: prints the help statement
    - -i: set the input file that will be decoded(default = stdin)
    - -o: set the output file that will receive the decoded file(default = stdout)
    - -v: prints the decompression stats to stderr, stats include compressed file size, decompressed file size, and space saved.
- How to decode file
    - Read the header from the input file
        - Check if the magic number is valid
    - Set the permission to the outfile using the imputed header
    - Read the dump tree into an array, then use that array to rebuild the tree
    - Read the rest of the input file bit by bit. Traverse down the huffman tree until you reach a leaf node, then you traverse again from the root.
    - This will help us rebuild the tree
    - Stop until the amount of symbols reaches the original file size

Makefile:

- make : build encoder and decoder
- Make encode: only builds encoder
- Make decode: builds just the decoder
- Make clean: removes all files that are generated from the compiler
- Make spotless: same as make clean but also removes executables
- Make format: formats all the .c files NOT THE HEADER FILES