Ernani Raffo

Design Document

Huffman Coding

Using Huffman Coding, this program is able to encode input files into a compressed form, and also decode its contents back to what it originally was.

Nodes

In the Node abstract interface, nodes of a Huffman tree are created and can also be joined to create a parent node. To create a node, I simply allocate enough memory for a node “n”, and set its symbol as an unsigned 8 bit integer and its frequency as an unsigned 64 bit integer which are passed through the constructor function. The destructor of the node frees the contents of the node “n” and makes the pointer to the node equal to “NULL” so that it cannot be accessed any longer. In the structure of a node, it contains a left child and right child of type “Node”, the node’s symbol, and its frequency. To join two nodes together, I simply initialize a pointer to the variable called “parent” with the node created out of the frequency of its left child and its right child. I set the symbol of the parent node in this case to be “$”. Inside this part of the program, I also make sure to set the parent’s left child and right child to be the children passed to the “node_join()” function. Lastly, a printing function is included which prints the passed node’s symbol and its frequency.

Pseudo Code for Node API

Constructor:

- allocate memory for node
- node symbol is set to symbol
- node frequency is set to frequency
- node’s left child is NULL
- node’s right child is NULL

Destructor:

- free node
- set node to null

Node join:

- make parent symbol ‘$”
- create node “parent” with parent symbol and frequency of its left child node and right child node
- parents left and child nodes are set to the left and right nodes passed to function

Priority Queue

The priority queue works similar to a queue, except that elements with the highest priority are dequeued. In this program, the element with the highest priority is the element in which has the least frequency. The priority queue abstract data type consists of a head, tail, capacity, and a queue of type Node pointer. When it is created, it needs to be allocated on the heap by using malloc, and if it is allocated properly, then the head and tail are set to 0, the capacity to the argument passed to the constructor function, and the queue is also allocated on the heap with size of Node pointer, with “capacity” amount of elements. If that is allocated properly, then the priority queue is returned, otherwise the priority queue is freed. The destructor of a priority queue frees its queue array and the queue itself. The interface of the priority queue contains functions that return whether or not the queue is empty or full, which is helpful for enqueuing and dequeuing elements off the queue. To enqueue an element the priority queue cannot be full, and to dequeue an element, the priority queue better not be empty. In my program, I implemented the priority queue as a min heap. When enqueuing, I fix the heap starting from the enqueued node and checking if its parent nodes are bigger than itself. To check the values in the priority queue, I use 1 based indexing instead of 0 based indexing to make the math work well. The parent node index is set to the variable “parent_node”, and the child node index is set to the where the tail of the queue points to and assigned to the variable “child_node”. While the parent_node index is greater than the index of the head (which is always 0), the values of the child node and the parent node are swapped if the parent node value

is greater than the child node value. Before the next iteration of the while loop, the child node index is set to the parent node index and the parent node is set to the next parent. For dequeuing a node, my algorithm first swaps the first and last elements in the queue, dequeues the node and passes it back to the double pointer “n”, shrinks the queue, and calls a helper function called “fix_heap()”. In this function, the swapping works the same as enqueuing, except that the algorithm starts at the element at the head (the element that was initially at the tail), and works its way down until it is in the right position. In this function, the parent node is swapped with its child with minimum frequency. This is calculated through another helper function called “min_child()” which simply returns either the parent’s left child index or right child index depending on which has the least frequency. The pseudo-code below shows the design of the priority queue and its functions.

Pseudo Code for Priority Queue API

Structure Priority Queue:

- head
- tail
- capacity
- queue array of type Node pointer

Constructor:

- allocate memory for priority queue
- if allocated properly
    - head = tail = 0
    - capacity = capacity given
    - allocated memory for queue array
    - if allocated proproelly
        - return the created priority queue
    - otherwise free the priority queue

- return null priority queue

Destructor:

- if allocated properly
    - free the queue array
    - free the priority queue data type
    - point the queue data type to NULL

Empty:

- head == tail?

Full:

- tail == capacity?

Size of priority queue:

- return the tail

Min child helper function (1 based indexing):

- return the child index of the child with the least frequency

Fix heap helper function for dequeuing (1 based indexing):

- only function if the queue has more than one element
    - found = false
    - parent node index is queue's head index + 1 to account for 1 based indexing
    - child node calls min child function
    - while not found
        - if parent node frequency > child node frequency
            - swap them
        - else
            - found = true
            - break out of while loop
        - parent node index = child node index

        - if the new parent node index is truly a parent node index
            - get the new min child
        - else
            - found = true

Dequeue:

- if queue is not empty
    - swap first and last elements
    - dequeue node and pass it back to function argument double pointer “n”
    - decrease size of the queue
    - call fix heap
    - return true
- return false

Enqueue:

- if queue is not full
    - enqueue the node
    - point the tail of the queue to the next empty slot in the queue
        - if size of queue > 1
            - parent node index = floor of tail / 2
            - child node index = tail of queue
            - while parent node > head of queue (0)
                - swap parent and child if parent frequency > child frequency
                - else break out of while loop
                - child node index = parent node index
                - parent node = floor of parent node index / 2
    - return true for success
- return false because could not enqueue

Stack

The Stack abstract data type works with nodes in its array of items. Similarly to the Code abstract data type, the structure of a stack contains a variable called “top” which represents the index of the next empty slot in the stack, a variable “capacity” that serves as the maximum number of items that can be pushed to the stack, and an array called “items” which represents the stack of items. The accessor functions in this interface return the size of the stack, whether the stack is empty or full. The manipulator functions are able to push items and pop items on and off the stack. A printing function is also included in order to be able to visualize what the stack looks like. To print the items of the stack, a call to the “node_print()” function is necessary since the Node data type is an opaque data type and the items inside the stack are of type Node.

Pseudo Code for Stack API

- Structure Stack:
    - top (index of next empty slot)
    - capacity (number of items)
    - array “items” of type Node pointer
- Constructor:
    - allocate using malloc
    - if allocated properly
        - stack top = 0
        - stack capacity = given capacity by caller
        - stack items = allocate using calloc
        - if stack items is not allocated properly

                - free stack
                - set stack to null
        - return stack
- Destructor:
    - if allocated properly
        - free the items array
        - free the stack data type
        - point the stack data type to NULL
    - return
- Size:
    - return top of stack
- Empty:
    - top == 0?
- Full:
    - top == capacity?
- Pushing:
    - if stack is full
        - return false because cannot push
    - otherwise push given node to the top
    - point the top of stack higher by 1
    - return true because successfully added
- Popping:
    - if stack is empty
        - cannot pop anything so return false

- otherwise go to the last item in stack (stack top - 1)
- dereference variable given and point it to the popped node
- return true because successfully popped

Code

The Code abstract data type deals with a stack of bits. Specifically, the stack (essentially an array) has a size of 32 elements, with each element representing one byte. To initialize a code, all I do is set a pointer to a variable called “code” with the type Code, which is defined in code’s header file “code.h”. I then set the code’s “top” to 0. The top is just the index at which the next empty slot in the stack is at. The next step in order to fully initialize a code is to zero out each index in the array of bits. What this is doing is make each byte in the array of bits equal to 0. The code is then returned. The code abstract data type has implemented functions that return the size of the code (which is the number indicated by “top”), if the code stack is empty or not or if it is full or not. Three similar functions in the code abstract data type are “code_set_bit()”, “code_clr_bit()”, and “code_get_bit()”. They all use bit vectors in order to do their specific function. To set a bit at a position “i”, I look for which byte it is in by dividing “i” by 8. Once I know which byte it corresponds to, I go into that byte and perform a left logical shift on the value “0x1” by (“i” modulus 8), which is the index in the byte, and perform a bitwise “OR” operation with the byte that it is in. This sets a bit to a 1 in a specific byte. For clearing a bit, the same is done except the value that is logically shifted goes through the bitwise “NOT” operator first, and a bitwise “AND” is performed between the two values. To get a bit, I go into the byte in which “i” is located, and I return true if that is equal to “0x1” or false otherwise. To actually check if that value is equal to 1, I perform a right logical shift to the byte by where the bit is located (“i” modulus 8) and I perform a bitwise “AND” with 0x1. This yields a 0x1 if the value at the bit location was 1, and a 0x0 if it was a 0. It is important to note that for these three functions, false is returned if the variable “i” happens to be bigger than 256, because then it is out of range of the ASCII language. Pushing and popping codes is pretty simple. To push a code onto the

stack of bits, I first check if the code is full and if it is not, then I check if the passed bit is equal to “0x1” or “0x0”. If it is 1, then I call the “code_set_bit()” function and tell it to set the bit at the top of the stack. Otherwise, the bit is 0, so I call the “code_clr_bit()” function to make the bit 0 at where the top of the stack is. The reason why I want to clear the bit is because there might be a value of 1 at the top of the stack that could have been “popped” before by the program. Speaking of popping bits, “code_pop_bit()” first checks if the code is empty, and if it is not, it decrements the “top” value by one in order to point at which bit to pop. In reality, what I am doing is calling the function “code_get_bit()” which returns true if the bit that it “gets” is 1 and false if it is 0. If the function yields true, then I assign the pointer to the variable “bit” as 0x1, otherwise it is set to 0x0. If the code was empty, then false is returned because there is nothing to pop. Lastly, the print function for the code simply prints each byte’s value in the bits array.

Pseudo Code for Code API

- Initializer:
    - code top = 0
    - zero out array “bits”
- Size:
    - return top of code
- Empty:
    - top == 0?
- Full:
    - top == ALPHABET?
- Set a bit:
    - if index is in range
        - left shift bit into right position and OR with byte

        - return true for success
    - return false because could not set bit
- Clear a bit:
    - if index is in range
        - left shift complement of 0x1 to right position and AND with byte
        - return true for success
    - return false because could not clear the bit
- Get a bit:
    - if index is in range
        - right shift byte by bit position
        - perform bitwise AND with 0x1
        - return equality between the bit and 0x01
    - return false because could not get bit
- Pushing:
    - if code is not full
        - if bit is 1
            - set the bit at code's top
            - increment top of code
        - else if bit is 0
            - clear the bit at code's top
            - increment top
        - return true for success
    - return false because could not push bit

- Popping:
    - if code is not empty
        - decrement top by 1
        - if bit is 1 at top
            - dereference bit and assign it to the value 1
            - return true for success
        - else the bit has to be 0 so pass that to variable bit
        - return true for success
    - return false

Encoding

The program encoder takes many steps in order to compress a file. Inside its main function, it first loops through the command line options to check if an infile or outfile was specified along with the verbose statistics option. If the user would like to understand how to use the program or they simply entered a wrong option or argument, then the synopsis message is printed and the program ends. Once that is figured out, then the program moves on to its initial steps in encoding. The first step is to read through the infile and construct a histogram which contains the count of each symbol in the infile. A buffer of size of BLOCK (4096) is initialized with each index representing a byte, a result variable which tracks the amount of bytes read is initialized, and a histogram of size ALPHABET (256) is also initialized. In a while loop, the function “read_bytes” is called while the amount of bytes read is greater than 0, meaning that it reads bytes until there are no more to read. Each time it is called, the histogram’s value at each symbol read is incremented by 1. To rest of the program will deal with building trees and using the histogram to do so, and if the program is given input of size 0 bytes, the histogram would have all non zero entries and that would not be possible. In order to account for this, the value at element 0 and 255 in the histogram is incremented by 1 so that the program is able to make a tree no matter the input.

Next, the huffman tree is built by calling its respective function which returns a pointer to the root of the tree. In my program, I named this node pointer “root” for simplicity. Then, a code table “code_table” is initialized with size ALPHABET, where each value at its indices are of type Code. The codes are put into it by calling the function “build_codes” and passing the root of the tree “root” and the code table as its arguments. The next step in encoding is to construct a header for the outfile. To do so, the function “fstat()” is called, where its arguments are the infile file descriptor and the address of the stat structure “stats”. I named the variable of type struct stat “stats” because it can access various types of stats within its structure, so a plural version of “stat” makes more sense. The program uses “stats.st_size” and “stats.st_mode” to get the file size and the permissions of the infile, assigning it to “header.file_size” and “header.permissions”, respectively. The given magic number in “defines.h” is assigned to “header.magic”, and “header.tree_size” is the result of the equation that calculates the tree size of the Huffman tree. The tree size depends on the amount of unique symbols in the infile, so that is calculated beforehand through a for loop. The header is then written to the outfile followed by the dumped tree, which is written to the outfile using the “dump_tree()” function. The last step in encoding the infile is to write out the code that is used by the program decoder to ultimately decode a compressed file. In order to write the code, the function “lseek()” is used to rewind to the beginning of the infile. Each byte in the file is then read into a buffer “c_buffer” of size BLOCK until there are no more bytes to read. For each byte in the buffer starting from the beginning byte until the last byte read, the code is written for that specific byte. This is done by calling the function “write_code()”, and is followed by the function “flush_codes()” outside of its loop in order to make sure any remaining code is written to the outfile. Finally, the encoding process is finished, but the program is not quite finished. I call “fstat()” at the end of it all, but on the outfile this time so that I can compare the uncompressed file size to the compressed file size. I only print these comparisons if the verbose command line option was enabled by the user. I change both the stats of the compressed file size and uncompressed file size to be of type “double” so that the statistics work well with the space saving percentage. The constructed Huffman tree is deleted from the root to free the memory, and the file descriptors of the infile and outfile are closed. The program is terminated and exists successfully.

Pseudo Code for Encoder

- default file descriptor is stdin file descriptor
- default file descriptor is stdout file descriptor
- loop through command line options (change file descriptors if specified, enable statistics if specified, print help message if specified or error)
- read through infile to construct histogram:
    - create buffer of size BLOCK
    - create histogram of size ALPHABET
    - while reading bytes from infile
        - fill histogram with count of symbols
- increment count of element 0 and 255 in histogram
- construct a huffman tree:
    - Node pointer root = root given after building the tree
- construct code table:
    - code table of size ALPHABET
    - call build codes function to fill code table
- construct a header:
    - call fstat() on infile file descriptor
    - set each field in header appropriately
    - change permissions of the outfile to match the infile permissions
- write constructed header to outfile
- dump the tree to the outfile
- write the code to the outfile:
    - create buffer of size BLOCK
    - use lseek() to rewind to beginning of infile

        - while reading bytes from infile into buffer
            - write code for each symbol in buffer
- flush any remaining codes
- get stats for the outfile:
    - if statistics enabled:
        - print uncompressed file size
        - print compressed file size
        - print space saving (100 * (1 - (compressed file size / uncompressed file size)))
- delete the tree to free memory
- close infile file descriptor
- close outfile file descriptor

Decoding

In the beginning portion of its main function, the decoding program is the same as the encoder in dealing with command line options. The algorithm then starts by first reading a header from the infile. If the magic number does not equal to the same magic number given by “defines.h”, this means that the infile given was not encoded by my encoder, so an error message is printed informing the user that the file does not have the right magic number, and the program is terminated. Otherwise, the outfile permissions are changed to the permissions in the infile header, and the program can reconstruct the Huffman tree. Firstly, an array called “tree_dump” of size MAX_TREE_SIZE is initialized, the encoded bytes representing the dumped tree in the infile is read into the array, and a Node pointer called “root” is assigned to the root of the rebuilt tree. Now, the program walks the Huffman tree based on the last bytes of the infile and outputs the decoded symbols to the outfile. In order to walk the tree, I assign a Node pointer variable called “n” to represent the current node in the tree. This algorithm of walking the tree ends when the number of symbols written to the outfile matches the file size given in the header portion of the infile, specifically, at “header.file_size”, so I initialize a variable called “symbols” to track how many symbols have been written. The while loop in which checks this condition is made of two nested

conditional statements. The first statement is an if-statement that checks if the current node that we are at is a leaf node. If it is a leaf node, then its symbol is written, the node “n” is reset to the root of the tree, and symbols are incremented by 1. The second statement is an else statement, which would occur if the current node is a parent node. Since it is a parent node, we walk the tree either left or right based on the code, and set the node “n” to either its left child or right if the next bit in the code is a 0 or a 1, respectively. By doing this repeatedly until the amount of symbols written matches the file size target, the compressed file is decoded back to its original form. The statistics are calculated and printed like the encoder in terms of the verbose option and the format of the variables, however, the calculation is slightly different. The pseudo-code below shows how it is done. Lastly, the rebuilt tree is sent to its destructor, and the file descriptors are closed.

Pseudo Code for Decoding API

- default file descriptor is stdin file descriptor
- default file descriptor is stdout file descriptor
- loop through command line options (change file descriptors if specified, enable statistics if specified, print help message if specified or error)
- read in header from infile
- if the header does not equal MAGIC
    - print error message and exit
- change permissions of outfile
- reconstruct the Huffman tree:
    - create array “tree_dump” of size MAX_TREE_SIZE
    - read the tree dump into array
    - set the root of the tree to the result of rebuilding the tree
- decode the dumped tree:
    - Node pointer n = root of tree

    - variable “symbols” to track amount of symbols written
    - while symbols < file size given by header
        - if n is a leaf node
            - write n’s symbol
            - reset n to the root
            - increment symbols read
        - otherwise it is a parent
            - read a bit from the code
            - if bit is 0
                - set n to its left node
            - else if bit is 1
                - set n to its right node
- get stats for the given compressed file:
    - if statistics enabled:
        - print compressed file size
        - print decompressed file size
        - print space saving (100 * (1 - (compressed file size / decompressed file size)))
- delete tree starting from root to free memory
- close infile file descriptor
- close outfile file descriptor

IO Module

The IO module contains four functions in which read bytes from an infile, write bytes to an outfile, reads bits one at a time, writes a code to an outfile. and flushes those codes if necessary. The reading and writing bytes functions work very similarly to one another, as they simply loop through either read() or write() and fill a specific buffer with bytes or write out the contents of a buffer based on either a read or a write being executed. The read bit function functions the same as “read_bytes()”, except that it

returns a bit each time that it is called. In this function, I have a static buffer of size BLOCK, an index to track which bit we are at, and a variable that tracks the amount of bytes read when calling the “read_bytes()” function. For the last two functions in the interface, they share a static buffer of size BLOCK, and a static index that tracks which bit to write. These two functions are the “write_code” and “flush_codes" functions. The writing code function’s main job is to write a given code into the shared static buffer, and it only writes to the outfile once that buffer is completely full. The flushing code function’s main job simply zeroes out any extra bits that may be in the buffer still and writes the contents of the buffer to the outfile.

Pseudo Code for IO Module

Read bytes:

- while read() reads more than 0 bytes out of the buffer
    - bytes read is increment by that value
- return amount of bytes read

Write bytes:

- while write() writes more than 0 bytes out of the buffer
    - bytes written is increment by that value
- return amount of bytes read

Read bit:

- static buffer of bytes
- static index of bit position
- static amount of bytes
- if index is 0
    - read BLOCK bytes into “read_buffer”
- dereference bit variable and assign it to the bit at position “index”:
    - this requires bit vector operations
- increment the index

- if the index == maximum amount of bits (BLOCK * 8)
    - reset index to 0
- return true if there still bits to read and false if not

Write codes:

- iterate through the code given
    - fill buffer with bits by using “code_get_bit()” function:
        - set the bit at the index of “write_buffer”if bit is 1
        - else clear the bit at the index of the write buffer
    - increment the bit index
    - if index == maximum bit index
        - write the contents of the buffer
        - reset index to 0

Flush codes:

- zero out extra bits in the last byte if needed
- determine how many bytes to flush
    - increment by 1 if the bits are not aligned cleanly in a byte
- flush the bytes by calling “write_bytes()”

Huffman Module

The Huffman module in this program is responsible for building a Huffman tree, populating a code table which is later used in the encoder to write out the full code to the outfile, dumping the tree to the outfile so that the decoder can then rebuild it (also a responsibility in the Huffman module), and the destructor for the trees that are built or rebuilt. The design choices that went into writing these functions were all very similar as they deal with root nodes, parent nodes, as well as left and right nodes. One specific choice that I ended up making in order to build codes is that I created a helper function which performs the post-order traversal recursive algorithm while the actual function that is called in the encoder

only initializes a code of type Code. The Pseudo-Code below contains the specifics of how each algorithm functions.

Pseudo Code for Huffman Module

Build tree:

- create priority queue
- for each symbol in histogram that occurs more than once
    - create a node with its frequency
    - enqueue node into queue
- while size of priority queue is greater than 1
    - dequeue left node
    - dequeue right node
    - make a parent node
    - enqueue parent node
- dequeue last node (root of tree)
- delete the priority queue to free memory

Helper function for “build_codes()”:

- if root is not NULL
    - if it is a leaf node
        - put the current code at index of node symbol in code table
    - else it is a parent node
        - push a 0 to the current code indicating going left
        - resurse left
        - pop the bit off the code
        - push a 1 to the current code indicating going right
        - resurse right
        - pop the bit off the code

Building codes:

- initialize code
- call helper function and pass the initialized code

Dump tree:

- if root exists
    - recurse left
    - recurse right
    - if it is a leaf
        - write ‘L’ to outfile
        - write the node’s symbol to outfile
    - else it is a parent node
        - write ‘I’ to outfile to indicate interior node

Rebuild tree:

- create a stack of size BLOCK
- iterate over tree array starting from 0 until number of bytes given
    - if current symbol is ‘L’
        - create a leaf node with symbol being the current index + 1 position in the tree array
        - push the created leaf node to the stack
        - ignore next symbol by incrementing index
    - else if the symbol is ‘I’
        - pop the right child off the stack
        - pop the left child off the stack
        - create a parent node by joining them
        - push parent onto stack
- pop the last node off the stack and assign it to node pointer in which to return

- delete the stack
- return the root

Delete tree:

- if the root is valid
    - if we reach a leaf node
        - delete the node
    - else it is a parent node
        - recurse left
        - recurse right
        - delete the parent node

Credit

- The constructor and destructor function for the Priority Queue interface was influenced by Professor Long's Fall 2021 instructional slides on Stacks and Queues.
- The bit vector operations done throughout my program were influenced by the bit vector code provided in Professor Long's CSE13S/Code Comments repository on the UCSC Git lab. The portion in my encoder's main function where the code is being written to the outfile was also influenced by the "dump.c" file within Professor Long's CSE13S/Code Comments repository, along with my code in which counts the occurrences of symbols into a histogram which was influenced by the "entropy.c" file in the same repository.
- The IO module was heavily influenced and built upon both Christian Ocon and Eugene Chou's zoom sections on 10/30/21 and 11/02/21, respectively.