Stanley Chen

CSE 13S

Assignment 5 Design.pdf

Purpose of program:

_____This program will have multiple .c files, which will be used to implement Huffman coding. Many of the .c files are abstract data type (ADT) files which aid in the encoding/decoding for the Huffman coding. Huffman coding is a form of compression that uses binary trees of nodes consisting of internal and leaf nodes. This program will have an encoder which encodes data with Huffman coding, and will also have a decoder that decodes the data back to its original state.

Deliverables for assignment 5:

- encode.c
    - encode.c contains the code for the Huffman encoder
- decode.c
    - decode.c contains the code for the Huffman decoder
- defines.h
    - defines.h contains the macro definitions used for the Huffman coding files
- header.h

    - header.h contains the struct definition for a file header
- node.h
    - node.h contains the interface for the node ADT
- node.c
    - node.c contains the code for the node ADT
- pq.h
    - pq.h contains the interface for the priority queue ADT
- pq.c
    - pq.c contains the code for the priority queue ADT
- code.h
    - code.h contains the interface for the code ADT
- code.c
    - code.c contains the code for the code ADT
- io.h
    - io.h contains the interface for the I/O module
- io.c
    - io.c contains the code for the I/O module
- stack.h
    - stack.h contains the interface for the stack ADT

- stack.c
  - stack.c contains the code for the stack ADT
- huffman.h
  - huffman.h contains the interface for the Huffman coding module
- huffman.c
  - huffman.c contains the code for the Huffman coding module interface
- Makefile
  - CC = clang must be specified
  - CFLAGS = -Wall -Wextra -Werror -Wpedantic must be specified
  - make must build the encoder and decode, as should make all
  - make encode should just build the encoder
  - make decode should just build the decoder
  - make clean must remove all files that are compiler generated
  - make format should format all your source code, including the header files.
- README.md
  - README.md is a text files that uses markdown formatting that includes:
    - the title of the program
    - the purpose and usage of the program

        - a description of the program and how it is used
    - how to build the program
        - the commands needed to run to produce the program
    - any known bugs
- DESIGN.pdf
  - DESIGN.pdf is a pdf file that describes the design and design process of the program, with information like pseudocode and diagrams

<u>Pseudocode for assignment 5:</u>

```
- node.c
    - node_create (the constructor for node):
        - Dynamically allocate memory for node
        - set node's symbol to symbol
        - Set node's frequency as frequency
    - node_delete (the destructor for node):
        - Free dynamically allocated memory
        - Set pointers to NULL
    - node_join:
        - Create a parent node which joins a left and right child node
        - Set the parent's symbol to '$' and the frequency to (left child's
          frequency + right child's frequency)
```

        - Return the parent node
    - node_print:
        - Function that aids in debugging and prints various node items
- pq.c
    - pq_create (the constructor for priority queue):
        - Dynamically allocate memory for priority queue
        - Set priority queue’s capacity to capacity
        - Dynamically allocate memory for the items node
        - Set priority queue’s size to 0
        - return priority queue
    - pq_delete (the destructor for priority queue):
        - Free the items node
        - Free dynamically allocated memory
        - Set pointers to NULL
    - pq_empty:
        - If priority queue’s size is 0
            - Then return true
        - Else
            - Return false

```
- pq_full:
    - If priority queue's size is the capacity
        - Return true
    - Else
        - Return false
- pq_size:
    - Return the size of priority queue
- enqueue:
    - If priority queue is full
        - Return false
    - Else
        - Enqueue node into priority queue (items)
        - Return true
- dequeue:
    - If priority queue is empty
        - Return false
    - Else
        - Dequeue the highest priority node from priority queue
          (items)
```

            - Return true
    - pq_print:
        - Print out node items
        - Using a heap, only has partial ordering
            - Display a tree
- code.c
    - code_init (constructor for code):
        - Sets code’s top to 0
        - Zero out the bits in code’s bits array
        - Return bits
    - code_size:
        - Return the size of code
    - code_empty:
        - If code is empty
            - Return true
        - Else
            - Return false
    - code_full:
        - If code is full (ALPHABET macro)

```
        - Return true
    - Else
        - Return false
- code_set_bit:
    - If i is out of range
        - Return false
    - Else
        - Set bits[i] to 1
        - Return true
- code _clr_bit:
    - If i is out of range
        - Return false
    - Else
        - Set bits[i] to 0
        - Return true
- code_get_bit:
    - Get value of bit at index i
    - If i is out of range or bits[i] = 0
        - Return false
```

    - Else if bits[i] = 1
      - Return true
  - code_push_bit:
    - If code is full (call code_full)
      - Return false
    - Else
      - Push bit onto code
      - Return true
  - code_pop_bit:
    - If code is empty (call code_empty)
      - Return false
    - Else
      - Pop bit out of code and pass into bit pointer
      - Return true
  - code_print:
    - Debugging function that prints out code to see if code_pop_bit and code_push_bit are working properly
- io.c
  - read_bytes (wrapper function that performs reads):

    - Use a loop to call read()
        - Stop loop when all bytes are read (nbytes) or no more bytes to read
        - Read bytes into byte buffer buf
        - Returns the number of bytes read from input (infile)
- write_bytes (wrapper function that performs writes):
    - Use a loop to call write()
        - Stop loop when all specified bytes are written (nbytes) from byte buffer buf or no bytes were written
        - Read bytes into byte buffer buf
        - Returns the number of bytes written from output (outfile)
- read_bit:
    - Read in blocks of bytes into a buffer
        - Get bits one by one
            - Once done, read in bits into buffer again
        - Use a static buffer of bytes (treat as a bit vector), index into buffer to track which bit to return to the pointer bit

            - Buffer stores BLOCK # of bytes
                - BLOCK is defined in defines.h
        - Return false if there are no more bits to read
        - Return true if there are still more bits to read
    - write_code:
        - Read in blocks of bytes into a buffer
            - Get bits one by one
                - Once done, read in bits into buffer again
            - Use a static buffer of bytes (treat as a bit vector), index into buffer to track which bit to return to the pointer bit
                - Buffer stores BLOCK # of bytes
                    - BLOCK is defined in defines.h
        - Write contents of the buffer to outfile when the buffer of BLOCK bytes is filled with bits
    - flush_codes:
        - Zero out any extra buffered bits
- stack.c
    - stack_create (the constructor for stack):

```
        -   Dynamically allocate memory for stack
        -   Set stack's capacity to capacity
        -   Set stack's top to 0
        -   Dynamically allocate memory for items node
        -   Return stack
    -   stack_delete (the destructor for stack):
        -   Free the items node
        -   Free dynamically allocated memory
        -   Set pointers to NULL
    -   stack_empty:
        -   If stack's size is 0
            -   Then return true
        -   Else
            -   Return false
    -   stack_full:
        -   If stack's size is the capacity
            -   Return true
        -   Else
            -   Return false
```

```
- stack_size:
    - Return the size of stack
- stack_push:
    - If stack is full (call stack_full)
        - Return false
    - Else
        - tems[top] = n
        - Add one to top
        - Return true
- stack_pop:
    - If stack is empty (call stack_empty)
        - Return false
    - Else
        - Subtract one from top
        - Pointer n = items[top]
        - Return true
- stack_print:
    - Debugger function that prints out the stack to see if the pop
      and push functions work
```

- huffman.c
    - build_tree:
        - Takes in a computed histogram
            - Histogram hist should have at least ALPHABET # of indices
                - ALPHABET macro from defines.h
        - Constructs a Huffman tree
            - [work in progress]
        - Returns root node of constructed tree
    - build_codes:
        - Build a code for each symbol in the Huffman tree
            - Copy the constructed codes to the code table (table)
                - Table has ALPHABET indices, one for each possible symbol
                    - ALPHABET macro from defines.h
    - rebuild_tree:
        - Reconstruct Huffman tree
            - Use data stored in tree_dump, use nbytes to determine the length in bytes to use for tree_dump

    - Return root node of reconstructed tree
  - delete_tree (destructor for Huffman tree):
    - post-order traversal of the tree
      - Free all the nodes
    - Set pointers to NULL
- encoder.c:
  - Read through infile
    - Construct a histogram
      - Histogram is an array of ALPHABET (256) # of uint64_ts
        - ALPHABET macro from defines.h
      - Increment count of element 0 through 255 one by one in histogram
  - Construct a Huffman tree using a priority queue
    - Use build_tree()
    - Create a priority queue
      - For each symbol in histogram which frequency > 0, create a node, and add to priority queue

- While there are 2 or more nodes in the priority queue, call dequeue() to dequeue two nodes. First is left node, second is right node
    - Use node_join() on the nodes and enqueue parent node
    - Last node is root of the Huffman tree
- Construct a code table using build_codes()
    - Code table is an array thats ALPHABET # of Codes
        - ALPHABET macro from defines.h
    - Create code using code_init()
        - Start at root, perform a post-order traversal
        - If node is a leaf
            - Save current code to code table
        - Else
            - Node is an interior node, so push a 0 to code and recurse down the left link
            - After recusing, pop a bit from code, push 1 to code and recurse down right link
                - Pop a bit from c when returning from right link

        - Construct a header from header.h
            - Write header to outfile
            - Perform post-order traversal of Huffman tree to write to outfile
                - Use 'L' followed by symbol, use 'I' for interior nodes
            - For infile, write corresponding code for each symbol to outfile using write_code()
                - Once done flush remaining buffered codes with flush_codes()
        - Close infile and outfile
- decoder.c:
    - Read in header from infile, verify magic #
        - If magic # does not match
            - Display error message and quit
        - Else
            - Continue
    - Set permissions with fchmod()
    - Read dumped tree from infile into array thats tree_size long

- Reconstruct Huffman tree using rebuild_tree()
    - Tree_dump is nbytes long is the dumped tree
        - Iterate over tree_dump from 0 to nbytes
        - If element = “L”
            - Next symbol is a symbol, create a new node with node_create()
                - push node to stack
        - If element = “I”
            - Pop stack once to get right child of interior node
                - Pop again to get left child
                    - Join left and right nodes with node_join()
                        - Push node to stack
        - The one node left is the root of the tree
    - Read infile one bit at a time with read_bit()
        - Begin at root of Huffman tree
            - If 0 is read
                - Go to the left child of node

```
            - If 1 is read (else)
                - Go to the right child of noce
            - If at leaf node
                - Write leaf symbol to outfile
                - Reset current node back to root of
                  the tree
        - Repeat until the size of decoded symbols matches
          the size of the original file (use file_size from
          infile)
- Close infile and outfile
```