Assignment 6 DESIGN.pdf

Victor Nguyen

March 3, 2022

1 Description of Programs:

In this assignment we are tasked to create Huffman encoding/decoding. Purpose of the Huffman algorithm is to compress and decompress files in a way where we don't lose any data.

2 Files to be included in directory "asgn6":

1. encode.c
   - Source file that contains the implementation of Huffman encoding.
2. decode.c
   - Source file that contains the implementation of Huffman decoding.
3. defines.h
   - Header file thats used to link with other files.
   - Contains helpful defines to use for the encoder/decoder.
4. header.h
   - Header file thats used to link with other files.
   - Contains the struct definitions for a header.
5. node.h
   - Header file thats used to link with other files.
   - Contains the node ADT interface.
6. node.c
   - Source file that contains the implementations of the node ADT.
7. pq.h
   - Header file thats used to link with other files.
   - Contains the priority queue ADT interface.

8. pq.c
    - Source file that contains the implementations of the priority queue ADT.
9. code.h
    - Header file thats used to link with other files.
    - Contains the code ADT.
10. code.c
    - Source file that contains the implementations of the code ADT.
11. io.h
    - Header file thats used to link with other files.
    - Contains the I/O interface.
12. io.c
    - Contains the implementations of the I/O module.
13. stack.h
    - Header file thats used to link with other files.
    - Contains the stack ADT interface.
14. stack.c
    - Contains the implementations of the stack ADT.
15. huffman.h
    - Header file thats used to link with other files.
    - Contains the Huffman coding interface.
16. huffman.c
    - This file contains the implementations of the Huffman coding module.
17. Makefile
    - File that formats the program into clang-format.
    - File also compiles the above .c programs.
    - Cleans files that were generated during the compilation process.
    - File also contains scan-build and valgrind to check for any errors/memory leaks.
18. README.md
    - A text file that's in Markdown format that describes how you would build/run the program.
19. DESIGN.pdf
    - Describes the design and thought process of the main program.

3 Structure and Pseudocode for node.c:

The huffman tree will require some sort of tree to use. This tree will be comprised of nodes, so we need to implement that first.

Node *node_create(uint8_t symbol, uint64_t frequency)
Creates a node containing the symbol of the node, the number of times that symbol shows up (store into frequency), and the left/right nodes. Return the address of the node created.

void node_delete(Node **n)
Delete the node. This is done by freeing the memory used for the node. Make sure to set the pointer to NULL after freeing.

Node *node_join(Node *left, Node *right)
Joins the left and right nodes and make a parent node. The parent node will be given the symbol '$' and the frequency of the parent node will be the sum of the left and right nodes. Return the address of the parent node.

void node_print(Node *n)
Presumably used for debugging our node ADT. Use this to print out whatever pertaining to the Node struct.

3.1 Note about the pseudocode:

- The left and right nodes, should be initialized as NULL when first creating a node.
- If creating the node was unsuccessful, free the node.
- When deleting a node, make sure to see if the node that was passed as an argument is even a node in the first place, don't try to clear something that isn't a node.

4 Structure and Pseudocode for pq.c:

The huffman encoding/decoding will also require a priority queue. We need a way to enqueue and dequeue things in a certain way.

void insertion_sort(PriorityQueue *q, Node *n)
Used to sort the priority queue in the correct sequence. Order is determined by node frequency. First find the location of where we should insert the node. Iterate over the nodes inside the pq. When we either find out that we are at the end of the priority queue, or if the frequency of the node passed in is less than or equal to the current node in the queue, we break. If none of those conditions are met, increment some index and move on to the next node in the queue.
Now we need to fix the queue, we do this by starting at the end of the queue and move everything up and decrement the index by 1. Stop the while loop when our index is less than or equal to our index from the while loop.

Set the node at the index from the while loop.

struct PriorityQueue
Create a struct that contains the things that make up a Priority queue. This includes:

- uint32_t head
- uint32_t tail
- uint32_t max_capacity
- Node **nodes

PriorityQueue *pq_create(uint32_t capacity)
Create a priority queue containing the head, the tail, max capacity, and a Nodes container to hold the nodes. If the priority queue was unsuccessful in creating, free the priority queue and set the pointer to NULL. Return the pointer to the priority queue.

void pq_delete(PriorityQueue **q)
Delete the Priority queue by freeing the memory used for the priority queue. Make sure to set the pointers to NULL.

bool pq_empty(PriorityQueue *q)
Check to see if the queue is empty by seeing if the queue head is pointing at the same position as the queue tail. Return true if it's empty, false otherwise.

bool pq_full(PriorityQueue *q)
Check if the queue is full. This is done by checking to see if the successor of the queue head is the queue tail. If it's full return true, otherwise false.

uint32_t pq_size(PriorityQueue *q)
Find and return the queue size by finding the difference of the queue head and tail.

bool enqueue(PriorityQueue *q, Node *n)
Set the node via calling our insertion function. Add one to the head. Return true if enqueueing was successful, otherwise false.

bool dequeue(PriorityQueue *q, Node **n)
Set the node to be the the item at the tail. Move all the nodes down by 1. Minus 1 from the head.

void pq_print(PriorityQueue *p)
Presumably used for debugging our priority queue ADT. Use this to print out whatever pertaining to the priority queue struct.

4.1 Note about the pseudocode:

- The node that takes the highest priority are nodes that have the smallest frequency.

- I decided to use an insertion sort. Performance should be fine due to the maximum theoretical size of some queue will be 256.
- For deleting, enqueueing, or dequeuing, always make sure to check if the priority queue exists before attempting the aforementioned. Same should be checked when checking if the queue is empty, full, and the size.
- When enqueueing a node, make sure the queue isn't full. The same should be done for dequeuing but check to see if the queue isn't empty.

5 Structure and Pseudocode for code.c

struct Code
Each character will be designated a code made of bits. This is where the characters will be represented as bits to save a ton of space for files.

Create a struct that contains the things that make up a Code. This includes:

- uint32_t top
- uint8_t bits[MAX_CODE_SIZE]

Code code_init(void)
Initialize the code of the size MAX_CODE_SIZE. Top should be initialize as 0. Return the pointer to the code.

uint32_t code_size(Code *c)
Return the exact size of the bits.

bool code_empty(Code *c)
Checks to see if the Code is empty. Return true if it's empty, false if otherwise.

bool code_full(Code *c)
Check to see if the Code is of length 256, if so return true, otherwise return false. This is because there are 256 ascii values that you'll ever see.

bool code_set_bit(Code *c, uint32_t i)
Perform a bitwise or at some index i to set the bit to 1. If successful return true, otherwise return false.

bool code_clr_bit(Code *c, uint32_t i)
Perform a bitwise and at the index i to set the bit to 0. If successful return true, otherwise return false.

bool code_get_bit(Code *c, uint32_t i)
Perform a bitwise nand at the index i to get the bit. If successful return true, otherwise return false.

bool code_push_bit(Code *c, uint32_t i)
Add a bit onto the Code struct. Make sure to check if pushing is possible. Return true if successful, otherwise return false.

bool code_pop_bit(Code *c, uint8_t *bit)
Remove a bit off of Code. Make sure to check if popping is possible. Return the value of the popped bit. Return true if successful, otherwise return false.

void code_print(Code *c)
Presumably used for debugging our Code ADT. Use this to print out whatever pertaining to the Code struct. I've went out of my way to print out the entire code in binary.

5.1 Note about the pseudocode

- code.c should rely on the definitions provided in the defines.h.
- Make sure to check that the code exists before checking for any conditions such as if it's full, empty, pushing/popping, and setting/clearing.
- When making the code make sure to set all the bits to 0 as a safety measure.

6 Pseudocode for io.c

Communications between different hierarchies can be relatively slow. We should make the most out of system reads/writes. To do so we need to fill a buffer to it's max before requesting to read or write something.

bool get_bit(uint8_t *buffer, int pointer)
Helper function to get a bit at a specific index.

void set_code(uint8_t bit)
Helper function to set a bit at a specific byte in a buffer.
int read_bytes(int infile, uint8_t *buf, int nbytes)
Make a while loop to keep calling read from the infile. Until either no more bytes can be read or when we've filled up our buffer, we will return the number of bytes that was read.

int write_bytes(int outfile, uint8_t *buf, int nbytes)
Similar to the read bytes function, we do the same with writing bytes.

bool read_bit(int infile, uint8_t *bit)
Given a file to read from, and the bit that we want, perform a bitwise and on the bit of a specific byte and store it into a buffer. Return true if there are still bits to read, otherwise return false to indicate that there are no more remaining bits to read. When the buffer is full we should only then read in more stuff from the infile.

void write_code(int outfile, Code *c)
Similar to read_bit, we will be writing to the outfile after filling the buffer to the fullest it can be. We should utilize our flush codes function to write the bits to the outfile.

void flush_codes(int outfile)
Fill in the rest of the leftover bits that weren't written to with 0's. Writes out the bits to the outfile.

6.1 Notes about pseudocode

- Making calls to read and specifying a certain amount of bytes to read isn't guaranteed to read in that many number of bytes. To accommodate for this we write a while loop to keep calling read. The same logic applies to write.
- The buffer and the pointer should be static in the read_bit function.
- The buffer and pointer should be static for write_code but for the whole file, not just the function.

7 Pseudocode for stack.c

Used to recreate the huffman tree after encoding. This is part of the process of decoding. We will need to maintain a stack of nodes so that we can rebuild the tree.

struct Stack
Make a struct stack to hold the head value, capacity, and the Nodes.

Stack *stack_create(uint32_t capacity)
Create the stack containing the capacity, head, and a Nodes container to hold the nodes. If the stack was unsuccessful in creating, free the stack and set the pointer to NULL.

stack_delete(Stack **s)
Delete the stack. This is done by freeing the memory used for the stack. Make sure to set the pointer to NULL after freeing.

bool stack_empty(Stack *s)
Check to see if the stack is empty by seeing if the stack head is less than or equal to 0.

bool stack_full(Stack *s)
Check if the stack is full. This is done by checking if the stack head is equal to the max capacity.

uint32_t stack_size(Stack *s)
Find the stack size by returning the head number.

bool stack_push(Stack *s, Node *n)
Pushes a node onto the stack. Return true if it was successful, otherwise return false.

bool stack_pop(Stack *s, Node **n)
Pops a node onto the stack and points at the popped item. Return true if it was successful, otherwise return false due to the stack being empty or not existing. You should decrement the head of the stack by 1 first before grabbing the node since the stack will always point to the empty slot of the stack.

void stack_print(Stack *s)
Presumably used for debugging our stack ADT. Use this to print out whatever pertaining to the Stack struct.

7.1 Notes about pseudocode

- Always make sure to check if the stack exists before performing any operations.
- I tried to follow the code given from the assignment pdf as closely as possible.

8 Pseudocode for huffman.c

Builds the huffman encoding, previous functions required in order to do this part.

Node *build_tree(uint64_t hist[static ALPHABET])
Constructs a Huffman tree. Given a histogram:

1. Create a priority queue.
2. Create a node for each unique char from the histogram that is greater than 0. Enqueue the nodes.
3. While the priority queue is greater than 1, dequeue two nodes. The first node will be the left node, second node will be the right node.
4. Join the nodes and enqueue that node.
5. The last node inside the priority queue will be the root node. Dequeue this node and return it.

void build_codes(Node *root, Code table[static ALPHABET])
Given the root node and a code table, construct a unique code table for each character. This will require a post order traversal. What you'll need to do is:

```
If the root node isn't null:
    Check to see if it's a leaf:
        If it is, in the code table for this specific node symbol, set the
        code to what it currently is.
    Otherwise:
        Push a 0 to the code
        Recurse the left side of the root node
        Pop the bit

        Push a 1 to the code
        Recurse the right side of the node
        Pop the bit
```

void dump_tree(int outfile, Node *root)

Write the tree all to the outfile. Each leaf should be labelled and printed with an L and it's corresponding symbol, and I for interior nodes. The code should look as follows:

```
If the root node isn't null:
    Recurse through the left side of the root node.
    Recurse through the right side of the root node.
    Check to see if the node is a leaf:
        If so write an L to the outfile then write the root symbol.
        Otherwise, write an I to the outfile.
```

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

Rebuild the Huffman tree. We will be given the tree size in nbytes, and the array of the tree from the tree_dump. Return the root node of the tree. This can be done by:

```
Create a stack big enough for the maximum size a file can be.
Iterate through nbytes (which should be the tree size):
    Check to see if the current character is an L:
        Create a node with the symbol that comes after the L.
        Push the node onto the stack.
    Check if the current character is an I:
        Pop the stack twice. The first pop will be the right child, second
        is the left child. Join the two nodes and push it back onto the stack.
There should be one leftover item in the stack, this will be our root node.
Pop it and return this node.
```

void delete_tree(Node **root)

Free the tree off the heap. This will require a post order traversal. Make sure to set the pointer of the tree to NULL.

8.1 Notes about pseudocode

- Some key words are defined in the define.h/huffman.h header files.
- Dumping the tree requires a post-order traversal of the Huffman tree.
- Rebuilding the tree should be rebuilt through post-order.

9 Pseudocode for encode.c

Things you'll need to do in encode.c to encode some file:

1. Compute a histogram of the infile. Read the infile in BLOCK chunks and count the number of occurrences of every character.
2. After computing the histogram, increment the 0th and 255th element by 1. This is to ensure that our file will have at least two nodes to work with and this will let us compress empty files.
3. Pass the computed histogram into your build tree function. This will give us the root node.

4. With the constructed tree, pass it to our build codes function along with a Code table. This will give us a Code table that we can use to represent characters as bits.
5. Create a header and write it out to the outfile.
6. Dump the tree to the outfile using our dump tree function.
7. Starting from the beginning of our file, we will read in the bytes and write the corresponding code table for each character seen in the file. This is done via our write code function.
8. Flush any remaining codes.
9. If verbose was specified, print out the statistics for how much space we saved from encoding.

9.1 Notes about pseudocode

- Don't forget to use getopt and switch cases to parse through user's inputs.
- If the user doesn't specify an infile, you'll need to make a temporary file to read in from stdin.
- Print out a helpful usage page for the user.
- The header's magic number field should be set to the macro defined in defines.h.
- Using fstat will allow us to grab the header permissions and file size of the infile.
- Tree size can be calculated via $3xunique\_symbols - 1$.
- Writing out the header will require us to cast the header as an array of uint8_t's.

10 Pseudocode for decode.c

Things you'll need to do in decode.c to decode some file:

1. Read in the header of the infile and check if the magic number matches with the one in defines.h. If it doesn't we won't be able to decode the message, so we'll need to exit the program.
2. Read in the tree portion of the file, this can be done using the header tree size specifications.
3. Call rebuild tree and pass in the read tree.
4. Now start reading in bits from the infile.
    - If the bit is 0, traverse down the left node of the root.
    - If the bit is 1, traverse down the right node of the root.
    - If the node is a leaf write out the corresponding symbol to the outfile. Start back at the root node afterwards.
5. If verbose was specified, print out the statistics for how much space we saved from decoding.

10.1 Noes about pseudocode

- Don't forget to use getopt and switch cases to parse through user's inputs.
- If the user doesn't specify an infile, you'll need to make a temporary file to read in from stdin.
- Print out a helpful usage page for the user.
- Reading in the header will require us to cast the array of uint8_t's to a header..

11 Error Handling:

Some problems and solutions I found while coding.

1. If the file that the user gave doesn't exist for reading, I'll return an error stating that the file couldn't be opened.
2. If the magic header doesn't match, this will signify that we can't decode the file, so I'll return an error stating the magic number doesn't match.
3. If the user were to input through stdin, I made a temp file so that we can make it possible to read through stdin.

12 Credit:

- I've tried to follow the information provided in asgn6 to implement my programs.
- I attended Eugene's section on February 18th and 25th. He helped me understand the basics of the assignment. He also helped me make the temp file for reading in stdin.
- I attended Eugene's office hours to get a better understanding of io.c and huffman.
- I attended Brian's section on February 24th. He helped me understand some of the ideas behind double pointers and getting started on io.c / huffman.c.
- Audrey's example design pdf helped me structure my own design pdf. This can be found on this quarters cse13s discord server in the asgn-1-faq-n-tips channel.
- The C Programming Language book may have influenced some of the code I've written.
- Ben's tutorial on how to compile and link multiple files together was really helpful. I also followed his scan-build guide on the hilalmorrar.com website.
- I've used my own Makefile from asgn5 as reference to build the makefile for this assignment.
- On the cse13s discord server, user Gecko10000#7137 helped me figure out a problem with my rebuild_tree function.
- User 190n#1979 on the discord server helped me find a bug in my priority queue, clear up a few misconceptions with writing the tree to the outfile and the header struct.
- I tried following the assignment pdf specifications as closely as possible.