Assignment 7 - A (Huffman Coding) Tree Grows in Santa Cruz

General Idea:

The content of files will be encoded using the Huffman tree algorithm and decoded using the same tree.

1. Makefile
    a. The makefile will automatically build the program when the “make” command is used on the command line

Pseudocode

-Include the clang label to compile
-include the cflags
-include the lflags
-include the all variable to store the target files encode and decode

-create variable for encode and include dependencies of encode.o, node.o, pq.o, code.o, io.o, stack.o, and huffman.o
    -add cflags and lflags to the file
-create variable for decode and include dependencies of decode.o, node.o, pq.o, code.o, io.o, stack.o, and huffman.o
    -add cflags and lflags to the file

-convert the executable files in .c form to .o so that they can be executable

-initialize clean that will remove .o files
-initialize the format option that will use clang-format on all code

2. encode.c
    a. Contains implementation of main() function
        i. Implement as written on document with options: -h, -i, -o, -v

Pseudocode

-Declare the main loop
    -initiate variables for verbose output and the files

-create while loop with all the options
    -if the option is i
        -set the input file
    -if the option is o
        -set the output file
    -if the option is v
        -set the verbose output to true
    -If the option is h
        -print the help option
    -set the default to the help option

-set the infile to the in stream
    -if the file can't be opened
        -print the error message
        -clear the variables
        -return 1
-set variables for the original message

-if the verbose output is set
    -print the uncompressed file size, the compressed file size, space saving with formula

-create the histogram array and set it's first and second bit to 0 and 1
-compute a histogram of the file by counting the number of occurrences of each unique symbol in the file
-construct Huffman tree with the histogram using a priority queue (build_tree())
    -create priority queue, for each symbol histogram where frequency is greater than 0, create a corresponding node and insert into queue
    -while there are more than 2 nodes in the queue, dequeue two nodes and join them
    -when there's only one node left, this is the root of the Huffman tree
-construct the code table with each index of the table representing a symbol and the value at the index of the symbol's code (stack of bits will be used)
    -create a new code using code_init() and perform post-order traversal of Huffman tree
    -if current node is a lead, save code into code table
    -else current node must be interior node, so push 0 to c and recurse down left link
    -after your go down left link, pop a bit from the code object then return to right link
-construct a header struct
-write the constructed header to outfile
-use post-order traversal of Huffman tree to encode the file (use dump_tree())
-step through tree and emit code into output file using write_code() and flush_codes()
-close infile and outfile

---

3. decode.c
    a. Contains implementation of main() function

i. Implement as written on document with options: -h, -i, -o, -v

Pseudocode

-Declare the main loop
    -initiate variables for verbose output and the files
    -create while loop with all the options
        -if the option is i
            -set the input file
        -if the option is o
            -set the output file
        -if the option is v
            -set the verbose output to true
        -If the option is h
            -print the help option
        -set the default to the help option

    -set the infile to the in stream
        -if the file can't be opened
            -print the error message
            -clear the variables
            -return 1
    -set variables for the original message

    -if the verbose output is set
        -print the uncompressed file size, the compressed file size, space saving with formula

    -read in the header from the infile and verify that the magic number matches 0xBEEFBBAD
    -the permissions field of header indicates permissions that outfile must be set to using fchmod()
    -size of dump_tree array if given by tree_size attribute in the header. Read the dumped tree from infile into array that is tree_size long then reconstruct Huffman tree by reading the encoded symbols from input file
        -a stack of nodes will be used to reconstruct the tree
        -iterate over tree_dump from 0 to nbytes
        -if element of array is "L", then next element will be the symbol for the leaf node so create new node using node_create() and push node onto stack
        -if element of array is "I", then an interior node is reached, so pop the stack to get the right child of interior node and pop again to get the left child and join them
        -when there's one node left, Huffman tree is rebuilt

    -read the input file bit by bit (read_bit()) and traverse Huffman tree link by link.
        -start at room of tree, if value is 0 then walk left, else walk right
        -if you're at the lead node, then write the lead's node symbol to the outfile

-repeat until number of decoded symbols matches original file size gives by file_size field in header read from infile

-close infile

-close outfile

---

4. node.c
    a. Node Struct: Abstract Data Type that will help create the Huffman tree. It contains two pointer attributes for the left and right node, one for the symbol, and one for the frequency
    b. node_create function: the passed in parameters for symbol and frequency get set in a new node struct
    c. node_delete function: destructor for a node, the node gets freed and set to NULL
    d. node_join function: join the left and right child node that are parameters and the parent of the left child is the child of the right child, the parent node's frequency is the frequency of the child nodes added together
    e. node_print function: debug function to print the node

Pseudocode

-declare the struct for Node

-initialize a variable for the symbol

-initialize a variable for the frequency

-initialize a node pointer to the left node

-initialize a node pointer to the right node

-declare the node_create function with two parameters, one for the symbol and one for the frequency

-create a new node struct object and allocate space

-initialize the symbol and the frequency with the parameters

-initialize the left and right node pointers to NULL

-declare the node_delete function with one parameter that is a double pointer

-free the node that is passed in

-set the node pointer equal to NULL

-declare the node_join function with two parameters, a pointer to the left node, and the pointer to the right node

-create new node with symbol being "$" and frequency as the sum of the frequency of left and right child parameters

-set left child pointer to the left child parameter

-set right child pointer to the right child parameter

-declare the node_print function with one parameter that's the node

-print the node's content

5. pq.c
    a. PriorityQueue Struct: The struct will have an attribute for the capacity, a double pointer list attribute, and one for the top value
    b. pq_create function: set the capacity attribute and return a new queue object
    c. pq_delete function: delete each node of the priority queue, free it's allocated memory and set to NULL
    d. pq_empty function: return true if the queue is empty
    e. pq_full function: return true if the queue is full
    f. pq_size function: returns the number of items in the priority queue (non-0 in the list)
    g. enqueue function: adds a queue to the priority queue, use heap sort algorithm
    h. dequeue function: dequeues a node from the priority queue by taking out the highest priority node
    i. pq_print function: debugging function

Pseudocode

-declare the struct for PriorityQueue
    -initialize a variable for the capacity
    -initialize a variable for the list that represents the Queue
    -initialize a variable for the total amount in queue

-declare the pq_create function with one parameter representing the capacity
    -create a new PriorityQueue object and allocate space
    -initialize the capacity attribute with the parameter value
    -initialize the top value as 0

-declare the pq_delete function with one parameter that's a double pointer to the PriorityQueue
    -go through the array of the object and delete each node and set array to NULL
    -free space allocated for Queue and set it equal to NULL

-declare the pq_empty function with one parameter with the PriorityQueue
    -if the size of the queue is 0
        -return true
    -else return False

-declare the pq_full function with one parameter with the PriorityQueue
    -if the size of the queue is equal to the capacity
        -return true
    -else return false

-declare the pq_size function with one parameter with the PriorityQueue

-return the total attribute of the priority queue if it's not null

-include all heap sort helper functions from assignment 4 except build heap and heap sort

-declare the enqueue function with two parameters, the PriorityQueue and the node n
-check if queue is full, if so return false
-insert the given node into the priority queue array at the beginning of the queue
-use up heap to move the value to the correct spot of the heap queue
-initialize the top value of the array to the new top value
-return true

-declare the dequeue function with two parameters, the PriorityQueue and the node n
-check if the priority queue is empty, if so return false
-removes the node from the highest priority from the list and sets the double pointer parameter equal to this node
-call down heap to rearrange the heap queue
-return true

-declare the pq_print function with one parameter, the PriorityQueue
-since heap sort was used, use bubble sort on the mostly sorted array to print in order
-print the priority queue by iterating node by node

---

6. code.c
    a. Code struct: the struct for this object will have two attributes, one for the top value, and another that is a list that represents the maximum code size (a stack)
    b. code_init function: create a new code object on the stack and set the top to 0 and zero out the array of bits. Return the initialized code object
    c. code_size function: returns the size of the Code object
    d. code_empty function: checks if the code object is empty, false otherwise
    e. code_full function: returns true if the code is full, false otherwise. The maximum length of a code is 256 bits
    f. code_set_bit function: the bit at a particular index is set to 1, if it's out of range then return false
    g. code_clr_bit function: clear the bit at the particular index, if i is out of range, return false
    h. code_get_bit function: get the bit at the particular index, if out of range return false
    i. code_push_bit function: push a code into the stack, the value of the bit is given as a parameter. Return false if the code is full before pushing a bit, true otherwise
    j. code_pop_bit function: pop a bit off the code through the pointer parameter. Return false if the code is empty before popping a bit
    k. code_print function: print function of the stack to help debug

Pseudocode

-declare the struct for Code
    -initiate a variable for the top value
    -initiate a variable for the array of bits

-declare the code_init function with no parameters
    -create new code struct
    -initialize the top value to 0
    -initiate all the values in the array of bits to 0
    -return code struct

-declare the code_size function with one parameter representing the code object
    -return the top value of the code passed in

-declare the code_empty function with one parameter representing the code object
    -if the top value is equal to 0
        -return true
    -return false

-declare the code_full function with one parameter representing the code object
    -if the top value is equal to ALPHABET
        -return true
    -return false

-declare the code_set_bit function with two parameters representing the code object and an integer i
    -set a temporary variable equal to the location of the byte which is equal to i/256
    -set another variable equal to the modulus: i % 256 which gives the position of the bit inside the correct byte
    -if a position at this index doesn't exist, return false
    -set index i/64 equal to 1 shifted left by i%256bits[i/256] |= (1 << (i % 256));
    -return true

-declare the code_clr_bit function with two parameters representing the code object and an integer i
    -set a temporary variable equal to the location of the byte which is equal to i/256
    -set another variable equal to the modulus: i % 256 which gives the position of the bit inside the correct byte
    -if a position at this index doesn't exist, return false
    -set index i/64 equal to the opposite of (NOT) 1 shifted left by i%256 (bits[i/256] &= ~(1 << (i % 256));
    -return true

-declare the code_get_bit function with two parameters representing the code object and an integer i
    -set a temporary variable equal to the location of the byte which is equal to i/256

-set another variable equal to the modulus: i % 256 which gives the position of the bit inside the correct byte
-if a position at this index doesn't exist, return false
-val = (((bv->vector[i/256]) & (one<<i%256))>>(i%256));
-if val equals 1
-return true
-return false

-declare the code_push_bit function with two parameters representing the code object and an integer i
-call the set_bit function on the integer provided as a parameter
-if the call above returns true, success, otherwise, fail

-declare the code_pop_bit function with two parameters representing the code object and an integer i
-get the bit at the top of the stack
-call the clr_bit function on the top value provided as a parameter
-if the call above returns true, success, otherwise, fail

-declare the code_print function
-print each bit of the array

---

7. io.c
    a. read_bytes function: when the file needs to be read, this function is called. Repetitive calls to read() are called until desired number of bytes are read
    b. write_bytes function: similar to function above, except repetitive calls to write() is called
    c. read_bit function: read in a block of bytes into a buffer and dole out bits one at a time. When all the bits in the buffer are doled out, fill the buffer back up again with bytes from the infile
    d. write_code function: when the buffer is filled with bits, write the contents into the outfile
    e. flush_code function: if there are any bits that write_code() did not write into the file, this function will write out any leftover bits

Pseudocode

-declare the read_bytes function with three parameters, the infile, the buffer, and the nbytes int variable
-counter for the number of bytes read
-while the total bytes read is less than nbytes
-call read() and add the bits to the buffer that stores these read bytes
-add the number of bytes read to the extern variable to output stats
-return the total number of bytes read

-declare the write_bytes function with three parameters, the outfile, the budder, and the nbytes int variable

--counter for the number of bytes read
-while the total bytes read is less than nbytes
-call write() onto the outfile and add the bits to the buffer that stores these written bytes
-add the number of bytes read to the extern variable to output stats
-return the total number of bytes written

-declare the read_bit function with two parameters, one for the infile, and another for the bit
-create a buffer to store the values read from the infile
-create an offset to track the bit
-read_bytes from the infile and store in the buffer
-go through the buffer and increment the offset until the parameter for the correct bit is reached (use bit shifting to dole out bits until the desired value)
-return true if after this buffer there are still bits to be read
-if there are no more bits to be read, return false

-declare the write_code function with two parameters, the outfile and a code object pointer
-initiate a buffer to hold the bits that will be written to the outfile
-initiate a buffer for the index
-call code_get_bit to get the bit at each value of the code
-if it's equal to false
-write a 0 into the buffer
-else
-write a 1 into the buffer
-if the index has reached the end of the block, write everything into the outfile

-declare the flush_code function with one parameter, the outfile
-get to the correct byte, which is the index value divided by 8
-create a mask to get the leftover bits, which is one left shifted by the index mod eight minus 1
-if the mask isn't null
-add the left over bits to the buffer by using &=
-write the leftover bits into the outfile using the buffer

---

8. stack.c
    a. Stack Struct: this struct will contain a stack of nodes to reconstruct the Huffman tree. It has three attributes, one for the top value, one for the capacity, and another that is a list of node items
    b. stack_create function: a list is created with the size capacity and the top value will be initialized to a new Stack object
    c. stack_delete function: the list of nodes will be cleared and the stack will get deleted
    d. stack_empty function: returns true if the stack is empty
    e. stack_full function: returns true if the stack is full
    f. stack_size function: returns the number of nodes in the stack

g. stack_push function: a node gets pushed onto the stack and if this is successful return true
h. stack_pop function: a node gets popped off the stack, return false if the stack is empty before popping
i. stack_print function: debugging function to print the stack values

Pseudocode

-declare the struct for Stack
    -initiate an attribute for the top value
    -initiate an attribute for the capacity
    -initiate a list that represents the items in the stack

-declare the stack_create function with one parameter that represents the capacity
    -create a stack object and allocate space for it
    -set the stack's capacity attribute equal to the parameter value
    -set the top value to 0
    -allocate space for the list of nodes

-declare the stack_delete function with one parameter that represents the Stack object
    -create a for loop through the array attribute of the Stack
        -delete each node
    -free space for the stack
    -set stack equal to NULL

-declare the stack_empty function with one parameter that represents the Stack object
    -if the top value is 0
        -return true
    -else return false

-declare the stack_full function with one parameter that represents the Stack object
    -if the top value is equal to capacity
        -return true
    -else return false

-declare the stack_size function with one parameter that represents the Stack object
    -return top attribute of stack

-declare the stack_push function with two parameters that represent the Stack and the node to push
    -if the stack is full, return false
    -insert the node to the beginning of the array
    -increment the top attribute
    -return true

-declare the stack_pop function with two parameters, one for the Stack one for the node to pop

-if the stack is empty, return false
-remove the last element from the stack
-decrement the top attribute
-return true

-declare the stack_print function with one parameter representing the Stack
-go through the array of values for the Stack and print each node

---

9. huffman.c
    a. build_tree function: constructs the Huffman tree with a histogram
    b. build_codes function: populates the code table with each symbol
    c. dump_tree function: conducts post-order traversal of the Huffman tree with the root, writing to the outfile
    d. rebuild_tree function: reconstructs the Huffman tree with the post-order tree dump in the array tree_dump. The length in bytes of the tree_dump is given by nbytes parameter. Returns the root node
    e. delete_tree function: conduct post-order traversal to free each node in the tree and set the tree to NULL after freeing its memory

Pseudocode

-declare the build_tree function with one parameter representing the histogram of indices
-while the length of the histogram is greater than 1
-set the left node to the dequeued value of the histogram
-set the right node to the dequeued value of the histogram
-set the parent equal to joining the left and right nodes above
-enqueue the parent node into the histogram
-set the root equal to dequeue of the histogram
-return the root

-declare the build_codes function with two parameters, the root and the Code table
-create a code object
-if the current node is not NULL
-if the left and the right parts of the node are not null
-set the table value equal to the code object creates
-else
-push a 0 bit into c
-recursively call build on the left node and the table
-pop a bit from c

-push the bit 1 onto the code object

-recursively call the function on the right node with the table as another parameter
-pop a bit from c

-declare the dump_tree function with two parameters, the outfile and the root node
-if the root is not NULL
-recursively call the function on the left node
-recursively call the function on the right node
-if the left and right nodes aren't null
-write "L"
-write the node's symbol into the outfile
-else
-write "I" onto the outfile

-declare the rebuild_tree function with two parameters, one for the number of bytes and another with the tree_dump array
-declare variables for the root, left, and right
-initiate a stack struct for building the tree
-initiate a variable i to 0
-while i is less than the number of bytes in the tree
-if the symbol of the node is equal to "I"
-pop the tree for the left node to the dequeued value of the tree_dump
-pop the tree for the right node to the dequeued value of the tree_dump
-set the parent equal to joining the left and right nodes above
-push the parent node into the resulting stack
-else
-create a node and push it into the stack
-pop from the stack and set the value equal to the root
-return the root

-declare the delete_tree function with one parameter representing the root node
-if the root is not equal to NULL
-recursively call the function on the left node
-recursively call the function on the right node
-call the node_delete function
