Assignment 5 - Huffman Coding

Sean Carlyle

October 25 2021

1 Introduction

This is my design for Assignment 5, Huffman Coding. This program will contain two executable files called encode/decode. Encode will encode text into the Huffman encoded version which helps to compress the amount of data it takes up. The decoder will decode this Huffman encoded text back into normal text. The two binary files will take multiple user inputs to control which file to use and other settings. The -h command will display how to use the program for both binary files respectively. The -i file specifies the input file for both binaries. The -o file specifies the output for both files. The -v tells the program to print out the statistics of the program, statistics like space saved and file size. The program contains many helpful header files that are required to make the file run properly. The main purpose of this program is to save file space by encoding files.

2 encode.c

Encode is the binary file that will encode text into Huffman encoding. It takes an infile and converts it into the encoded text then writes the output. Firstly the encoder gets user input. Then it constructs a histogram. A histogram is an array of all possible symbols. We then go through the infile and count how many times a specific symbol is seen. We increment the histogram[symbol] by one each time we see it in the infile. We then pass the histogram into build_tree from huffman.c which constructs the Huffman tree contained in a root node. Then we call build_codes also from huffman.c to build a code table that has a code for each symbol in the Huffman tree. We then create a header that copies info (like permissions) from the infile along with our MAGIC number which tells us that the file was encoded by us and write it to the outfile. Then we call dump_tree from huffman.c to dump the contents of our Huffman tree in text format to the outfile. This way when we decode we will be able to rebuild the tree. We then walk through every character in the infile and write its corresponding code to the outfile. We print statistics if we need to and close the outfile/infile. This shows generally how our encode encodes text.

2.1 Encode Pseudocode

Get the user input
    Using getopt and booleans
    Open files with open() to get file descriptors
    If help is set to true by user, print help message and do not encode file
Construct a Histogram
    While the number of bytes read is equal to a block
        For number of bytes read
            Add one to unique symbols if histogram of buffer index is 0
            Add one to histogram of the buffer index
Construct a Huffman tree using build_tree
Build a code table using build_codes
Create a header from infile
    Copy info such as file permissions form infile to the header
    Write the header to the outfile
Dump the tree with dump_tree
Walk the input, write out each symbols code into the outfile.
    Use lseek to return to beginning of file
    Write corresponding code of each symbol in infile to outfile
Print statistics with verbose
Close the outfile/infile and delete the tree.

3 decode.c

Decode is another binary file that decodes Huffman encoding. It takes an infile and converts it into normal text then writes to output. Firstly it gets user input. Then gets the header from the infile and checks whether it was encoded by us. If the infile was encoded by us we can proceed to decode it. We then rebuild the Huffman tree by reading it from the infile then using rebuild_tree from huffman.c. This gives us the Huffman tree which we previously dumped into our outfile in encode.c. Now all we have to do is keep reading bits from the infile and using them to traverse the huffman tree. When we reach a leaf node we print out a symbol since that order of bits represents that symbol from our code table. Eventually when we've printed out the same number of bits as the original file we stop. At this point the file should be decoded!

3.1 Decode Pseudocode

Get the user input
    Using getopt and booleans
    Open files with open() to get file descriptors
    If help is set to true by user, print help message and do not decode file

Get header and exit if magic number is different
    read_bytes from infile to the size of the header
    If header magic != our magic number exit with error
Read dumped tree
Reconstruct the Huffman tree with rebuild_tree
Traverse down the tree and decode symbols to outfile
    While the bytes written are not equal to the file size
        Read a bit one by one
        If bit is 0 go down left of tree
        If bit is 1 go down right of tree
        Once you hit a leaf write its symbol to outfile
        Increment symbols written
        Return to root node
Print statistics if verbose
Close infile/outfile and delete tree

4 node.c

Nodes compose the Huffman trees which we will be using to encode/decode. A node contains a pointer to its left and right child, a symbol, and the frequency that the symbol appears. Node.c is our node ADT file and is very helpful. We need node_create to create the nodes. We need node_delete the delete the nodes. We need node_join so that we can add two children into a parent node. We also have node_print just for debugging.

4.1 Node Pseudocode

Make a structure called node with a pointer to left and right child node, a symbol, and the frequency in which that symbol appears.
node_create
    Allocate memory to node
    Set symbol as the symbol
    Set frequency as the number of times the symbol shows up
    Return node
node_delete
    Free node
    Set pointer to NULL
node_join
    Create a node with symbol "$"
    Make left child left
    Make right child right
    Make frequency sum of children frequency
    return node

node_print

Print symbol and frequency of node. If the node is not a leaf node call node_print on its children as well

5 pq.c

pq stands for priority queue and helps encode the text. Our priority queue really only cares about the smallest item. When we dequeue we just want to dequeue the smallest item so I decided to implement like a min heap. As item are enqueued they are kept in min heap order. This is so that once we pop, the first item in the array will be the smallest frequency node. We then swap the now empty first node with the last and resort the array to be min heap status again. We choose to do this rather than searching for the smallest with insertion sort because it is generally faster. pq_create helps create the pq. pq_delete deletes the priority queue. pq_empty checks if it is empty. pq_full checks if it is full. pq_size returns the size. Enqueue enqueues a node. Dequeue dequeues the smallest node. pq_print prints out the pq.

5.1 Priority Queue Pseudocode

pq_create

Construct priority queue and allocate memory
Set size to 0
Allocate memory to node array
return pq

pq_delete

Free pq
Free pq nodes
Set pq pointer to null

pq_empty

If pq node is empty
Return true
else
Return false

pq_full

If pq is full
Return true
else
Return false

pq_size

Return size of pq

enqueue

If pq is full

Return false
else
Enqueue selected element
Keep min heap ordered by swapping upward if any nodes above element enqueued are bigger
Return true on success
dequeue
If pq is empty
Return false
else
Dequeue selected item and replace it with last item in queue
Swap downwards until item just placed first in the array is correctly ordered by min heap properties
Return true on success
pq_print
Call node print for every node in queue

6 code.c

The code ADT will help create a code for each symbol in the Huffman tree. It does this using a bit vector. This code will help us create a code for each symbol in our text. We need code_init to initialize the code. code_size to check the size. code_empty to check if its empty. code_full to check if its full. code_set_bit to set a bit at a specific index. code_clr_bit to clear a bit at a specific index. code_get_bit to get a bit at a specific index. code_push_bit to push a bit into an index. code_pop_bit to pop a bit from its index. code_print is for debugging.

6.1 Code Pseudocode

Create a structure called code that holds a top and an array of bytes.
code_init
Make new code
set top to zero
zero all bits
return code
code_size
Return the code size
code_empty
If code is empty
Return true
else
Return false
code_full

If code is full
    Return true
else
    Return false
code_set_bit
    If index asked for is above ALPHABET return false
    Else use a bit mask to set bit at i / 8 to 1
code_clr_bit
    If index asked for is above ALPHABET return false
    Else use a bit mask to set bit at i / 8 to 0
code_get_bit
    If index asked for is above ALPHABET return false
    Else use a bit mask to get a bit at i / 8
    Return true if bit is 1 false is bit is 0
code_push_bit
    If code is full return false
    Else use a bit mask to shift bit to the right place
    Place it in the code
    Return true
code_pop_bit
    If code is empty return false
    Else if code_get_bit is 1 return a 1 in *bit, if 0 return a 0 in *bit
    Return true
code_print
    Print every byte in code

7 io.c

This is the input output file that will help read and write bytes from the infile or to the outfile. We need read_bytes to read characters from the infile. We use this many times to read headers, dumped trees, codes and characters. write_bytes writes bytes to the outfile. We need this when writing encoded text and also writing normal text. read_bit is for reading each individual bit from the infile. It helps us traverse the Huffman Tree to rebuild normal text from our coded one. write_code helps us write our encoded text from normal text. flush_code flushes the buffer that we use in write_code and prints everything that has been written in the buffer out. It ignored bits past the index we are currently writing on.

7.1 I/O Pseudocode

read_bytes
    Set a byte counter
    While byte counter is less than number of bytes to be read and file has not reached end

Keep reading to buffer
Return total bytes read

write_bytes
Set a byte counter
While byte counter is less than number of bytes to be written and buffer has not reached end
Keep writing from buffer
Return total bytes written

read_bit
Set a static tracker, static bit buffer, and end position
If tracker is 0, reach more bytes into the buffer
Place bit from buffer at tracker position in *bit and increase tracker
If tracker reaches end of buffer
Reset tracker to 0
Return if tracker has reached the end

write_code
For every index in code, get the bit
If the bit is 1 set it in the buffer as 1
If the bit is 0 set it in the buffer as 0
Increase the index
Once buffer is full flush_code to write it to outfile

flush_code
If index is not 0
If index is at the end of a byte set counter to index / 8
If index is in the middle of a byte set counter to (index / 8) + 1
If in the middle of a byte clear the part of byte past index
Write from buffer to outfile a counter amount

8 stack.c

We use the stack in our decoder to reconstruct the Huffman tree. stack_create creates the stack. stack_delete deletes the stack. stack_empty checks if the stack if empty. stack_full checks if the stack is full. stack_size returns the size of stack. stack_push pushes an item to the stack. stack_pop grabs an item from the stack. stack_print just prints the stack visually.

8.1 Stack Pseudocode

stack_create
Accepts the wanted capacity of stack as uint32_t
Allocate memory to stack
If allocating worked properly
Set stack top of stack to 0

Set stack capacity to wanted capacity
Dynamically allocate array of items
If items allocated incorrectly
Free stack and set to null
Return stack

stack_delete
Accepts the stack pointer address as a parameter
If both stack and stack items exist
Free them both
Set stack pointer to null
Return

stack_empty
Accepts stack pointer as parameter
If top of stack is index 0
Return true
Else
Return false

stack_full
Accepts stack pointer as parameter
If top of stacks index is above capacity
Return true
Else
Return false

stack_size
Accepts stack pointer as parameter
Return top of stack index minus one

stack_push
Accepts stack and a uint32_t item
If stack is full
Return false
Else
Put x on top of stack
Make top of stack plus one
Return true

stack_pop
Accepts stack and a uint32_t item
If stack is empty
Return false
Else
Put top of stack value in x
Make top of stack minus one
Return true

stack_print
Node print all items in the stack

9 huffman.c

The huffman coding modules it the actual file that helps us build Huffman trees. It has commands to build, dump, rebuild, and delete the tree. It additionally helps us build our code table. build_tree builds the Huffman tree by inserting nodes into a pq for every histogram symbol. Then while pq is less than 1 it dequeues two nodes, joins them then requeues their parent. Once there is one node left the Huffman tree is complete. The one node left is the root node. build_codes builds the code table by traversing the Huffman tree. It pushes 1's and 0's depending on going left and right in the tree and when it reaches a leaf node that combination of 1's and 0's to get there is the symbol of that nodes code. dump_tree helps write the tree to the outfile. It traverses the tree and writes L if it reaches a leaf node then the symbol and I if it is at an interior node. rebuild_tree rebuilds the tree from the dump_tree output. It traverses the tree and when it hits an L it stores the symbol next to it in a stack. When it hits an I it knows to join the the last two items in the stack. When there is one item left, the Huffman tree has been completed. delete_tree deletes all the nodes from a root node. It goes down and deleted the leaves of the tree. On the way back it up it deletes the nodes that were previously interior nodes.

9.1 Huffman Pseudocode

build_tree
Create a priority queue
Insert node into pq for every histogram symbol
While pq is greater than 1
Dequeue two nodes, join them and enqueue their parent
The last node left in pq is the root node of huffman tree
Dequeue it and delete pq then return the root node
build_codes
If root node exists
If leaf node add coded symbol to table
Else push bit 0 and traverse down left
On way back up pop the 0 and push 1 and traverse right
On way back up pop 1
dump_tree
If root exists
call dump_tree on right
call dump_tree on left
If you hit a leaf node write L and then symbol
If you hit a interior node write I to outfile

rebuild_tree

For each byte in tree

If tree symbol is L push a node with symbol tree[i+1]

Skip the next index

Else if tree symbol is I

Pop right node and left node then join them

Push their parent node

Last node in stack is the root node of Huffman tree

delete_tree

Traverse through huffman tree deleting the leaves

Do this by calling delete on left and right if not at a leaf

On the way back up from the leaves delete the newly formed leaf nodes

10 Layout of program

The layout of my program is the two files with main: encode and decode that use the many header files to abstract much of what they are doing. Code, node, pq, and stack are all the headers for our ADT's while huffman and io are for abstracting encode/decode functions. This is the structure of my program.