Alex Asch
Assignment 5 Design
Description of Program
This program will begin by implementing a Huffman encoder and decoder. These are methods to compress and uncompress text. This encoding program will take an input file and compress it to an output file. To encode the file, first we will construct a histogram of the file, then make a Huffman tree with the histogram, then create a code table, then emit an encoding of the tree to a file. The decoder will read down the given input file and traverse the tree bit by bit thus decoding it.
Deliverables

- encode.c
    - This contains the implementation of the Huffman encoder
- decode.c
    - This contains the implementation of the Huffman decoder.
- defines.h
    - This declares the macros necessary for completing the assignment. This is given in the resources repository by Professor Long
- header.h
    - This contains the struct definition for a file header
    - This is given by Professor Long in the resources repository.
- node.h
    - This file contains the interface for the node ADT.
    - This is given by Professor Long in the resources repository
- pq.h
    - This contains the interface to the ADT queue.
    - This is given by Professor Long in the resources repository
- code.h
    - This file will contain the code ADT interface.
    - This is given by Professor Long in the resources repository
- io.h
    - This file will contain the I/O module interface.
    - Given in the resources repository by Professor Long.
- stack.h
    - This file will contain the stack ADT interface.
    - Given in the resources repository by Professor Long.
- huffman.h
    - This file will contain the Huffman coding module interface. This file will be provided.
    - This code is given in the resources repository by Professor Long.
- node.c

    - This file will contain the implementation of the node ADT/
- pq.c
    - This file will contain the implementation of the priority queue ADT.
- code.c
    - This file will contain the implementation of the code ADT.
- io.c
    - This file will contain the implementation of the input and output module.
- huffman.c
    - This file will have the implementation of the Huffman coding module interface.
- Makefile
    - Make the above program.
    - Links the above files
- README.md
    - A document in markdown syntax on the usage of the program.
- DESIGN.pdf
    - The pdf being viewed at the moment (this document).

Notes

- **Nodes**
    - This ADT is the implementation of the nodes in the Huffman tree.
    - Each node has a pointer to it's left child, a pointer to its right child, the symbol the node represents, and the frequency of that node.
    - The input of the symbol is a uint8_t as we take user input as raw bytes and not strings.
    - node_create()
        - This function allocates the memory for a node and initializes the symbol and frequency values to those passed to the function.
    - node_delete()
        - This deallocates a node's memory and sets its pointer to free.
    - node join()
        - This takes two child nodes and returns a pointer to the created parent node.
        - The parent node's left child will be left and it's right child will be right. The frequency will be the sum and the symbol will be $.
    - node_print()
        - debug function to make sure we have implemented nodes correctly.
- **Priority queue**
    - This is a queue in which each element has a priority so that elements with high priority are dequeued first.
    - To do this we make sure to encode the items in a specific order

  - To do this,we can implement a heapsort in a way that when an item is added, it obeys a min heap, and when an item is removed the heap is fixed to follow the constraints of a mine heap
  - pq_create()
    - Create an array of nodes of capacity size.
    - Make an int counter for the number of things in the array starting at zero.
  - pq_delete
    - Go through the array deleting nodes.
    - deallocate the priority queue
    - Make the pointer null.
  - pq_empty
    - If the int counter is zero then return true.
  - pq_full
    - If the int counter is capacity - 1 return true.
  - pq_size
    - return the int counter of capacity
  - enqueue
    - Put a node into the front of the queue, and then fix the heap, which means that it must follow the main heap structure, in which every parent has children with greater values than it. To do this, we add a node at the front of the queue, and ensure that the array is sorted in a way that follows the constraints of a min heap.
  - dequeue
    - This pops the first element off of the array and swaps it with the last, then calls heap fix to fix the heap.
    - It returns the popped element at the given pointer.
  - pq_print
    - print the tree in, this is for a debug
    - This will print the tree, which might be somewhat difficult but just print child by child with newline and spacing.
- **Codes**
  - A stack of bits while traversing the tree in order to create a code per symbol
  - This represents this.
  - code_init
    - Does not require a dynamic memory allocation, instead a new Code is created, setting top to zero and the array of bit.
  - code_size
    - returns exactly the number of bits pushed to code
  - code_empty
    - returns if the code_size is 0

    - code_full
        - returns true if the code is 256 bit
    - code_set_bit
        - Set the bit at index i in code to 1.
        - Clear the bit at index i in Code clearing it to 0
    - code_clr_bit
        - Clears the bit at the index to 0.
    - code_get_bit
        - Get the bit at i, only true if it is 1 at i.
    - code_push_bit
        - pushes a bit to code, given a bit
        - boolean for success
    - code_pop_bit
        - pops a bit off the code
        - boolean for success
    - code_print
        - check if bits are push/popped properly
- **I/O**
    - This takes input and output
    - Uses syscalls as we will be using bits.
    - int_read_bytes.
        - This function will be a wrapper to perform reads, looping until there are no more things to read, as we do not know if we get all the bytes at once
    - int_write_bytes
        - This is the same as above, but instead performs looped writes
    - read_bit
        - This takes a block of read bytes and pops bits out
    - write_code
        - This function will use a static buffer as in index and each bit the code c will be bufferein into the buffer
    - flush_code
        - This flushes leftover unbuffered bits
- **Stack**
    - This is a stack that stores nodes, same as asgn4 but with nodes
- **Huffman coding module**
    - Builds a Huffman tree using a computed histogram,
    - build_codes
        - populates a code table, building the code for each symbol in the huffman tree
        - The constructed codes are copied to the cod tab;e

        - dump_tree
            - Writes the traversal of the huffman tree from root to writing it to outfile
        - rebuild_tree
            - takes the dumped tree and returns the root of the tree
        - delete_tree
            - free the memory after sorting and the free itself

Pseudo

- **Nodes**
    - Node_create
        - Take the two values for symbol and freq and set it as the node's symbol and frequency
    - node_delete
        - Free the appropriate memory for a node.
        - Set the pointer to null.
    - node_join
        - Take two nodes
        - Use node create to make a new node of symbol $ and frequency which is sum of child frequencies
        - Set the left and right child to the given left and right nodes
    - node_print
        - print frequency and symbol of the given node, call itself recursively on it's children
- **Priority Queues**
    - pq_create
        - First allocate space for the priority queue.
        - The allocated space for the array of length capacity to be used.
        - Allocate one integer for the number of items in the queue.
    - pq_delete
        - Free the list
        - Then free pq.
        - Then set the pointer to null.
    - pq_full
        - Check if the top is size.
    - pq_empty
        - Check if end of queue is 0
    - pq_size
        - Return the value of the end of the queue.
    - enqueue
        - Check if the queue is full
            - IF so return false

        - Take a node to put into the queue.
        - Increment the top count by one.
        - Add the current node into the index pointed to by top
        - While the frequency of the current node is less than the frequency of its parent node(and a parent node exists), swap the two nodes.
        - Return true.
    - dequeue.
        - Check if the queue is empty
            - return false if it is
        - Swap the first and last node.
        - Put the last node into the pointer and decrement the top by one
        - Starting at the first node, check that it is less than its child nodes,
            - If it is not, swap them, and check again on the children with the parent being the newest swapped.
    - node_swap
        - Set a placeholder equal to one of the nodes passed.
        - Then set the first node equal to the second node
        - Then set the second node equal to the placeholder.
    - pq_print
        - Go over the min heap from first to child to child printing each child.
        - Print k, then print 2k and 2k+1 and call self recursively for each child while still in the appropriate array.
- **IO**
    - read_bites
        - if the index is zero get BLOCK bytes
            - set the end equal to the number of bytes obtained
            - use read bytes to read BLOCK bytes
        - return the bit at the index using same
        - if the index is equal to BLOCK reset index to zero and go again.
        - if the index is equal to end, and not block, set we are out of bytes
        - return false when we are out of bits to read.
    - read_bytes
        - take a number of bytes to read
        - Loop calls to read() until either we have read the given number of bits, or read returns 0, which is the end of file.
    - write_bytes
        - take a number of bytes to read
        - Loop calls to write to write from buffer until we have read either the given number of bytes.
    - write_codes

        - Take a code
        - loop through the code for the entire code
            - Get the bit off the top of the code
            - If the bit is one, add a bit of one into the current spot in the buffer
            - if the bit is zero add a zero into current spot in the buffer
            - increment spot in buffer by one
        - if the buffer is the size of BLOCK
            - write the buffer
    - flush_codes
        - share a buffer and indices with write codes
        - Write whatever is in the buffer, regardless if it is a full buffer
        - Zero the extra bits at the end of the last byte if the last byte is not full.
- **Stack**
    - stack_create
        - Allocate memory for a stack of size capacity
        - set the top and capacity variables
    - stack_delete
        - free the array in the stack
        - Free the stack.
        - set the pointer to null
    - stack_full
        - If the top is equal to capacity return true, else return false.
    - stack_empty
        - If the top is 0, return true, else false.
    - stack_push
        - If the stack is full, return false.
        - Else, add a node to the top of the stack and increment top.
    - stack_pop
        - If the stack is empty return false
        - Else, take the top node out, pass it through the pointer, and decrement the top.
    - stack_print
        - Loop through the array in stack and print all nodes.
- **Code**
    - code_init
        - No memory allocation is required since the size of the array is static.
        - Zero the array.
        - Initialize top
        - Return a pointer to the array.
    - code_size

        - Return the value of the top of the code.
    - code_empty
        - If the top is 0 return true, else return false.
    - code_full
        - If the top is equal to max code size * 8, return true, else false.
    - code_set_bit
        - If the index is greater than max code size return false
        - Set the byte a the index/8 to itself bitwise or 1 shifted right by index modulo 8.
    - code_clr_bit
        - If the index is greater than max code size return false
        - Set the byte a the index/8 to itself bitwise and not 1 left shifted by index modulo 8.
    - code_get_bit
        - If the index is greater than max code size return false.
        - If not return the current index/8 shifter left by index module 8 and 1.
    - code_push_bit
        - If the code is full return false
        - If the bit given is one, set the top bit.
        - If the bit given is 0 clear the top bit.
        - Increment the top by one.
    - code_pop_bit
        - If the code is empty, return false.
        - Get the bit at the top of the code
        - If it is one, set the bit pointer given to 1 and decrement the top.
        - If it is zero, set the bit pointer given to 0 and decrement the top.
    - code_print
        - Iterate through the array in code until the top and print each byte.
- **Huffman**
    - build_tree
        - Make a priority queue of size alphabet
        - For each number in the given histogram, create a node with a symbol of that index in the histogram, and the frequency value at that index. Enqueue that node.
        - While the priority queue has more than 1 element
            - dequeue two nodes
            - Join the nodes.
            - Enqueue the newly created node.
        - Return the last node left in the priority queue
    - build_code

        - Use a static code.
        - Iterate through a tree from the build tree.
        - If the node is an interior node, push a 0 to the given code and go to the left child. Then pop one off of the code, and push one to the code and then recursive down the right child. Pop one off code again
        - If the node is a leaf node
            - Put the current code into the code table at the index of current symbol
    - dump_tree
        - print a tree to the output from root.
        - Check if the root is NULL. if it is, stop here.
        - Call itself recursively on the given nodes left and right children.
        - If the node is a leaf
            - Print L and the symbol of the node
        - If the node is an interior node
            - Print I.
    - delete_tree
        - If the root is null stop here.
        - Call itself on the tree left and right child of the current node.
        - If it is a leaf node, delete the current node.
- **Encode**
    - Parse through the command line options
    - If -h print help and return
    - Open infile and outfile if given
    - Make an array for histogram
    - Read bytes, and for every byte increment that spot in histogram by one
    - Increment 0 and 255 in the histogram by one.
    - Build a tree using the histogram.
    - Make a code table for the tree using build_codes.
    - Count the number of unique characters by going through the code table and seeing where there are codes.
    - Make the header struct, set the magic number to magic, the tree size to unique characters *3 -1, the permissions to the permissions of the input and the size to the size of input.
    - Write the header out.
    - Dump the given tree to the output
    - Set the read pointer back to the front of the input
    - Read through all the bytes in the input, writing each given code for the bytes.
    - Set the permissions of the output to the permissions of the input.
    - Flush remaining codes

- **Decode**
    - Parse through the input.
    - Parse through the command line options
    - If -h print help and return
    - Open infile and outfile if given
    - Check if the input file has a valid header, by parsing the magic number.
    - Read the tree size, the permissions and the size in bytes of the tree.
    - Set the permissions of the outfile to the read permissions
    - Rebuild the tree from the next tree_size bytes of the input file
    - Go through the given tree and the codes that have been input
    - If we find a 0 go down the left of the tree if a 1 go down the right.
    - If we find a leaf node, print the symbol of the leaf and go back to the start of the tree.