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.

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.

Insert pseudocode here

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.

Insert pseudocode here

bool pq_empty(PriorityQueue *q)

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

Insert pseudocode here

bool pq_full(PriorityQueue *q)

Returns true if the priority queue is full and false otherwise

Insert pseudocode here

uint32_t pq_size(PriorityQueue *q)

Returns the number of items currently in the priority queue.

Insert pseudocode here

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.

Insert pseudocode here

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.

Insert pseudocode here

void pq_print(PriorityQueue *q)

A debug function to print a priority queue. This function will be significantly easier to implement if your enqueue() 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).

Insert pseudocode here

Codes Functions:

Code code_init(void)

You will immediately notice that this “constructor” function is unlike any of the other constructor functions you have implemented in the past. You may also have noticed, if you glanced slightly ahead, that there is no corresponding destructor function. This is an engineering decision that was made when considering the constraints of the Huffman coding algorithm. This function will not require any dynamic memory allocation. 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.

Insert pseudocode here

uint32_t code_size(Code *c)

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

Insert pseudocode here

bool code_empty(Code *c)

Returns true if the Code is empty and false otherwise.

Insert pseudocode here

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).

Insert pseudocode here

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.

Insert pseudocode here

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.

Insert pseudocode here

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.

Insert pseudocode here

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.

Insert pseudocode here

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.

Insert pseudocode here

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.

Insert pseudocode here

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.

Insert pseudocode here

<u>int write_bytes(int outfile, uint8_t *buf, int nbytes)</u>

This functions 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. You should use this function whenever you need to perform a write.

Insert pseudocode here

<u>bool read_bit(int infile, uint8_t *bit)</u>

You should all know by now that it is not possible to read a single bit from a file. What you can do, however, is 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.

Insert pseudocode here

<u>void write_code(int outfile, Code *c)</u>

The same bit-buffering logic used in read_bit() will be used in 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.

Insert pseudocode here

<u>void flush_codes(int outfile)</u>

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.

Insert pseudocode here

Stacks Functions:

<u>Stack *stack_create(uint32_t capacity)</u>

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

Insert pseudocode here

<u>void stack_delete(Stack **s)</u>

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

Insert pseudocode here

<u>bool stack_empty(Stack *s)</u>

Returns true if the stack is empty and false otherwise.

Insert pseudocode here

<u>bool stack_full(Stack *s)</u>

Returns true if the stack is full and false otherwise.

Insert pseudocode here

<u>uint32_t stack_size(Stack *s)</u>

Returns the number of nodes in the stack.

Insert pseudocode here

<u>bool stack_push(Stack *s, Node *n)</u>

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.

Insert pseudocode here

<u>bool stack_pop(Stack *s, Node **n)</u>

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.

Insert pseudocode here

<u>void stack_print(Stack *s)</u>

A debug function to print the contents of a stack.

Insert pseudocode here

Huffman Coding Module Functions:

<u>Node *build_tree(uint64_t hist[static ALPHABET])</u>

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.

Insert pseudocode here

<u>void build_codes(Node *root, Code table[static ALPHABET])</u>

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.

Insert pseudocode here

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.

Insert pseudocode here

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.

Insert pseudocode here

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.

Insert pseudocode here

Error Handling:

- As I encounter errors while implementing this assignment they will be documented here.

Citations:

- As I begin to implement this assignment and work through the issues, any citations I have will be documented here, as well as in the program itself should the citation be specific enough.