Krisha Sharma

CSE13s

Winter 2022 Long

02/18/2022

Assignment 6 DESIGN.pdf

Description of Program:

Assignment 6: Huffman Coding

The first task for this assignment is to implement a Huffman encoder. This encoder will read in an input file, find the Huffman encoding of its contents, and use the encoding to compress the file. The encoder program, named encode, must support any combination of the command-line options specified in the assignment 6 PDF.

The second task for this assignment is to implement a Huffman decoder. This decoder will read in a compressed input file and decompress it, expanding it back to its original, uncompressed size. Your decoder program, named decode, must support any combination of the command-line options specified in the assignment 6 PDF.

Files to be included in directory "asgn6":

- `encode.c`
    - This contains the implementation of the Huffman encoder.
- `decode.c`
    - This contains the implementation of the Huffman decoder.
- `defines.h`
    - This file will contain the macro definitions used throughout the assignment. You may not modify this file.

- `header.h`
    - This will contain the struct definition for a file header. You may not modify this file.
- `node.h`
    - This file will contain the node ADT interface. This file will be provided. You may not modify this file.
- `node.c`
    - This file will contain your implementation of the node ADT.
- `pq.h`
    - This file will contain the priority queue ADT interface. This file will be provided. You may not modify this file.
- `pq.c`
    - This file will contain your implementation of the priority queue ADT. You must define your priority queue struct in this file.
- `code.h`
    - This file will contain the code ADT interface. This file will be provided. You may not modify this file.
- `code.c`
    - This file will contain your implementation of the code ADT
- `io.h`
    - This file will contain the I/O module interface. This file will be provided. You may not modify this file
- `io.c`

    - This file will contain your implementation of the I/O module.
- `stack.h`
    - This file will contain the stack ADT interface. This file will be provided. You may not modify this file.
- `stack.c`
    - This file will contain your implementation of the stack ADT. You must define your stack struct in this file.
- `huffman.h`
    - This file will contain the Huffman coding module interface. This file will be provided. You may not modify this file.
- `huffman.c`
    - This file will contain your implementation of the Huffman coding module interface.
- Makefile
    - `CC = clang` must be specified
    - `CFLAGS = -Wall -Wextra -Werror -Wpedantic` must be specified
    - `pkg-config` to locate compilation and include flags for the GMP library must be used.
    - `make` must build the encoder and the decoder, as should `make all`.
    - `make encode` should build only the `encode` program.
    - `make decode` should build only the `decode` program.
    - `make clean` must remove all files that are compiler generated.
    - `make format` should format all the source code, including the header files

- README.md
    - Text file in Markdown format that describes how to build and run the program, how the program handles erroneous inputs, and any problems encountered while developing the program.
- DESIGN.pdf
    - This must be a PDF. The design document should answer the pre-lab questions, describe the purpose of your program, and communicate its overall design with enough detail such that a sufficiently knowledgeable programmer would be able to replicate your implementation. This does not mean copying your entire program verbatim. You should instead describe how your program works with supporting pseudocode. C code is not considered pseudocode.

General Notes:

- Huffman coding is a data compression algorithm
- How does the actual algorithm work?
    - The idea behind huffman coding is that we want to be able to represent symbols, such as an ASCII character, that appear frequently with a small amount of bits, and ones that don't appear as often with a large amount of bits.
    - Run length encoding (RLE)
        - Given a string of “aaaabbcc” (9 symbols/bytes) you can convert it to → 4a2b3c (6 symbols/bytes)
    - Huffman coding strives to give each symbol a unique bit string
        - The bit string is what we call a code

        - Frequently appearing symbols should have a shorter bit strings whereas infrequently appearing symbols have longer bit strings
        - Given some input “ababac” where a is represented in code as 0, b is 10, and c is 11.
            - The string of “ababac” would then translate to → 010010011
                - Going from 6 bytes (48 bits) with the string “ababac” to 9 bits with the code string 010010011
- How do you assign each symbol a unique code string?
    - This is done through a huffman tree
    - 1. Check which symbols appear frequently and infrequently and make a histogram, beginning by just walking over an input.
        - There are 256 total ASCII characters (inc. extended)
        - Create an array of 256 indices (0 to 255) somewhere in the middle, where indices 97, 98 and 99 exist. The indice value for ‘a’ is 97, ‘b’ is 98, and ‘c’ is 99.
        - Every time you come across an symbol that appears, increment the count by one and do this until the input is exhausted.
            - The histogram now gives us the frequencies of each character/symbol and how many times they appeared
        - Now create a “container” for them called a node.
        - A node is going to contain a symbol and the number of times it appeared (the frequency)

- 2. Create a priority queue, where the node with the highest priority is at the front and the node with the lowest priority is at the tail.
    - The node with the highest priority is the node with the smallest frequency
    - If there are multiple nodes with the same frequency, just break arbitrarily.
    - Min heap, insertion sort, binary search are all techniques you can use to sort through the heap in your priority queue
- 3. Construct the huffman tree
    - The frequency of the parent is the frequency of the two children added together.
    - Psuedocode:

```
while queue.size > 1
        dequeue 1 for the left child
        dequeue 1 for the right child
        the parent = join(left child, right child)
        enqueue parent
```

    - The tree works from the bottom up to combine nodes together
    - The final node is called the “root”
- 4. Walk the tree to build codes
    - Start off at the root of the now constructed huffman tree
    - Walk down the tree
    - When you go to the left you label your path with a 0 and anytime you go to the right, you label your path with a 1.
    - Code traversal @ 24:00 in eugene’s yuja video recording on 02/18/2022

- Pseudocode:

```
function walk(node)
     If node == NULL
          If the node has no children (if it is a
          leaf)
               Assign the current code to the
               symbol
          Else
               Push a 0 onto the code
               walk(node → left)
               pop the code
```

-

Node Functions:

Node *node_create(uint8_t symbol, uint64_t frequency)

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

pseudocode:

```
Node *node_create() {
     initialize a node pointer
     set n->left to NULL;
     set n->right to NULL;
     set n->symbol to symbol;
     Set n->frequency to frequency;
     return the node pointer;
```

```
}
```

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.

pseudocode:

```
void node_delete() {
    free the memory for the node;
    set the node pointer to NULL;
}
```

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.

pseudocode:

```
Node *node_join() {
    initialize and set the parent node symbol to '$';
    initialize and set the parent node frequency to the
    left->frequency + right->frequency;
    create the parent node using node_create();
    set the parent node's left child to left;
    set the parent node's right child to right;
    return the parent node;
}
```

void node_print(Node *n)

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

pseudocode:

```
void node_print() {
     print out anything that might be useful for debugging
     purposes;
}
```

Priority Queue Functions:

PriorityQueue *pq_create(uint32_t capacity)

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

pseudocode:

```
PriorityQueue *pq_create() {
     initialize a priority queue pointer q and allocate
     memory for it;
     if (q)
          set q->head to 0;
          set q->tail = 0;
          set q->capacity to capacity;
          make queue Q
     return q;
}
```

void pq_delete(PriorityQueue **q)

The destructor for a priority queue. It is important to set the pointer to NULL after freeing the memory for a priority queue.

pseudocode:

```
void pq_delete() {
     if the queue exists
```

```
            free((*q)->Q), first free what is inside the
            queue;

            free(*q), then free the queue;

            *q = NULL, set the queue to NULL;
    }
```

bool pq_empty(PriorityQueue *q)

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

pseudocode:

```
bool pq_empty() {
     if the capacity of the queue is 0
          return true;
     else
          return false;
}
```

bool pq_full(PriorityQueue *q)

Returns true if the priority queue is full and false otherwise

pseudocode:

```
bool pq_full() {
     if the capacity of the queue is equal to q->head
          return true;
     else
          return false;
}
```

uint32_t pq_size(PriorityQueue *q)

Returns the number of items currently in the priority queue.

pseudocode:

```
uint32_t pq_size() {
     returns q->capacity;
}
```

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.

pseudocode:

```
bool enqueue() {
     if (q)
          if the priority queue is full {
               return false;
          n = q->Q[q->head];
          increment the q->head pointer by 1;
          for (i = 1, i < q->head, i ++)
               j = 1, copy what index you are on;
               Node *tempvar = q->Q[i], save the value;
               while j is greater than 0 and tempvar is
               less than q->Q[j - 1] {
                    set q->Q[j] = q->Q[j - 1], swap;
                    j -= 1, decrement the copy of the
                    index;
               q->Q[j] = tempvar;
          return true;
     else
          return false;
```

```
}
```

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.

pseudocode:

```
bool dequeue() {
     if (q)
          if the priority queue is empty
               return false;
          decrement the value of q->head by 1;
          set n equal to &q->Q[q->head];
          return true;
     else
          return false;
}
```

void pq_print(PriorityQueue *q)

A debug function to print a priority queue.

pseudocode:

```
void pq_print() {
     print out anything that will help debug;
}
```

Codes Functions:

Code code_init(void)

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

pseudocode:

```
Code code_init(void) {
     Code c, create a new Code on the stack;
     c.top = 0, set the top to 0;
     return c;
}
```

uint32_t code_size(Code *c)

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

pseudocode:

```
uint32_t code_size() {
     return c->top;
}
```

bool code_empty(Code *c)

Returns true if the Code is empty and false otherwise.

pseudocode:

```
bool code_empty() {
     if the value of c->top is 0
          return true;
     else
          return false;
```

```
}
```

bool code_full(Code *c)

Returns true if the Code is full 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).

pseudocode:

```
bool code_full() {
     if c->top = ALPHABET
          return true;
     else
          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.

pseudocode:

```
bool code_set_bit() {
     if i is out of range, meaning the value of i is
     greater than ALPHABET / 8, or equal to 0
          return false;
     else
          perform the necessary bitwise arithmetic;
          return true;
}
```

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.

pseudocode:

```
bool code_clr_bit() {
     if i is out of range, meaning the value of i is
     greater than ALPHABET / 8, or equal to 0
          return false;
     else
          perform the necessary bitwise arithmetic;
          return true;
}
```

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.

pseudocode:

```
bool code_get_bit() {
     if i is out of range, meaning the value of i is
     greater than ALPHABET / 8, or equal to 0
          return false;
     else
          if the bit does not equal 1
               return true;
}
```

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.

pseudocode:

```
bool code_push_bit() {
     if the code is full before pushing
          return false;
     else
          c->bits[c->top] = bit;
          increment the value of c->top by 1;
          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.

pseudocode:

```
bool code_pop_bit() {
     if the code is empty before you pop
          return false;
     decrement the value of c->top by 1;
     *bit = c->bit[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.

pseudocode:

```
void code_print() {
     print out anything that would be useful for debugging;
}
```

I/O Functions:

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

This will be a useful wrapper function to perform reads. As you may know, the read() syscall does not always guarantee that it will read all the bytes specified (as is the case with pipes). For example, a call could be issued to read a a block of bytes, but it might only read part of a block. So, we 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. You should use this function whenever you need to perform a read.

pseudocode:

```
int read_bytes() {
     initialize a variable to keep track of bytes_read and
     set to 0;
     initialize a variable to keep track of the
     current_bytes_read and set to 0;
     if the value of nbytes is 0
          return false;
     set current_bytes_read to the bytes being read in from
     infile.
```

```
        while (there are bytes to be read, current_bytes_read
        > 0)

             bytes_read += current_bytes_read;

             if the value of bytes_read = nbytes

                  break;

        return the number of bytes read in from the infile,
        return bytes_read;

   }
```

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(). 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. This function should be used whenever you need to perform a write.

pseudocode:

```
int write_bytes() {

     initialize a variable to keep track of bytes_write and
     set to 0;

     initialize a variable to keep track of the
     current_bytes_write and set to 0;

     if the value of nbytes is 0

          return false;

     set current_bytes_write to the bytes being written
     out.

     while (there are bytes to be written,
     current_bytes_write > 0)

          bytes_write += current_bytes_write;

          if the value of bytes_write = nbytes
```

```
            break;

        return the number of bytes written out to the outfile,
        return bytes_write;

    }
```

bool read_bit(int infile, uint8_t *bit)

Because it is not possible to read a single bit from a file, in order to create this function we need to read in a block of bytes into a buffer and dole out bits one at a time. Whenever all the bits in the buffer have been doled out, you can simply fill the buffer back up again with bytes from infile. This is exactly what you will do in this function. 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 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.

pseudocode:

```
bool read_bit() {

    make a static buffer[BLOCK];

    initialize a static index and set it to 0 because no
    bits have been read in yet;

    if the buffer is empty

        set the bit_index equal to 0;

        read in the file and fill the buffer;

    current_byte = buffer[byte_index / 8];

    current_bit = current_byte >> (bit_index % 8);

    current_bit &= 0x1;

    set the bit pointer to current_bit;

    increment the bit_index by 1;
```

```
        byte_index  -= bit_index / 8;
        if the byte_index = 0
            return false;
        else
            return true;
    }
```

void write_code(int outfile, Code *c)

The same bit-buffering logic used in read_bit() will be used here as well. This function will also make use of a static buffer (we recommend this buffer to be static to the file, not just this function) and an index. Each bit in the code c will be buffered into the buffer. The bits will be buffered starting from the 0 th bit in c. When the buffer of BLOCK bytes is filled with bits, write the contents of the buffer to outfile.

pseudocode:

```
    void write_code() {
        if the bit at the buffer index is equal to 1
            perform the bitwise arithmetic necessary and set
            the bit to 1;
            increment the buffer index by 1;
        if the bit at the buffer index is equal to 0
            perform the bitwise arithmetic necessary and set
            the bit to 0;
            increment the buffer index by 1;
        write out the contents of the buffer to outfile using
        write_bytes();
    }
```

void flush_codes(int outfile)

It is not always guaranteed that the buffered codes will align nicely with a block, which means that it is possible to have bits leftover in the buffer used by write_code() after the input file has been completely encoded. 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.

pseudocode:

```
void flush_codes() {
     while the buffer index is not 0
          perform the necessary bitwise arithmetic;
          decrement the buffer counter by 1;
     write out the leftover buffered bits to outfile using
     write_bytes();
}
```

Stacks Functions:

Stack *stack_create(uint32_t capacity)

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

pseudocode:

```
Stack *stack_create() {
     initialize a variable to track the top of the stack;
     initialize a variable for capacity;
     Node **items;
} ;
```

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.

pseudocode:

```
void stack_delete() {
      free the memory allocated by the stack;
      set the stack pointer *s to NULL;
}
```

bool stack_empty(Stack *s)

Returns true if the stack is empty and false otherwise.

pseudocode:

```
bool stack_empty() {
      return s->top == 0;
}
```

bool stack_full(Stack *s)

Returns true if the stack is full and false otherwise.

pseudocode:

```
bool stack_full() {
      return s->top == s->capacity;
}
```

uint32_t stack_size(Stack *s)

Returns the number of nodes in the stack.

pseudocode:

```
bool stack_size() {
```

```
        return s->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.

pseudocode:

```
bool stack_push() {
        if the stack is full before pushing a node
            return false;
        s->items[s->top] = n;
        increment s->top by 1;
        return true;
    }
```

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 successful popping of a node.

pseudocode:

```
bool stack_pop() {
        if the stack is empty before popping a node
            return false;
        decrement s->top by 1;
        *n = s->items[s->top];
        return true;
    }
```

void stack_print(Stack *s)

A debug function to print the contents of a stack.

pseudocode:

```
bool stack_print() {
     print out the contents of the stack for debugging
     purposes;
}
```

Huffman Coding Module Functions:

Node *build_tree(uint64_t hist[static ALPHABET])

Constructs a Huffman tree 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 ALPHABET number of indices.

pseudocode:

```
Node *build_tree() {
     initialize capacity at 257 to account for the open
     spot that will be made by the circular queue;
     create a priority queue using pq_create(capacity);
     initialize an index variable and set it to 0;
     for (index = 0, index <= 256, index += 1)
          if the histogram[index] does not equal 0
               create a child node at that index;
               enqueue the node to the priority queue;
     while the priority queue's size is greater than
          dequeue the left node;
```

```
            dequeue the right node;
            join both nodes together and set equal to the
            parent node using node_join();
            enqueue the parent node to the priority queue;
        initialize the root
        dequeue the root;
        return the root;
    }
```

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.

pseudocode:

```
void build_codes() {
    if the root node exists
        if the the left and right nodes do not exist
            Set c  to table[node->symbol];
        else
            code_push_bit(c, 0);
            build(node.left, table);
            code_pop_bit(c);
            code_push_bit(c, 1);
            build(node.right, table);
            code_pop_bit(c);
}
```

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.

pseudocode:

```
void dump_tree() {
     if (root)
          dump(outfile, root->left);
          dump(outfile, root->right);
          if (!root->left & !root->right)
               write(‘L’);
               write(node->symbol)
          else :
               write (‘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.

pseudocode:

```
Node *rebuild_tree() {
     initialize a stack pointer;
     initialize a variable L = ‘L’;
     initialize a variable I = ‘I’;
     for (i = 0, i < nbytes, i += 1)
          if (tree[i] = L)
               decrement the value of nbytes by 1;
```

```
                create a child node;

                use stack_push to push the child node onto
                the stack;

           if (tree[i] = I)

                pop the left and right child from the stack;

                use node_join() to join the left and right
                child and set it equal to the Node pointer
                *parent;

                use stack_push to push the parent node onto
                the stack;

      Node *root;

      stack_pop(stack, &root);

      use stack_delete to delete the stack;

      return root;

}
```

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.

pseudocode:

```
void delete_tree() {

      if (*root)

           recursively call delete_tree in order to delete
           the left node

           recursively call delete_tree in order to delete
           the left node;

      free the pointer *root;

      set the pointer *root to NULL;

}
```

Error Handling:

- Please note that I was not able to complete the assignment in time and therefore the code does not work properly.

Citations:

- Professor Long is cited throughout this assignment for the following:
    - Pseudocode in the assignment 6 PDF document
    - Tomai (Elmer) on discord in the CSE13s - Winter 2022 - Professor Long server
- Throughout this assignment, I had high-level pseudocode collaboration with my sister Twisha Sharma (tvsharma). We bounced ideas off of each other and generally talked out the best ways to go about implementing Huffman coding.
    - We also specifically had high-level pseudocode collaboration in io.c and huffman.c.
- I watched the video recording of Eugene's section on 02/18/2022. The section was focused on Assignment 6 and during it, Eugene covered many important topics that were relevant to the assignment. He generally went over the best way to begin the assignment and gave a basic pseudocode/overview of what we should do.
- I attended Audrey's tutoring session on 02/25/2022 and during it she helped look over my node.c.
- I attended Brian's tutoring session on 02/28/2022 and during it he helped look over my pq.c and also helped to explain how to proceed with code.c. Brian gave me a general overview on the best way to go about implementing the functions.
- I attended Ben's section on 03/01/2022 and during it he helped me understand how to do read_bit in io.c. Ben also looked over my read_bytes and write_bytes during the tutoring

session and explained how to do them properly. Ben also helped me by looking over my code.c file.

- I attended Audreys tutoring session on 03/02/2022 and during it she helped me by looking over my huffman.c and correction anything she saw that was incorrect, as well as by helping me fix a syntax error in my stack.c, the error was specifically in stack_pop()
- I attended Brian's tutoring session on 03/02/2022 for help with looking over my io.c file as well as for help in fixing compilation and syntax errors.