Tanya Gyanmote
cruz id - tgyanmot
1887664

Assignment 7

Purpose: The purpose of this program is to implement the Huffman encode and decode. With encode we compress a input file, and using decode it will read the given input file.

FILES

- Encode.c
  - This file will contain your implementation of the Huffman encoder.
- Decode.c
  - This file will contain your implementation of the Huffman decoder.
- Defines.h
  - This file will contain the macro definitions used throughout the assignment.
- Header.h
  - This will will contain the struct definition for a file header.
- Node.h
  - This file contains the node ADT interface
- Node.c
  - This file will contain your implementation of the node ADT.
- Pq.h
  - This file will contain the priority queue ADT interface.
- Pq.c
  - This file will contain your implementation of the priority queue ADT.
- Code.h
  - This file will contain the code ADT interface.
- Code.c
  - This file will contain your implementation of the code ADT.
- Io.h
  - This file will contain the I/O module interface
- Io.c
  - This file will contain your implementation of the I/O module.
- Stack.h
  - This file will contain the stack ADT interface.
- Stack.c
  - This file will contain your implementation of the stack ADT.
- Huffman.h
  - This file will contain the Huffman coding module interface.
- Huffman.c

    - This file will contain your implementation of the Huffman coding module interface.
- MakeFile
    - Complies the program
- Readme.pdf
    - describes how the program will run
- Design.pdf
    - describes the design of my program
- Encoder
    - Have a function for the print help
    - In the main
        - Have all the variables set to their default values
        - case I
            - Specificies the input file to encode using huffman
        - Case h
            - Print help message
            - Return -1
        - Case v
            - Set stats variable to true
        - Case o
            - Specificies the output file to write the compressed input to
        - Default case that prints the help message
            - Returns -1
        - Read the infile to compress
        - Build the tree from the histogram
        - Buld the code table from the tree
            - Using build code
            - Count the number of unique characters going from the code table
        - Set header
        - Write the header out
        - Dump the tree to the output file
        - Read data from input
            - Read bytes
            - Flush remaining code from buffer
        - If stats is true print out the compressed stats

- Decoder
    - Have a function for the print help
    - In the main
        - Have all the variables set to their default values

        - case I
            - Specificies the input file to decode using huffman
        - Case h
            - Print help message
            - Return -1
        - Case v
            - Set stats variable to true
        - Case o
            - Specificies the output file to write the decompressed input to
        - Default case that prints the help message
            - Returns -1
        - Rebuild the tree from the dump tree data
        - Read the remaining bits
            - Till it's equal to the original message size
            - Transverse until the leaf is reached
        - Print stats if it's true
- Node
    - Node_create
        - Allocating size to create a node
        - If the node is true
            - Set left and right to null
            - Set symbol and frequency to symbol and frequency passed from the parameters
        - Return the node
    - Node_delete
        - Free the node n
        - Set the node n to NULL
    - Node_join
        - Create a parent node, using node_create with the symbol $ and the frequency of left and right
        - Set parent left to left and parent right to right
        - Return the parent node
    - Node _print
        - If the node is true then print the node's left, right and symbol
    - Node_cmp
        - If the frequency of the node n is greater than the frequency of node m
            - Return true
            - If it's not return false
    - Node_print_sym
        - If iscntrl of symbol is false and is print of symbol is true

            - Print the symbol
        - If not then print the symbol with a unprintable symbol as 0x%02"PRIx8

- PriorityQueue
    - Pq_create
        - Allocating size to create a pq, if the pq is true
            - Set size and capacity to 0
                - Allocating size to create node of items
                    - If the items are false then free them
                    - Set them to NULL
        - Return q
    - Pq_delete
        - If the priority queue and items are true
            - Free the items
            - Free the priority queue
        - Return the priority queue
    - Pq_empty
        - If the size of the queue is 0
            - Return true
        - Return false if not
    - Pq_full
        - If the priority queue size and capacity are the same
            - Return true
            - If not return false
    - Pq_size
        - Return the size of the priority queue
    - Enqueue
        - Check if the pq is full
            - If it is return false
        - Have a while loop until the parentsis smaller than the node or index is 0
            - Swap added node to parents
    - Dequeue
        - Check if the pq is full
            - If it is return false
        - Using a helper function
            - To maintain heap
            - Pop the element
            - Decrease pointer
    - Pq_print
        - Have a loop that goes through the queue and prints each of the items

- Code
    - Code_init
        - Create code c
        - Set the top to 0
        - Have a for loop that goes to the max size
            - Set the bits to 0x0
        - Return c
    - Code_size
        - Return the top of code c
    - Code_empty
        - If the code c top is 0
            - Return true
        - Return false
    - Code_full
        - If the top od code c equals to ALPHABET
            - Return true
        - If not return false
    - Code_get_bit
        - If the bit of the index i is out of range
            - Return false
        - If the bit of the index i is equal to 0
            - Return false
        - If the bit of the index i is 1
            - Return true
    - Code_set_bit
        - Set the bit of the index i in code
        - Set it to 1 if i is out of range
            - Return false
    - Code_clr_bit
        - Bit at index i in code, clear it to 0
        - If i is not in the range
            - Return false
        - Else return true
    - Code_push_bit
        - If the code is full
            - Return false
        - If the bit is 1
            - Set the bit c and c top
        - If not clear the bit
        - Add one to node pointer

        - Return true
    - Code_pop_bit
        - If the code was empty
            - Then return false
        - Subtract 1 from top pointer
        - Return the popped bit
        - Clear the bit position
        - Return true
    - Code_print
        - For loop that goes till max_code_size
            - Print each of the bits
- I/O
    - Read_bytes
        - Set current read to read of infile, buf, and nbytes
        - Set total read to current read
        - Have a while loop that checks if current read isn't 0 and total read is less than n bytes
            - Set current read to read of fil, buf and total read, nbytes minus total read
            - Add current read to total read
    - Write_bytes
        - Set current read to write of outfile, buf, and nbytes
        - Set total read to current read
        - Have a while loop that checks if current read isn't 0 and total read is less than n bytes
            - Set current read to write of file, buf and total read, nbytes minus total read
            - Add current read to total read
    - Read_bit
        - Have a variable for buffer, index, end
        - Read the bytes and put them in a buffer
        - Check if all the bites are read
        - Find byte position in buffer
        - Find bit position in byte
        - Get the state of the byte
    - Write_code
        - Iterate through code c
            - Get the top bit
            - If code get bit is 1, true
                - Add bit of one to current spot of the buffer

                - If code get bit is 0, false
                    - Add 0 to the current spot in the buffer
                - If buffer equals the size of the block, write it
        - Flush_codes
            - Find the last position where there are bytes to flush
            - Flush the bytes using write bytes

- Stacks
    - Stack_create
        - Allocate memory for stack
        - If the stack is true
            - Set the top to 0
            - Set capacity to the capacity passed
            - Set the items to allocated memory
        - Return the stack
    - Stack_delete
        - If that stack is true
            - Free the items
            - Free the stack
            - Set the stack to null
    - Stack_empty
        - Set top to 0
    - Stack_full
        - If the top and capacity are the same
            - Return true
    - Stack_size
        - Return top
    - Stack_push
        - If the stack is full return false
        - inccrease top pointer
        - Set the top node to the push node
        - Return true
    - Stack_pop
        - If the stack is empty
            - Return false
        - Decrease the top pointer
        - Set node n stack s' items with the idex of the top
        - Return true
    - Stack_print
        - Have a loop that goes to the capacity

                - Using node print to print out every item
- Huffman
    - Build_tree
        - Create a priority queue
        - In a for loop
            - Create a node for every
            - Insert the node into the priority queue
            - Enqueue the priority queue and the node
            - dequeue 2 nodes from priority queue
            - Enqueue with new node
            - Get the last node of the priority queue
    - Build_code
        - Takes the node
        - Checks if the node is not null
        - Iterate through the tree
        - If the node, is the interior node then push 0, go to the left child
            - Pop one off the code
            - Push one to the code
            - Recursive down to right child
            - Pop one off again
            - If its the leaf node
                - Put current code into table at the index of the current symbol
    - Dump_tree
        - Chcek if node pointer is not null
        - Call left and right children recursivly
        - If nodes if a leaf
            - Print L and the symbol
        - If node is interior
            - Print I
    - Rebuild_tree
        - Rebuild huffman tree
            - Using sequence of characters
            - Using stack to rebuild tree
        - Push leaf node to stack
        - Pop two nodes
        - Push the combined popped nodes
        - Get the rebuilt huffman tree
    - Delete_tree
        - If root is null

        - Stop
    - Else call itself on the tree left child and right child of the current node
    - Else delete the node