Lynelle Goh
ljgoh
Assignment 7

General Idea

With our program, we will be able to encode and decode Huffman. It will read in a file to construct a histogram that will be used to construct the Huffman tree using a priority queue. This code will be written out to another file. Then, for the decoding process, it will read in the file and reconstruct the Huffman tree until it has finished decoding and that the content matches the original file.

Pseudocode

- The encoder
  - We need to implement these command-line arguments.
    - -h: Prints out a help message describing the purpose of the program and the command-line options it accepts, exiting the program afterwards.
    - -i infile: Specifies the input file to encode. The file input should be set as stdin.
    - -o outfile: Specifies the output file to write the compressed data to. Should be set as stdout.
    - -v: Prints compression statistics to stderr. These statistics include the uncompressed file size, the compressed file size, and space saving.
      - To calculate the space saving, we must do $100 \times (1 - (\text{compressed size}/\text{uncompressed size}))$.
  - Set the count for each of the first two symbols (0 and 1) to 1 if and only if the symbol's count is 0. In other words, if histogram[0] == 0, set histogram[0] = 1, and do the same for symbol 1.
  - Construct the Huffman tree using the priority queue. Call the function by calling build_tree.
    - Create a priority queue by calling its constructor. For each symbol histogram where its frequency is greater than 0 (check the frequency with an if-statement and a temporary variable), create a corresponding Node and insert it into the priority queue.
    - While there are two or more nodes in the priority queue, dequeue two nodes. The first dequeued node will be the left child node. The second will be the right child node. Join these nodes together by calling node_join() and enqueue the joined parent node. The frequency of the parent node is the sum of its left child's frequency and its right child's frequency.
    - Eventually, there will only be one node left in the priority queue. This is the root node of the constructed Huffman tree.

- ➢ Construct a code table by traversing the Huffman tree that we have created. This will be done by calling build_codes().
  - ■ First, construct a new Code by calling code_init(). Perform a post-order traversal by starting at the root of the Huffman tree.
  - ■ Using an if-statement, if the current node is a leaf, the current Code represents the path to the node, and thus is the code for the node's symbol. Save this code into the code table.
    - ● Else, the current node must be an interior node. Push a 0 to Code by using our own function and recursive down to the left link.
    - ● After you return from the left link, pop a bit from Code by calling our function. Pop a bit from Code, push 1 to Code and recursive down the right link. Remember to pop a bit from Code when you return from the right link.
- ➢ Construct a header.
  - ■ Set all macros to their respective variables.
  - ■ The header's permissions field stores the original permission bits of infile. You can get these using fstat(). Set the permissions of outfile to match permissions of infile using fchmod().
  - ■ The header's tree_size field represents the number of bytes that make up the Huffman tree dump. The size is calculated as $(3 \times \text{unique symbols}) - 1$.
  - ■ The file_size is the size in bytes of the file to compress, infile. Obtain this size through fstat().
- ➢ Write this constructed header to outfile by calling dump_tree().
- ➢ Starting at the beginning of infile, write the corresponding code for each symbol to outfile by calling write_code(). When finished all the symbols, make sure to flush any remaining buffered codes by calling flush_codes().
- ➢ Close infile and outfile.

❖ The decoder

- ➢ We need to implement these command-line arguments.
  - ■ -h: Prints out a help message describing the purpose of the program and the command-line options it accepts, exiting the program afterwards. Refer to the reference program provided to you, for an idea of what to print.
  - ■ -i infile: Specifies the input file to decode. Set as stdin.

    - -o outfile: Specifies the output file to write the decompressed input to. The default output should be set as stdout.
    - -v: Prints decompression statistics to stderr. These statistics include the compressed file size, the decompressed file size, and space saving.
        - To calculate the space saving, we must do $100 \times (1 - (\text{compressed size}/\text{decompressed size}))$.
- The permissions field in the header indicates the permissions that outfile should be set to. Set the permissions using fchmod().
- Read in the header from infile and verify the magic number using an if-statement.
    - If the magic number does not match MAGIC, then an invalid file was passed to your program. Print out a helpful error message and quit.
- Read the dumped tree from infile into an array that is tree_size bytes long (we get the size of the dumped tree using the tree_size field in the header). Reconstruct the Huffman tree using rebuild_tree().
    - The array containing the dumped tree will be referred to as tree_dump. The length of this array will be nbytes().
    - Iterate over the contents of tree_dump from 0 to nbytes. This will be done using a for-loop and having the terminating condition be the counter variable being greater than nbytes.
    - If the element of the array is an L, this means to use an if-statement to check, then the next element will be the symbol for the leaf node. Use that symbol to create a new node with node_create(). Push the created node onto the stack by calling our function.
    - If the element of the array is an I, this means to use an if-statement to check, then the element we have encountered is an interior node. Pop the stack one time using our own function to get the right child of the interior node, then pop again to get the left child. Join the left and right nodes with node_join and push the joined parent node on the stack.
    - There will be one node left in the stack after you finish iterating over the contents of tree_dump. This node is the root of the Huffman tree.
- Read infile one bit at a time using read_nit(). You will be traversing down the tree one link at a time for each bit that is read.
    - Starting at the root of the Huffman tree, begin reading.
        - If a bit of value 0 is read, walk to the left child of the current node.

        - Else, if a bit of value 1 is read, then walk to the right child of the current node.
    - If we are at a leaf node, write out the leaf node's symbol to outfile. After writing the symbol, reset the current node back to the root of the tree.
    - Repeat this process until the number of decoded symbols matches the original file size, which is given to us by file_size field in the header that was read from infile. We also use this for our terminating condition.
  - Close infile and outfile.
- Nodes
  - It should be noted that we have worked on a node ADT in the previous assignment, which I will use as a basis for this program as well.
  - Node_create()
    - Allocate memory for nodes. Set node's symbol as symbol and the frequency as frequency. Set the node's left and right to equal NULL. Construct the node to be used for the following functions.
  - Node_delete()
    - Free the specified node and set the pointer to NULL.
  - Node_join()
    - Allocate memory for the node. Set the parent's left and right to equal the passed in left and right from the function. The parent node's symbol will be set as $ and its frequency is the sum of its left child's frequency and its right child's frequency. That means we will need to use the addition operation to add the values and set the value for the parent node.
  - node_print()
    - Prints the symbol followed by the frequency as a uint64_t. I will print control characters or non printable symbols as 0x%02"PRIx8, and printable non-control characters as the character itself. Use iscntrl() and isprint() to identify printable characters that aren't control characters/
- Priority queues
  - We will find our Heapsort implementation from assignment 4 useful.
  - Create our own struct definition and include inside capacity, size, and a node.
  - pq_create()

- Allocate memory for priority queue. Construct priority queue. Using capacity that was passed in, we will set the priority queue's maximum capacity as that. Initialize the priority queue's size as well as allocate memory for the nodes.

➢ pq_delete()
- Create an if-statement to check if the priority queue exists. Free the memory for the priority queue passed in & its nodes and set it to NULL.

➢ pq_empty()
- Create an if-statement to check if the priority queue has no items.
  - Return true if the size is equal to 0 and false otherwise.

➢ pq_full()
- Create an if-statement to check if the priority queue has items.
  - Return if the size of the priority queue is equal to the capacity..

➢ pq_size()
- Return the size of the priority queue.

➢ enqueue()
- Check if it's full before enqueue. If it is full, return false.
  - If not, then return true and enqueue the node.
  - To enqueue, set the array at index of the priority queue's size to equal the passed in node. Using our min heapsort function where we go up the heap, call that function and pass in the priority queue. Increment the size.

➢ dequeue()
- Dequeue a node from the specified priority queue at index 0 and pass it back through the double pointer n. The node dequeued should have the highest priority over all the nodes in the priority queue.
- Return false if, after checking if the priority queue is empty, priority queue is empty prior to dequeuing a node. Return true if the node has been successfully dequeued.
- Decrement the size.
- Also set the 0 index of the priority queue to size index of the priority queue.
- Go down the heap using our function from assignment 4's heapsort.

➢ pq_print()
- This will be a debug function to print out a priority queue. Just add print statements and loops deemed necessary to print out the items in the priority queue.

- Codes
  - code_init()
    - Simply create a new Code on the stack, setting top to 0, and zeroing out the array of bits using a for-loop. The initialized Code is then returned.
  - code_size()
    - Return the top of the code, which works as a counter variable.
  - code_empty()
    - Create an if-statement to check if the Code has no items.
      - Return true if the top is equal to 0 and false otherwise.
  - code_full()
    - Create an if-statement to check if the Code has items. The maximum length of a code in bits is 256, which we have defined using the macro ALPHABET.
      - Return true if the top is equal to ALPHABET and false if otherwise.
  - code_set_bit()
    - Sets the bit at index i in the Code, setting it to 1. Create an if-statement to check if i is out of range, return false.
      - Otherwise, return true to indicate that we have successfully set the bit at index i.
      - To set the bit, we will have to left shift 1 by index modulus and bitwise OR it into our bits at index divided by 8.
  - code_clr_bit()
    - Create an if-statement to check if i is out of range. If yes, then return false.
      - Otherwise, return true and clear the bit at index i in the Code.
      - To clear the bit, we will have to left shift 1 by index modulus 8 (then negate) and bitwise AND it into our bits at index divided by 8.
  - code_get_bit()
    - Create an if-statement to check if i is out of range. If yes, then return false.
      - Otherwise, return true if and only if bit i is equal to 1. To get the bit, we will have to return if this code, where we right shift index modulus 8 into our bits at index divided by 8 and use bitwise AND to 1, is true or not.
  - code_pop_bit()
    - Create an if-statement to check if Code is empty prior to popping a bit. If yes, then return false.

- Otherwise, return true and pop the bit off the Code. The value of the popped bit is obtained using our get bit function. Then we clear the bit. Decrement the top.

➢ code_push_bit()
- Check if our Code is full. If yes, then return false.
  - If not, check if the bit passed in is equal to 1. If yes, then call our set bit function. Increment the top.

➢ code_print()
- A debug function to help you verify whether or not bits are pushed onto and popped off a Code correctly.

❖ I/O

➢ Define the external variables defined in io.h: bytes_read and bytes_written. These variables are used for collecting statistics.

➢ read_bytes()
- Write a wrapper function to loop calls to read() until we have either read all the bytes that were specified by nbytes into the byte buffer buf (this was passed in as well) or until there are no more bytes to read.
- The number of bytes that were read from the input file descriptor, infile, is returned. This means creating a counter variable and a loop to count the amount of bytes that were read.

➢ write_bytes()
- Use the same process as read_bytes, but with write() for the looping calls. We must loop until we have either written out all the bytes specified from the byte buffer buf, or if no bytes were written.
- The number of bytes written out to the output file descriptor, outfile, is returned. This means creating a counter variable and a loop to count the amount of bytes that were written.

➢ read_bit()
- Read in a block of bytes into a buffer and dole out bits out at a time. Whenever all the bits in the buffer have been doled out, simply fill the buffer back up again with bytes from infile.
- Maintain a static buffer of bytes and an index into the buffer that tracks which bit to return to the pointer. The buffer will store BLOCK number of bytes, where BLOCK is yet another macro defined in defines.h.

        - Return false if there are no more bits that can be read or return true if there are still bits to read.
    - write_code()
        - Use the same logic from read_bit(). Make use of a static buffer and an index. Each bit in the code will be buffered into the buffer. The bits will be buffered starting from the 0th bit in c. When the buffer of BLOCK bytes is filled with bits, write the contents of the buffer to outfile.
    - flush_codes()
        - Write out any leftover, buffered bits. Make sure that any extra bits in the last bytes are zeroed before flushing the codes.
- Stacks
    - stack_create()
        - The constructor for stack. The specified maximum number of nodes the stack can hold is passed in.
        - Allocate memory for the stack.
        - Set the stack's top to 0 and its capacity as the struct's capacity. Allocate memory for the items inside the stack.
    - stack_delete()
        - Free the memory allocated by the stack and set the pointer to NULL. Free the stack's items first.
    - stack_empty()
        - Create an if-statement to check if the stack has no items.
            - Return true if it is empty and false otherwise.
    - stack_full()
        - Create an if-statement to check if the stack has items.
            - Return true if it is full and false if empty.
    - stack_size()
        - Return the stack top.
    - stack_push()
        - Push a node onto the stack. Create an if-statement to check if the stack is full prior to pushing the node and return true otherwise. If true, actually push the node onto the stack by setting the stack item at index top to equal the passed in node. Increment the top.
    - stack_pop()

        - Pops a node off the stack, passing it back through the passed in double pointer n. Return false if the stack is empty prior to pooping a node and return true otherwise. If true, successfully pop off the node by freeing the item at index top minus 1 after setting our passed in node. Set the item at the stack at index top minus 1 to NULL. Decrement the top.
    - stack_print()
        - Print the contents of the stack out.
- Huffman coding module
    - build_tree()
        - Returns the root node of the constructed tree. Construct a Huffman tree given a computed histogram, which will have ALPHABET indices (one index for each possible symbol).
    - build_codes()
        - Populates a code table and builds the code for each symbol in the Huffman tree. The constructed codes are copied to the code table, table, which has ALPHABET indices, one index for each possible symbol.
    - dump_tree()
        - Conducts a post-order traversal of the Huffman tree rooted at root (which was passed in), and writes it to outfile. This should write an L followed by the byte of the symbol for each leaf and an I for interior nodes. You should not write a symbol for an interior node.
    - rebuild_tree()
        - Reconstructs a Huffman tree given its post-order tree dump stored in the array tree_dump. The length in bytes if tree_dump is given by nbytes. Returns the root node of the reconstructed tree.
    - Delete_tree()
        - This requires a post-order traversal of the tree to free all the nodes. Set the pointer to NULL after we are finished freeing all the allocated memory.
- Makefile
    - make and make all should build the encoder and the decoder. Set encode, decode, and all files needed for it into the option line.
    - make encode should build just the encoder. Set encode and all files needed for it into the option line.

- make decoder should build just the decoder. Set decode and all files needed for it into the option line.
- make clean will remove all files that are not compiler generated.
- make spotless will remove all fields that are not compiler generated as well as the executable.
- make format will format all of our source code, including header files.