Huffman Coding - CSE13S Assignment 5

Nicholas Reis

October 26, 2021

Program Description

This program will implement multiple abstract data types and functions that provide functionality to encode and decode files. The encoder can be used to compress a file into less memory than it normally would use, and the decoder can be used to decipher an encoded file back into its original contents. The encoder will go through the process of: reading through infile to make a histogram of ASCII characters, making a Huffman tree using a priority queue ADT I create, making a code table by traversing through the Huffman tree, putting the information together into a header, writing the header and the Huffman tree to outfile, writing the corresponding code for each symbol to outfile, and then flushing the remaining buffered codes and closing the files. The decoder will go through the process of: reading in the header from infile, reading the dumped tree into an array and reconstructing it, reading infile one bit at a time and writing the each leaf node's symbol to outfile until the number of decoded symbols matches the original file size.

Encode

My encode.c file takes in an input file and compresses it so that it takes up less space.

Pseudocode

```
include all necessary files
define OPTIONS "hvi:o:"

void print_help():
    print out the usage/help section
    exit the program

void print_stats():
    prints out the uncompressed and compressed file sizes, along with percent space saved

main:
    set the default input and output to be stdin/stdout
    create booleans for help and stats

    while loop that uses getopt to access the options:
    switch:
    case 'h' '?' and default:
        set help boolean to true
    case 'v':
        set stats boolean to true
    case 'i':
```

```
    open the given input file
    use fstat() to access permissions
cse 'o':
    open the given output file
    use fchmod() to copy file permissions from infile

if (help):
    print_help()

create a histogram of size ALPHABET (256)
create a buffer of size BLOCK (4096)

read in bytes from infile while there are bytes to be read:
    increment values in the histogram if a specific byte is read

increment element 0 and 255 in the histogram (minimum two elements needed)

call build_tree() with the histogram and get the root node

create an array of Codes
call build_codes() with the root and Codes array

use a for loop to count the number of unique symbols in the histogram

define a Header
.magic = MAGIC
.permissions = infile permissions
.tree_size = 3 * unique symbols - 1
.file_size = size of infile

write out the header using write_bytes()
write out the tree using dump_tree()

lseek() to beginning of infile
while there are bytes to read:
    read_bytes() from infile
    write the code for each read byte using write_code()
flush_codes()

delete_tree() starting from the root node

if (stats):
    print_stats()

close(infile)
close(outfile)
```

---

1. The print_help() and print_stats() function are used in case the user specifies these OPTIONS. exit(0) is used to end the program after print_help is called.
2. fstat() and fchmod() are used in order to store the file permissions from infile and copy them to the outfile.
3. To keep track of the frequencies of each ASCII character in the input file, a histogram is used. We also

use this histogram to build the Huffman tree that will create nodes for each byte that has a frequency greater than 0.

4. It then call build_codes() with an array of Codes to store Codes for each leaf in the Huffman tree that will be used in the decoder.
5. The header must then be created and written out as the first thing in the output file. It contains a member magic that indentifies a file as one that has been compressed by my encoder, permissions that store the input file permissions, tree_size which contains the Huffman tree size in bytes, and file_size which is the original input file size.
6. The Huffman tree is then dumped to the output file. Then it reads through the input file again and writes each byte's code to outfile using the previously created array of Codes. flush_codes() is used to ensure everything is flushed out and written to outfile.
7. It finally needs to deallocate the tree, print the stats if the option was enabled, and close both files.

Decode

My decode.c file takes in a compressed file and decompresses it back into the original file.

Pseudocode

```
include all necessary files
define OPTIONS "hvi:o:"

static buffer
static decoded_symbols = 0
void traverse(infile, outfile, current_node, root_node, file_size):
    if the current_node and root_node exist:
        while decoded_symbols < file_size:
            use read_bit() to get the next bit
            if the bit is 0:
                set the current_node = current_node's left child
            else:
                set the current_node = current_node's right child

            if a leaf node is reached:
                buffer the symbol of current_node
                increment decoded_symbols
                go back to the root_node

            if the buffer is full:
                write out the buffer using write_bytes()
            else if the decoded_symbols == file_size:
                write out the rest of the buffer

void print_help():
    print out the usage/help section
    exit the program

void print_stats():
    prints out the compressed and decompressed file sizes, along with percent space saved

main:
```

```
set the default input and output to be stdin/stdout
create booleans for help and stats

while loop that uses getopt to access the options:
switch:
case 'h' '?' and default:
    set help boolean to true
case 'v':
    set stats boolean to true
case 'i':
    open the given input file
cse 'o':
    open the given output file

if (help):
    print_help()

use read_bytes() to read in the header from infile

if the magic member in the header != MAGIC:
    print out an error message
    exit the program

copy the file permissions from the header to the outfile

use read_bytes() to read in the dumped tree from infile

use rebuild_tree() to reconstruct the tree starting at the root node

use traverse() with the codes from infile to decompress the file

delete_tree() starting from the root node

if (stats):
    print_stats()

close(infile)
close(outfile)
```

---

1. traverse() is a function I made that goes down either side of a Huffman tree and buffers in the ASCII characters in the order that the original file had them, and writes out the buffer once it is filled or the entire file has been decompressed.
2. The print_help() and print_stats() function are used in case the user specifies these OPTIONS. exit(0) is used to end the program after print_help is called.
3. It first needs to read in a given header and copy the information into a header that it created beforehand. If the magic number of the header is invalid, an error message is printed and the program is exited.
4. It then reads in the tree dump and reconstructs the tree starting from the root node by calling rebuild_tree().
5. Starting from the root node, it calls the traverse() function, which will decompress the file.
6. It finally needs to deallocate the tree, print the stats if the option was enabled, and close both files.

Node

Within node.c, I implemented code to create an ADT called Node. It provides functions that allow me to create a Huffman tree using nodes. These functions include: creating a node, deleting a node, joining two nodes together into one node, and a debug function for the nodes.

Pseudocode

```
include node.h, stdlib.h

Node *node_create(symbol, frequency):
    use malloc to dynamically allocate memory for a Node *n
    set node n's symbol to symbol
    set node n's frequency to frequency
    set node n's left to NULL
    set node n's right to NULL
    return the node

void node_delete(Node **n):
    free() the memory allocated to the node
    set the pointer to the node to NULL
    return

Node *node_join(left, right):
    add the two frequencies together into a temp variable
    create a parent node with the '$' symbol and temp frequency
    set the parent node's left to left
    set the parent node's right to right
    return the parent node

void node_print(node *n):
    debug function that is used to check that nodes work properly
```

---

1. Node *node_create makes a node with dynamically allocated memory, a symbol that is set to symbol, and a frequency that is set to frequency before returning the node.
2. void node_delete frees the memory allocated for the node, and then dereferences the pointer to NULL and returns.
3. Node *node_join joins two nodes together into one parent node that has the symbol '$' and a frequency that equals the sum of the left and right nodes' frequncies. Its left child is set to left and its right child is set to right, and then the parent node is returned.

Priority Queue

Within pq.c, I implemented code to create an ADT called PriorityQueue. It provides functions that allow me to enqueue Nodes onto a PriorityQueue that will then order them into a min heap so that the first Node will always have the least frequency. These functions include: creating a priority queue, deleting a priority queue, checking if it is full or empty, returning the size, enqueuing and dequeuing, and a debug function. I also had to implement functions that would make a min heap, so I brought in some code from assignment 3 and adjusted it to have it fit this program.

Pseudocode

```
include pq.h, node.h, math.h, stdlib.h

int cmp(Node x, Node y):
    if the frequency of y is less than x:
        return 1
    else if the frequency of y is greater than x:
        return -1
    else:
        return 0

void swap(Node *x, Node *y):
    set a temp Node to *x
    set *x equal to *y
    set *y to temp Node

uint32_t min_child(Node **items, first, last):
    set left equal to 2 * first
    set right equal to left + 1
    if (right is less than or equal to last) and (right frequency is less than left frequency):
        return right
    else:
        return left

void fix_heap(Node **items, first, last):
    boolean found = false
    mother = first
    small = min_child(items, mother, last)

    while (mother is less than or equal to the floor of (last / 2.0)) and not found:
        if (smallest child is less than mother):
            swap them
            mother = small
            small = min_child(items, mother, last)
        else:
            found = true

void build_heap(Node **items, first, last):
    for (father = floor of (last / 2.0)) while father is greater than or equal to first - 1, decrement
        fix_heap(items, father, last)

struct PriorityQueue:
    int top
    int capacity
    Node **items

PriorityQueue *pq_create(capacity):
    use malloc to dynamically allocate memory for a PriorityQueue *q
    if q:
        set the top of q to 0
        set the q's capacity to capacity
        use calloc to dynamically allocate memory for nodes in q's items array
        if there are no items:
```

```
            free() q
            set q to NULL
    return the q

void pq_delete(PriorityQueue **q):
    if the PriorityQueue exists and has items:
        free() the memory allocated to the items
        free() the memory allocated to the PriorityQueue
        set the pointer to the PriorityQueue to NULL
    return

bool pq_empty(PriorityQueue *q):
    return (top equals 0?)

bool pq_full(PriorityQueue *q):
    return (top equals q's capacity?)

uint32_t pq_size(PriorityQueue *q):
    return the top of the PriorityQueue

bool enqueue(PriorityQueue *q, Node *n):
    if q is full:
        return false
    set the value in the items array at [top] to n
    increment the top
    build_heap()
    return true

bool dequeue(PriorityQueue *q, Node **n):
    if q is empty:
        return false
    dereference n to change the value it points to as the first item
    swap the first and last items
    decrement the top
    build_heap()
    return true

void pq_print(PriorityQueue *q):
    debug function that is used to check that PriorityQueues work properly
```

---

1. cmp() compares two nodes and is primarily used to determine if one node's frequency is smaller than another.
2. swap() swaps the location of two nodes in memory.
3. min_child() is used to return the child node with the smallest frequency.
4. fix_heap() is used to move nodes with smaller frequencies up to the top of the min heap. It does this by doing comparisons between children and parent nodes and then swaps them, before then repeating the process until a node is in the right spot in the heap.
5. build_heap() runs fix_heap until every node in the heap is sorted into the correct location based on frequencies (lower freq is at the top with higher priority).
6. The PriorityQueue struct sets each PriorityQueue to have variables top and capacity along with an

array of nodes.

7. pq_create() makes a PriorityQueue with dynamically allocated memory, a top that is set to 0, a specified capacity, an array of nodes with dynamically allocated memory, frees the PriorityQueue if there are no nodes, and returns the PriorityQueue.
8. pq_delete frees the memory allocated for the nodes array and the PriorityQueue, and then dereferences the pointer to NULL and returns.
9. pq_empty returns true if the top of the PriorityQueue is at 0 and false otherwise.
10. pq_full returns true if the top of the PriorityQueue is at capacity and false otherwise.
11. pq_size returns the top of the PriorityQueue (which indicates how many items are in it).
12. enqueue returns false if the stack is full, otherwise sets the item at the top of the PriorityQueue to the input node before incrementing the top, building the heap, and returning true.
13. dequeue returns false if the stack is empty, otherwise dereferences n to change the value it points to as the first node in the array (which is the node with the smallest frequency) and then swaps the first and the last node in the array before decrementing the top. Build the heap and then return true.
14. void pq_print is a debug function that prints out the PriorityQueue.

Code

Within code.c, I implemented code to create an ADT called Code. It provides functions that allow me to use a code that stores specific bits that will be used to create a code for each symbol in my encoder. These functions include: creating a code, returning the code size, checking if the code is full or empty, setting the value of a bit to 1, clearing the value of a bit to 0, getting the value of a bit, pushing a bit, popping a bit, and a debug function.

Pseudocode

```
include code.h, defines.h

Code code_init():
    create a Code c
    set the top of c to 0
    for loop iterates through every bit
        set each bit to 0
    return c

uint32_t code_size(Code *c):
    return the top of c

bool code_empty(Code *c):
    if the top of c is 0:
        return true
    else:
        return false

bool code_full(Code *c):
    if the top of c is equal to the ALPHABET:
        return true
    else:
        return false
```

```
bool code_set_bit(Code *c, i):
    if i is out of range:
        return false
    or equals the bit at index (i/8) with 0x1 left shifted (i%8)
    return true

bool code_clr_bit(Code *c, i):
    if i is out of range:
        return false
    and equals the bit at index (i/8) with not(0x1 left shifted (i%8))
    return true

bool code_get_bit(Code *c, i):
    if i is out of range or ((bit at index (i/8) right shifted (i%8)) and 0x1) equals 0x0:
        return false
    else:
        return true

bool code_push_bit(Code *c, bit):
    if the code is full:
        return false
    if bit == 0:
        clear the bit at the top
    else if bit == 1:
        set the bit at the top
    increment the top
    return true

bool code_pop_bit(Code *c, *bit):
    if the code is empty:
        return false
    decrement the top
    if code_get_bit is true:
        *bit = 1
    else:
        *bit = 0
    return true

void code_print(Code *C)
    debug function that is used to check that Codes work properly
```

---

1. Code code_init creates a Code, sets the top to 0, and then sets every bit in the Code to 0 before returning the Code.
2. uint32_t code_size returns the top of the Code, which is the equivalent to the size.
3. bool code_empty returns true if the top is 0, meaning there are no set bits, otherwise it returns false.
4. bool code_full returns true if the top is equal to the MAX_CODE_SIZE, otherwise it returns false.
5. bool code_set_bit returns false if i is out of range, otherwise performs an or equals with the bit at index i / 8 and 0x1 left shifted i % 8 times before returning true.
6. bool code_clr_bit returns false if i is out of range, otherwise performs an and equals with the bit at

index i / 8 and not 0x1 left shifted i % 8 times before returning true.

7. bool code_get_bit returns false if i is out of range or if the bit at i / 8 right shifted i % 8 times and 0x1 does not equal 0x1 (meaning the bit at that location is 0), and returns true otherwise.
8. bool code_push_bit returns false if the code is full. If the bit argument is 0, clear the bit at the top. If the bit argument is 1, set the bit at the top. Increment the top and then return true.
9. bool code_pop_bit returns false if the code is empty. Decrement the top. If code_get_bit is true (meaning the bit is 1), set *bit to 1. Else set *bit to 0 and then return true.
10. void code_print is a debug function that prints out the code.

I/O

Within io.c, I implemented code that will read from a file and write to another file. It provides functions that allow me to read and write bytes, along with reading and writing bits, and then a function to flush out the bits that do not get written at the end.

Pseudocode

```
include io.h, code.h, defines.h, unistd.h, stdbool.h

static bytes_read = 0
static bytes_written = 0
static write_buffer
static write_index = 0
static read_buffer
static read_index = 0
static read_end = -1

int read_bytes(infile, *buf, nbytes):
    if nbytes <= 0:
        return 0
    do:
        bytes = read() from the infile, buf + bytes, nbytes - bytes
        bytes_read += bytes
    while bytes are greater than 0
    return bytes

int write_bytes(outfile, *buf, nbytes):
    if nbytes <= 0:
        return 0
    do:
        bytes = write() to the outfile, buf + bytes, nbytes - bytes
        bytes_written += bytes
    while bytes are greater than 0
    return bytes

bool read_bit(infile, *bit):
    if read_index is 0:
        bytes = call read_bytes() to read in a BLOCK of bytes
        if bytes < BLOCK:
            set the read_end to bytes * 8 + 1
    if the bit at the read_index is 0x1:
```

```
        *bit = 1
    else:
        *bit = 0
    increment read_index
    if read_index == BLOCK * 8:
        set the read_index to 0
    return true if the read_index does not equal read_end

void write_code(outfile, Code *c):
    for loop iterate from 0 to code_size(c):
        get the bit at current index
        if the bit is 1:
            set the bit in write_buffer
        else:
            clear the bit in write_buffer
        increment write_index
        if write_index / 8 == BLOCK:
            write out the buffer to outfile using write_bytes()
            set the write_index to 0

void flush_codes(outfile):
    if write_index > 0:
        bytes_needed = write_index / 8
        if the write_index % 8 is not 0:
            until write_index % 8 == 0:
                clear out extra bits
                increment write_index
            increment bytes_needed
        write_bytes() the number of bytes_needed
```

---

1. A lot of static variables are needed for this file because the variables are shared across the different functions. For example, read_bit, write_code, and flush_codes all depend on variables that are altered in read_bytes and write_bytes (such as read_buffer and write_buffer).
2. read_bytes needs to loop calls to read() until every byte is read from an input file, which means until read() returns 0, the function will read in more bytes before returning the amount of bytes that were read.
3. write_bytes also needs to loop calls to write() until every byte is written to an output file, which means until write() returns 0, the function will write more bytes before returning the amount of bytes that were written.
4. read_bit reads in an entire buffer of bytes on its first call, and from that point forward gets one bit each time it is called from that buffer. Once the buffer is filled, it reads in another entire buffer of bytes.
5. write_codes buffers each bit from a given Code into a buffer and writes out the buffer once it is full.
6. flush_codes will ensure that every byte is written out, as write_codes only writes out the buffer if it is full and there could be left over bytes in the buffer.

Stack

Within stack.c, I implemented code to create an ADT called Stack. It provides functions that allow me to use a stack to deal with adding and removing vertices from the traveled path. These functions include: creating

a stack, deleting a stack, checking if a stack is full or empty, returning the number of items in the stack, pushing a vertex, popping a vertex, and printing the contents of the stack to an output file.

Pseudocode

```
include stack.h, stdbool.h, stdint.h, stdio.h, stdlib.h

struct Stack:
    int top
    int capacity
    Node items

Stack *stack_create(capacity):
    use malloc to dynamically allocate memory for a Stack *s
    if s:
        set the top of the stack to 0
        set the stack's capacity to capacity
        use calloc to dynamically allocate memory for nodes in the stack
        if there are no items:
            free() the stack
            set the stack to NULL
    return the stack

void stack_delete(Stack **s):
    if the stack exists and has items:
        free() the memory allocated to the items
        free() the memory allocated to the stack
        set the pointer to the stack to NULL
    return

bool stack_empty(Stack *s):
    if the top of the stack is 0:
        return true
    else:
        return false

bool stack_full(Stack *s):
    if the top of the stack is at capacity:
        return true
    else:
        return false

uint32_t stack_size(Stack *s):
    return the top of the stack

bool stack_push(Stack *s, Node *n):
    if the stack is full:
        return false
    set the value in the items array at [top] to n
    increment the top
    return true

bool stack_pop(Stack *s, Node **n):
    if the stack is empty:
```

```
        return false
    decrement the top
    dereference n to change the value it points to as the popped item
    return true

void stack_print(Stack *s):
    debug function that is used to check that Stack work properly
```

---

1. struct Stack defines the structure we will be using in the rest of these functions. It creates uint32_t variables top, capacity, and an array of Node items.
2. Stack *stack_create makes a stack with dynamically allocated memory, a top that is set to 0, a specified capacity, an array of nodes with dynamically allocated memory, frees the stack if there are no nodes, and returns the stack.
3. void stack_delete frees the memory allocated for the stack nodes and the stack, and then dereferences the pointer to NULL and returns.
4. bool stack_empty returns true if the top of the stack is at 0 and false otherwise.
5. bool stack_full returns true if the top of the stack is at capacity and false otherwise.
6. bool stack_size returns the top of the stack (which indicates how many items are in it).
7. bool stack_push returns false if the stack is full, otherwise increments the top of the stack to move on to the next index of the array and sets the value of the stack's items[top] to n. Returns true.
8. bool stack_pop returns false if the stack is empty, otherwise dereferences n to change the value it points to as the stack's items[top] and then decrements the top of the stack to move on to the previous index of the array. Returns true.
9. void stack_print is a debug function that prints out the stack.

Huffman

Within huffman.c, I implemented functions that are utilized in both encode.c and decode.c. There are five functions, which include: a function that creates a huffman tree, dumps a tree, rebuilds a tree, deletes a tree, and builds codes for symbols in a tree.

Pseudocode

```
include huffman.h, io.h, pq.h, defines.h, stack.h, unistd.h, stdio.h

Node *build_tree(hist[ALPHABET]):
    create a PriorityQueue
    for i = 0, i < ALPHABET:
        if hist[i] > 0:
            create a node for the hist element
            enqueue the node into the PriorityQueue
    while the PriorirtyQueue size > 1:
        dequeue left
        dequeue right
        create a parent node using node_join(left, right)
        enqueue parent
    dequeue root
    delete the PriorityQueue
```

```
    return root

void build_codes(Node *root, Code table[ALPHABET]):
    code_init a Code
    if the root exists:
        if a root is a leaf node:
            table[root symbol] = code
        else:
            push bit 0
            recursive call left child
            pop bit

            push bit 1
            recursive call right child
            pop bit

void dump_tree(outfile Node *root):
    if root exists:
        recursive call left child
        recursive call right child

        if root is a leaf node:
            buffer an 'L'
            buffer the root symbol
        else:
            buffer an 'I'
    if buffer has tree_size elements:
        write_bytes() the buffer

Node *rebuild_tree(nbytes, tree_dump[nbytes]):
    create a Stack
    for i = 0, i < nbytes:
        if tree_dump[i] == 'L':
            increment i
            create a leaf node for tree_dump[i]
            push the node to the stack
        else:
            pop right node
            pop left node
            create a parent node using node_join(left, right)
            push parent node
    pop root node
    delete the Stack
    return root

void delete_tree(Node **root):
    if *root exists:
        recursive call left child
        recursive call right child
        node_delete(root)
```

---

1. build_tree uses a PriorityQueue to manage the Huffman tree. For any elements greater than 0 in the histogram, they must be added to the queue because it means that they have shown up at least once

in the input file that it is encoding. Once all nodes are added, the function dequeues two nodes and creates a parent node with them until there is only one node left in the queue. This node is dequeued and returned as the root node of the tree. Before returning, the PriorityQueue must be deleted, as it is no longer needed.

2. build_codes takes in the root node of a Huffman tree along with an empty array of Codes. The function then performs post-order traversal from the root of the tree in order to give each leaf node a code that is then put into the array.

3. dump_tree also performs a post-order traversal, but instead uses a buffer to store 'L', 'I', and node symbols in a specific order before writing the dumped tree to outfile once the post-order traversal is complete.

4. rebuild_tree uses a Stack to manage the tree dump. If the element at tree_dump[i] is an 'L', the following element is a node symbol which will then be pushed to the stack. If the element is an 'I', two nodes are popped from the stack and they are joined into a parent node and pushed back onto the stack. Once the iteration is complete, the final thing popped off the stack is the root node which is returned after the stack is deleted.

5. delete_tree frees nodes one by one starting from the bottom of the tree, so as to not cause any issues deleting nodes at the top and losing access to the bottom nodes.
