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, bits is
    - 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