Devan O'Boyle
CSE13s
Assignment 6: Huffman Coding

Design

Purpose: For this lab we will be compressing and decompressing files using a variety of various data structures by using Huffman's coding algorithm. Data compression can help a lot in saving storage capacity, speeding up file transfers, and decreasing the amount of network bandwidth.

The data structures used for this lab consist of: nodes, priority queues, codes, stacks, the Huffman tree, and input/output

Nodes: A node consists of a symbol and frequency. The symbol corresponds to any character that appears in the message, and the frequency is the number of times that that symbol appears in the message. Nodes can be joined together to form a parent node that contains pointers to it's left and right children. A node that isn't a parent is called a leaf node.

Pseudocode:
Node Join:

- takes in two nodes: left and right
- create a new node with '$' as the symbol, and the sum of the left and right frequencies for the frequency
- set the left node as the left child for the new node
- set the right node as the right child for the new node
- return the new node

Priority Queues: The priority queue will consist of nodes. When enqueueing a node, the priority queue will iterate through the current nodes on the queue and check if the node being enqueued has a higher frequency of each node. If it doesn't find a node that has a higher frequency than it, it will shift each node that it passes by one to make room for the current node that is being enqueued. Once it does find a node that it has a higher frequency than, that node will be inserted at its current position. If there are no nodes that the node being enqueued has more than, then the current node will be enqueued at the end. Dequeuing works mostly the same as it does with a regular queue. Since queues follow first in first out order, the node with the highest priority, a.k.a. the one at the beginning should be dequeued.

Pseudocode:
Enqueue:

- check if the queue is full, if so return false
- otherwise, create a temp variable that is equal to the queue tail
- while the temp is greater than the queue head
    - if the frequency of the item at temp - 1 is greater than the frequency of the current node
        - shift node at index temp-1 to index temp
        - decrement temp

        - otherwise break from the loop
    - increment the queue size
    - set the node at the temp index
    - increment the tail
    - return true to indicate a successful enqueue

Dequeue:
- check if the queue is empty, if so return false
- otherwise, decrement the queue size
- set the dequeued node to the parameter node
- increment the head
- return true to indicate a successful dequeue

Codes: The code works like a very simple stack, where each bit in can be pushed and popped onto the code. Like with the stack and the priority queue, there are functions that tell if the code is empty or full to ensure that the push and pop functions work properly. However, unlike the stack, there will be no memory allocation needed for the code.

Pseudocode:
Push Bit:
- check if the code stack is full, if so return false
- otherwise set the bit to at the top index of the code
- increment the top
- return true

Pop Bit:
- check if the code stack is empty, if so return false
- otherwise decrement the top
- set the bit at the top index to the parameter bit variable
- return true

Stack: The stack, like the priority queue, will store nodes. Otherwise, it is very similar to how the stack has been implemented in the previous assignments. There are empty and full functions to tell if an element can be popped or pushed onto the stack respectively. The stack follows a last in first out order so the last node on the stack should be the first one removed when the pop function is called. The pseudocode for push and pop is very similar to that of the code.

Huffman Tree: The Huffman Coding Module has a variety of functions since the Huffman tree is used multiple times throughout the program. The build tree function will create a priority queue and enqueue and dequeue nodes until the root node is all that's left. This function will be discussed more in depth in the encoding section. There is also the code table function which will populate the table with codes from the huffman tree by recursively traversing the huffman tree. This will also be discussed in further detail later.

Pseudocode:

Build Tree:

- takes in the histogram as a parameter
- create a priority queue
- iterate through the histogram
    - if there is a character with a frequency greater than 0 in the histogram
        - create a node for it
        - enqueue the node to the priority queue
- define nodes for the left, right, and parent
- while the queue contains more than one node
    - dequeue a node and set it to the left child variable
    - dequeue a node and set it to the right child variable
    - join the left and right nodes and set it to the parent node
    - enqueue the parent node onto the stack
- define a root node
- dequeue the last node and set it to the root node variable
- delete the queue
- return the root node

Build Codes:

- Takes in a root node and the code table
- Initialize the code
- check that the root isn't null
- check that the left and right children of the root are null
    - if so, then set the current code to the code table at the index of the current node's symbol
- otherwise
    - push 0 to the code stack to indicate that the left child is being traversed
    - call the build code function with the left root and code table as parameters
    - pop the 0 bit from the code stack after that finishes
    - push 1 to the code stack to indicate that the right child is being traversed
    - call the build code function with the right root and code table as parameters
    - pop the 1 bit from the code stack after that finishes

Rebuild tree:

- takes in the a dumped tree and the length of the dumped tree
- create a stack
- define the root node
- iterate through the dumped tree by using the length given in the parameter
- if the current element in the tree is 'L' then a leaf node has been reached
    - create a new node with the next element of the tree as the symbol and give it a frequency of 1
    - push the new node to the stack
    - increment the index of the loop
- otherwise if the current element in the tree is 'I' then a parent node has been reached

        - define a right and left node
        - pop a node from the stack and set it to the right node
        - pop a node from the stack and set it to the left node
        - join the left and right nodes and set them equal to a parent node
        - push the parent node onto the stack
- after the iteration has completed, pop the root from the stack
- delete the stack
- return the root

Delete Tree:
- takes in the root node
- check if the current node exists
    - if so, then check if it has a left node that isn't null
        - call the delete tree function on the left node
    - check if the right node isn't null
        - call the delete tree function on the right node
    - delete the current node

Input/Output: There will be functions to read bytes from the input, write bytes to the output as well as reading in bits from the input for the decoder. The read bytes function will read in every byte from the input until it reaches the end of the file and it should return the number of bytes read. The write bytes function is similar as it loops through the bytes set to be written and writes them to the output. The number of bytes written is returned. The read bits function reads in a block of bytes into a buffer and reads in each bit at a time from the bytes. The write code function writes the codes from the Huffman Tree to the outfile. Lastly, there is the flush codes that writes out the remaining bits leftover from write code. There are also external variables bytes_read and bytes_written that keep track of the total number of bytes read and written in the file which will be used later in the encode and decode functions to print out compression statistics.

Pseudocode:
Read Bytes:
- take in the input file descriptor, the buffer, and the number of bytes to be read
- initialize a bytes variables to a nonzero value to keep track of the number of bytes read with each read
- initialize a total variable to keep track of the total number of bytes read
- while bytes is greater than 0 and total does not equal the number of bytes to be read (from the parameter)
    - read from the file with the input file descriptor, the buffer, and the current total subtracted from the number of bytes to be read and set the number of bytes read to the bytes variable
    - add bytes to the global variable bytes_read
    - add bytes to the current total
- return the current total (should be equal to number of bytes in the parameter)

Write Bytes is pretty much the same except you write to the outfile rather than read from the infile, and increment bytes_written rather than bytes_read.
Read Bit:

- takes in the input file descriptor and a bit pointer
- initialize a static index variable and set to 0
- initialize a static buffer
- create an end of buffer variable
- if the bit index is not zero
    - call read bytes using the input file descriptor, the buffer, and the 4096 aka the size of the BLOCK, set return value to a bytes variable
    - if bytes isn't equal to BLOCK
        - end of buffer equals bytes times 8
- get the bit at the static index and set it to the bit pointer
- increment the index so that it wraps around the BLOCK time 8
- if the index is greater than equal to the end of buffer
    - return false
- otherwise return true

Write Bit:

- take in the output file descriptor and a code
- iterates through the code
    - if the bit at the current index in the code is 1
        - set the buffer to 1 at the index
    - if the bit at the current index in the code is 0
        - set the buffer to 0 at the index
    - increment the bit index
    - if the index equals 8 times the BLOCK
        - call write bytes to write the code to the output file
        - set the bit index to zero

Encoder:
The encoder will read input in either from standard input or an input file. The program will then count the number of occurrences of each different character from the input. These characters and frequencies will then be enqueued into a priority queue as nodes. Then, each node will be dequeued from the priority queue in pairs of two. The first node will be the left child and the second will be the right child. These nodes will be joined together, producing a parent node that will be enqueued onto the priority queue. Each node will be joined together until only one node is left in the priority queue: the root node of the Huffman tree. In order to construct the code table of the Huffman tree, a post-order traversal will be performed. A stack will be used to keep track of all of the codes. So starting from the root node, recurse down the left path first, if a leaf node is hit, then save the current code to the code table. If the current node isn't a leaf, meaning that it is one of the parent nodes, then push 0 to the code to signify that the left path is being traversed. After the left path has been traversed, 0 should be popped and 1 should be pushed

instead to signify that the right path is then being traversed. After the right path has been traversed 1 should also be popped. After this recursion has been completed, construct a header file to write to the outfile. Then perform a post-order traversal once again on the Huffman tree, except this time, recursively find each leaf node and write 'L' followed by the node's corresponding character to the outfile. Each parent node should have an 'I' written to the output with no character following it.

Pseudocode:
Encode:

- initialize variables: input file descriptor, output file descriptor, buffer, histogram, unique character counter, tree dump array, tree index, statistics boolean
- get opt loop
    - -i set input
    - -o set output
    - -h display help message
    - -v set bool variable to true to print statistics later
- increment the first and last values of the histogram to account so that there are always at least two elements
- while loop that calls read bytes and sets the return to a variable bytes
- while bytes isn't zero
    - iterate through bytes and increment the histogram value at the buffer index to increase the frequency of the corresponding ASCII character
- call lseek to reset the input index back to the beginning of the input file
- iterate through all of the ASCII values a.ka ALPHABET and count the number of unique characters there are in histogram
- create a code table
- call the build tree function and set it to new root node variable
- call the build code function
- construct the header and set all of its corresponding values
- write the header to the outfile
- perform the tree dump by making a separate function
- Tree Dump:
    - takes in the root node, tree dump buffer, index, and outfile
    - checks if the left node and right node of the current node are null
        - if so, set the buffer at the current index to 'L' to indicate a leaf node
        - then write the buffer at that index to the outfile
        - increment the index and set the buffer at that index to the node's symbol
        - then write the buffer at the index to the outfile
        - increment the buffer index
    - otherwise
        - call tree dump on the left child
        - call tree dump on the right child
        - set the buffer at the current index to 'I'
        - write the buffer at the index

        - increment the index
- after the tree dump
- create a temp variable and set to a nonzero value
- while temp is greater than zero
    - call read bytes and set its return value to temp
    - increment through temp and call write codes on each index of the buffer in code table
- then flush the remaining codes
- if the statistics variable is true, then print the stats of bytes_read and bytes_written
- delete the tree
- close the input and output files

Decoder:
The decoder will first read in a magic number that should correspond to the one defined in the program. If it doesn't, print an error message and exit the program. Otherwise, set the corresponding permissions. Next read in the size of the tree and create an array of the same size. Then read the input file into the array. Iterate through the array. If the character in the array is 'L', then you know you have hit a leaf and so the character after will be the character of that leaf. This character should be used to create a new node. If the character in the array is 'I', then you've hit a parent node meaning that the stack should be popped from twice to get the left and right children which should then be joined and pushed onto the stack as the parent node. After this, read from the input file one bit at a time. We will be traversing through the Huffman tree one last time. If the bit value is 0, go to the left child, otherwise traverse to the right child. If the child is a leaf, then write the leaf's corresponding character to the output. After a symbol has been written, go back to the root of the tree and read in the next bit to find a new leaf node. After the number of decoded symbols matches the file size that was read in in the beginning, then the decoder has completed its task and the message should be correctly outputted.

Pseudocode:
Decode:
- initialize variables: input file and output file descriptors, statistics boolean, bit variable
- get opt loop
    - -i set input
    - -o set output
    - -h display help message
    - -v set bool variable to true to print statistics later
- read the header from the input file
- if the header's magic value isn't equal to the magic value defined in the program
    - print an error message and quit
- create an array to store the dumped tree using the tree size provided by the header
- call rebuild tree using the tree size and tree dump array and set the return value to a root node variable
- traverse the tree reading each bit and check if it is a 1 or 0
    - if it is 0 traverse the left branch

        - if it is 1 traverse the right branch
        - if the left and right nodes are null
            - write out the symbol
            - reset the current node to the tree root node
- if the statistics variable is true, then print the stats of bytes_read and bytes_written
- delete the tree
- close the input and output files