Karthi Sankar

kasankar@ucsc.edu

11/8/21

CSE 13S Assignment 5 Design Document

Purpose:

The purpose of this assignment is to implement both a Huffman encoder and decoder. What a Huffman encoder does is to encode the contents of a file and to compress it. An input file is given, its contents are encoded, and this encoding of the contents works to compress the file, or represents its contents in a way that takes up less bits than its regular state. Decoding works the same way, but in the opposite way. The decoder will take in a compressed input file and decompress it, making it its original size that it was before the compression.

Structure:

This program will require the use of many ADTS and a specific file input/output file (io.c). The ADTs that will be used for this program are a node, a priority queue, a code, a stack, a tree (or Huffman tree). Each of these will be contained in its own file. The encoding will be handled in a file called encode.c; the decoding will be handled in a file called decode.c. Essentially, a Huffman tree will be constructed using nodes and queues. Then, the codes for each symbol in a Huffman tree will be kept track of. Then the corresponding codes will be written to an output file (a compression of the input file). The decoding will take a tree and “unravel” it in order to print out the symbols in the frequencies that have been kept track of. The resulting file will be decompressed and back to its larger size.

Description/Pseudocode:

(pseudocode in Python + English)

Note: ptr is a pointer and -> is a . in my pseudocode

node.c:

```
Node ptr node_create (int symbol, int frequency):
   using malloc, allocate a Node n (size of Node)
   if node ptr n exists:
      n.symbol = symbol
      n.frequency = frequency
   #end if
   return n

void node_delete (double ptr n):
   if ptr n, n.left, n.right exist:
      free left child
      free right child
      free n (pointer)
      set n to NULL
   #end
   return

Node ptr node_join (Node left, Node right):
   Node parent;
   int sum = left.frequency + right.frequency
   parent = node_create('$', sum);
   parent.left = left
   parent.right = right
   return parent;

void node_print (Node n) -> used for debugging
```

The first two functions, node_create and node_delete, are responsible for the construction and destruction of a Node. The function node_create returns a pointer to a node. Its parameters are an unsigned 8-bit integer, symbol (which represents any possible ASCII character) and an unsigned 64-bit number frequency (which represents the number of times a symbol appears in a file). First, a Node is dynamically allocated using malloc. Then, if the malloc was successfully executed (and the node pointer named n exists), then the node’s symbol is set to the parameter symbol, and the node’s frequency is set to the parameter frequency. Finally, the pointer to a node, n, is returned. The return type of node_delete is void. It takes a double pointer to a node as a parameter. If the node, the node’s left child, and its right child (which are both nodes) exist, then all the memory must be freed. The node’s right and left child are both freed. Then, the node pointer n is freed and set to NULL. Finally, I put a return with no value at the end.

The next function is node_join. The function returns a pointer to a node. This node will be the parent node of the two parameters given to the function, a node left and a node right. First, I declare a node named parent. Then, I create an integer variable sum, that will hold the sum of the frequencies of the parameter’s left and right, which will be the frequency of the parent node. Then I initialize the parent node using node_create, using the arguments ‘$’ as the symbol (which will be the same for every parent node) and the variable sum as the frequency. Finally, I set the parent nodes’ left and right child to be the parameters left and right. Lastly, I return the parent node. The final function in node.c, node_print takes in a node and is used to debug.

pq.c:

(enqueue/dequeue went over in Eric’s tutoring section)

```
PriorityQueue ptr pq_create(int capacity):
    allocate the PriorityQueue pointer q with malloc
    if q:
        q.capacity = capacity
        q.size = 0
        allocate q.items (array of Node ptrs) with calloc
    #end if
    return q

void pq_delete(PriorityQueue double ptr):
    if q and q.items:
        free(q.items)
        free(q)
        q = NULL
    #end if
    return
```

The first two functions are pq_create and pq_delete. Pq_create takes one parameter, a 32-bit unsigned integer capacity, which is the max amount of elements in the priority queue, and returns a pointer to a priority queue. First, I allocate a priority queue pointer named q using malloc. Then if the malloc was successful, I set the capacity to the given parameter capacity, the queue's size to 0, and finally, I allocate the array of Node pointers representing the queue (named items). The array items is set to the size of capacity + 1 (due to 1-based indexing). Finally, I return the pointer q. Pq_delete's return type is void and takes a double pointer to a priority queue. If the pointer to the queue, q, and the array, items, exists, I first free the array items and then the pointer q. Finally, I set the pointer q to NULL, and use a return with no value.

```
bool pq_empty (PriorityQueue ptr q):
    if q.size == 0:
        return true
    return false

bool pq_full (PriorityQueue ptr q):
    if q.size == q.capacity:
        return true
    return false

int pq_size (PriorityQueue ptr q):
    return q.size
```

The next three functions check if the queue is empty or full and then its size. All three functions take in a pointer to a priority queue. Because the variable size tracks how big the queue is, it is used to check whether a queue is empty or full. For pq_empty, if the queue's size is 0, true is returned, otherwise false. For pq_full, if the queue's size is equal to its capacity, true is returned, otherwise false. Another function, pq_size, returns the queue's size.

```
void min_heap_insert(PriorityQueue ptr, Node ptr n):
    int = q.size + 1
    int parent = i/2
    q.items[i] = n
    while i > 1 and parent frequency > q.items[i] frequency:
        swap nodes at parent and i
        i = parent
        parent = i/2
    #end while

void min_heapify (PriorityQueue ptr, int i):
    int left = i*2
    int right = i*2 + 1
    int small = 0
    if left <= q.size and left frequency < q.items[i] frequency:
        small = left
    else:
        small = i
    if right <= q.size and right frequency < q.items[i] frequency:
        small = right
        if small != i:
            swap nodes at parent and i
            min_heapify(q, small)
        #end if
    #end if
```

The next two functions are essentially from the book *Introduction to Algorithms*, but they are modified in order to use a min heap instead of a max heap. Min_heap_insert takes in a priority queue pointer and a node pointer, and its return type is void. The function works by placing a node into the array items (the queue). Then, it will perform swaps until the min heap property is maintained, using a while loop. It keeps the heap so that it is a min heap after inserting a new node into the queue. Min_heapify takes in a priority queue pointer and a 32-bit unsigned integer i, and its return type is void. It essentially is called to maintain the min heap property of the queue. The left and right child of the parent (at index i) is calculated. Then, the smallest child is found, and min_heapify is called recursively until the heap is once again a min heap.

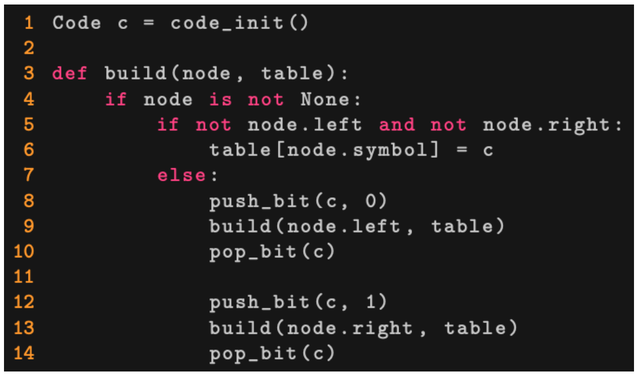
```
bool enqueue (PriorityQueue ptr, Node ptr n):
    if queue is full:
        return false
    else:
        min_heap_insert(q, n)
        q.size += 1
        return true
    #end if-else

bool dequeue (PriorityQueue ptr, Node double ptr n):
    if queue is empty:
        return false
    else:
        ptr n = q.items[i]
        q.items[i] = q.items[q.size]
        q.size -= 1
        min_heapify(q, 1)
        return true
    #end if-else
```

Enqueue and dequeue both take in as parameters a priority queue pointer and a node pointer, with dequeue taking in a node pointer, and return a boolean that says if the operation was successful. In enqueue, I first check if the queue is full; if it is, false is returned. Otherwise, the node n passed to the function is inserted into the queue, the queue's size is increased by one, and true is returned. In dequeue, I first check if the queue is empty; if it is, false is returned. Otherwise, the node at the first index of the queue (the head) is passed to the parameter node pointer n because it will always be dequeued first. Then I set the value at index 1 in the queue to be the value at the index size. Then, I decrement the queue's size by 1 and call min_heapify on the queue to maintain the min heap property. Finally, true is returned.

huffman.c:

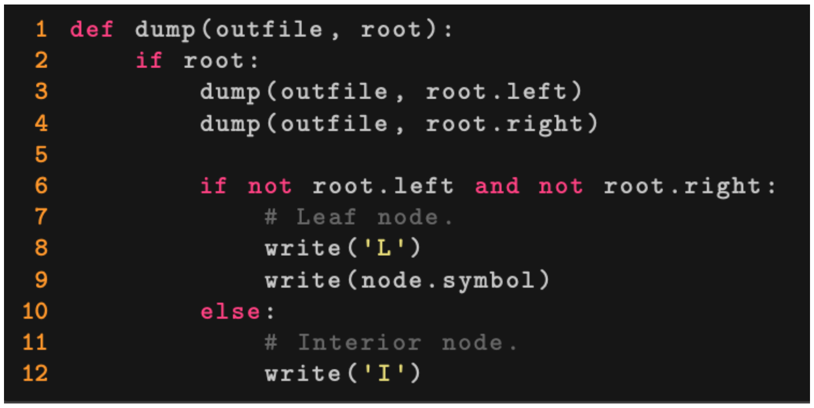
```
Node ptr build_tree (int hist[static ALPHABET]):
    PriorityQueue ptr q = pq_create(ALPHABET)
    for int i = 0; i < ALPHABET; i += 1:
        if hist[i] > 0:
            Node ptr n = node_create(i, hist[i])
            enqueue(q, n)
    #end for
    while (q.size > 1):
        dequeue(q, left)
        dequeue(q, right)
        Node ptr parent = mode_join(left, right)
        enqueue(q, parent)
    #end while
        dequeue(root)
        delete(q)
        return root
```

This file is responsible for everything to do with the Huffman tree to be constructed and deleted in order to perform encoding and decoding. This first function build_tree takes in a histogram (of 64-bit unsigned ints), and returns a node pointer. First, I create a priority queue with the capacity of ALPHABET, the same size as the histogram parameter. Then I construct the priority queue by enqueuing a node everytime a symbol in the histogram has a frequency greater than 0. Then, the queue is “deconstructed”, but with parent nodes. This is done by dequeuing two nodes, and then enqueuing a parent node constructed with node_join. Finally, the last remaining node in the queue is the root. So that node is dequeued, the priority queue is deleted, and the root node is returned.



```
Code c = code_init()

def build(node, table):
    if node is not None:
        if not node.left and not node.right:
            table[node.symbol] = c
        else:
            push_bit(c, 0)
            build(node.left, table)
            pop_bit(c)

            push_bit(c, 1)
            build(node.right, table)
            pop_bit(c)
```

The next function, build_codes, constructs a code table for all the symbols in the Huffman tree. It takes in a node pointer root (root of the Huffman tree) and a “code table”, an array of codes, and its return type is void. It essentially works by traversing the Huffman tree. If a leaf node is reached (no left or right child), the value of the code at a certain symbol is set in the code table. Of course, this involves a code named c, which is local to the function. Otherwise, a 0 is pushed to a code if a left child is reached and a 1 is pushed if a right child is reached. Build_codes is called recursively in order to populate the code table.



```
def dump(outfile, root):
    if root:
        dump(outfile, root.left)
        dump(outfile, root.right)

        if not root.left and not root.right:
            # Leaf node.
            write('L')
            write(node.symbol)
        else:
            # Interior node.
            write('I')
```

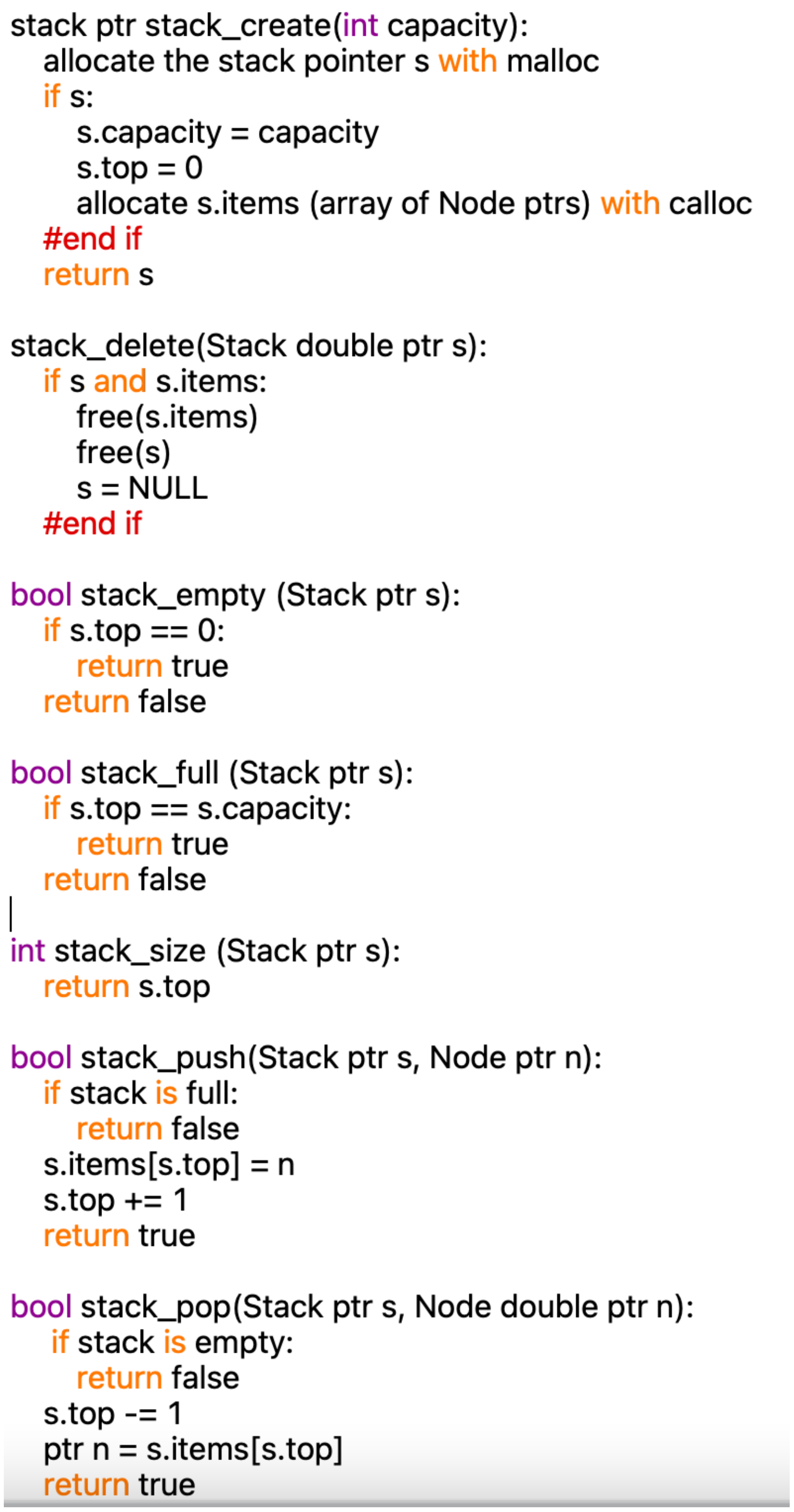
The next function, dump_tree performs a post-order traversal of the Huffman tree in order to create a “dump” representing the tree. It takes in an integer file descriptor (for an outfile) and a node pointer (the root of the tree). First, if the node root isn’t NULL, dump_tree is called recursively on the root’s left and right child to begin the post-order traversal. If a leaf node in the tree is reached (it has no left or right child), then an ‘L’ and the corresponding node’s symbol are printed. It is done using write_bytes from io.c. If an interior node is reached, an ‘I’ will be printed to the outfile. This is also done using the function write_bytes(). The function’s result will be a full dump of the Huffman tree printed to an output file.

```
Node ptr rebuild_tree(int nbytes, int tree[static nbytes]):
  Stack ptr s = stack_create(nbytes)
  for i in range (0, nbytes, 1):
    if tree[i] = 'L':
      Node ptr n = node_create node_create(tree[i+1], 0)
      stack_push(s, n)
    if tree[i] = 'I':
      stack_pop(s, right)
      stack_pop(s, left)
      Node ptr parent = mode_join(left, right)
      stack_push(s, parent)
    #end if
  #end for
  stack_pop(s, root)
  delete stack s
  return root

void delete_tree (Node double ptr root):
  if ptr root != NULL:
    delete_tree(root.left)
    delete_tree(root.right)
    node_delete(root)
  #end if
```

Rebuild_tree returns a node pointer, which will be the root of the re-constructed tree. Its parameters are a 16-bit unsigned int nbytes, and an array of unsigned 8-bit ints called tree (which will be the tree dump). The tree will be reconstructed using a stack of nodes. First, the stack is constructed to be of the size nbytes. Then, I iterate through the tree dump. If an L is encountered (leaf node), a new node is constructed and pushed with the corresponding symbol. If an I is encountered (interior node), then its children are popped off the stack and joined to make a parent node. Then that parent node is pushed to the stack. The last remaining node in the stack will be the root node. So the root is popped off the stack, the stack is deleted, and the root node is returned.

The final function in huffman.c is delete_tree, responsible for deleting all memory relating to the Huffman tree. Its return type is void and takes in a double pointer to a node (the root of the tree). If the root is not NULL, its left and right child are passed to delete_tree and called recursively. This is a part of a post-order traversal. Finally, the root node is deleted at the end of the function

stack.c:



```
stack ptr stack_create(int capacity):
   allocate the stack pointer s with malloc
   if s:
      s.capacity = capacity
      s.top = 0
      allocate s.items (array of Node ptrs) with calloc
   #end if
   return s

stack_delete(Stack double ptr s):
   if s and s.items:
      free(s.items)
      free(s)
      s = NULL
   #end if

bool stack_empty (Stack ptr s):
   if s.top == 0:
      return true
   return false

bool stack_full (Stack ptr s):
   if s.top == s.capacity:
      return true
   return false

int stack_size (Stack ptr s):
   return s.top

bool stack_push(Stack ptr s, Node ptr n):
   if stack is full:
      return false
   s.items[s.top] = n
   s.top += 1
   return true

bool stack_pop(Stack ptr s, Node double ptr n):
    if stack is empty:
      return false
   s.top -= 1
   ptr n = s.items[s.top]
   return true
```

The first two functions, stack_create and stack_delete, construct and destruct the stack respectively. Stack_create returns a pointer to a stack and takes in a 32-bit unsigned int capacity as a parameter. First, the stack pointer is created using malloc. If the malloc was successful, the capacity is set, the top is set to 0, and the array of nodes (representing the stack) is dynamically allocated using calloc. Finally, the pointer to the created stack is returned. Stack_delete's return type is void, and it takes in a double pointer to a stack. If the stack and its items exist, they are both freed, and the stack pointer is then set to NULL.

The next three functions take in a pointer to a stack as a parameter. Stack_empty returns a boolean. If the stack top is equal to 0, true is returned, otherwise false. Stack_full returns a boolean as well. If the stack's top is equal to its capacity, true is returned, otherwise false. Finally, stack_size returns a 32-bit unsigned int, and it returns the value of the stack's top.

Stack_push returns a boolean that states if the push was successful or not and takes in as parameters a stack pointer and a node pointer (node to be pushed). If the stack is full, false is returned. Otherwise, the value at the index top of the array of nodes is set to the passed node, the top is incremented by 1, and true is returned. Stack_pop returns a boolean that states if the pop was successful or not and takes in as parameters a stack pointer and a node pointer (node that has been popped off). If the stack is empty, false is returned. Otherwise, the stack's top is decremented by 1, the node at the top of stack (the one that has been popped off) is passed to the parameter node pointer, and true is returned.

code.c:

```
Code code_init (void):
   Code c
   c.top = 0;
   for i in range(0, 256/8):
      c.bits[i] = 0
   #end for
   return c

int code_size (Code c):
   return c.top
```

A code is another ADT to be implemented for Huffman encoding. However, it requires no dynamic memory allocation, so it only needs an initialization or construction function. This function comes in the form of code_init(). A code has a top and an array of bits, so these will need to be set. First, a code named c is simply declared. Then, the code's top is set to 0. Then, a for loop is used to fill up the array bits up with zeroes to start. The for loop iterates until 256/8, which is the max size of the bits array. Finally, the declared code c is returned.

The next function, code_size() returns a 32-bit unsigned integer representing the size of the code at the current point. It takes a code c as a parameter. The top of a code should always point to its current size, so the code's top is just returned.

```
bool code_empty (Code c):
   if c.top == 0:
      return true
   #end if
   return false

bool code_full (Code c):
   if c.top = 256/8:
      return true;
   # end if
   return false
```

The next two functions check if a code is either empty or full. Both return a boolean and take a code as a parameter. Code_empty returns true if the code's top is equal to 0 and false if it's not. Code_full is very similar. It checks if the code's top is equal to 256/8 (the max size for the bits array of a code). If the top is equal to 256/8, true is returned; otherwise, false is returned.

```
bool code_set_bit (Code c, int i):
   if !(i < MAX_CODE_SIZE):
      return false
   c.bits[i/8] |= 1 << (i % 8))
   return true

bool code_clr_bit (Code c, int i):
   if !(i < MAX_CODE_SIZE):
      return false
   c.bits[i/8] &= 1 << (i % 8))
   return true

bool code_get_bit (Code c, int i):
   if !(i < MAX_CODE_SIZE):
      return false
   return (c.bits[i/8] >> i % 8) & 1

bool code_push_bit (Code c, int bit):
   if code is full:
      return false
   if bit == 1:
      code_set_bit(c, c.top)
      return true
   else:
      code_clr_bit(c, c.top)
      return true
   #end if-else

bool code_push_bit (Code c, int ptr bit):
   if code is empty:
      return false
   c.top -= 1
   ptr bit = code_get_bit(c, c.top)
   return true
```

Code_set_bit is used to set a certain bit in the bits array of code. It returns a boolean stating if the setting of a bit was successful, and it takes a code and an integer parameter i (an index to set) as parameters. First, I check if the parameter i given is out of bounds, namely if it is greater than the max code size. If it is, false is returned. Otherwise, I set the bit. This is done using bit operations, including bitwise or and the left shift operator. Because a bit is an 8-bit integer, the index to be accessed is i/8. Then, to place a one in the bits array, a bitwise or is used. Essentially, a 1 is left-shifted by (i % 8) places in an 8-bit number (because of a bit's size). Then that number is bitwise-ored with the index [i/8] in the array bits. Because a 0 and 1 orred together will always yield a 1, this results in a 1 being placing the index necessary within the array bits. Then, a true is returned, signaling a successful setting of a bit.
Code_clr_bit is very similar to set_bit. Code_clr_bit is used to clear a certain bit in the bits array of code. It returns a boolean stating if the clearing of a bit was successful, and it takes a code and an integer parameter i (an index to set) as parameters. First, I check if the parameter i given is out of bounds, namely if it is greater than the max code size. If it is, false is returned. Otherwise, I clear the bit. This is done using bit operations, including bitwise and and the left shift operator. Because a bit is an 8-bit integer, the index to be accessed is i/8. Then, to place a one in the bits array, a bitwise and is used. Essentially, a 1 is left-shifted by (i % 8) places in an 8-bit number (because of a bit's size). Then that number is bitwise-anded with the index [i/8] in the array bits. Because a 0 and 1 anded together will always yield a 0, this results in a 0 being placing the index necessary within the array bits. Then, a true is returned, signaling a successful clearing of a bit.

Code_get_bit is also similar. It returns a bit at a certain index in the bits array. It returns a boolean stating if the getting of a bit was successful, and it takes a code and an integer parameter i (an index to get) as parameters. First, I check if the parameter i given is out of bounds, namely if it is greater than the max code size. If it is, false is returned. Otherwise, I get the bit. This is done using bit operations, including bitwise and and the right shift operator. Because a bit is an 8-bit integer, the index to be accessed is i/8. In order to have the same value at a certain index be returned, a bitwise and is used. This is because a value and 1 will always return the value of the number given. Then, the whole operation of getting a bit is returned, because this will signal that the getting of the bit was successful.

Finally, there is code_push_bit and code_pop bit. Each returns a boolean stating if the push/pop of a bit is successful or not, and take in as a parameter a pointer to a code. Code_push_bit also takes in a bit value to be pushed, an 8-bit unsigned int. If the code is full, false is returned. Otherwise, if the bit is 1, I use set_bit to put a one in the bits array at the top and return true. If the bit is 0, I clear the bit at the top (making it 0) and return true. Code_pop_bit takes in a pointer to a bit, which will be used to hold the popped off bit from the array. If the code is empty, false is returned. Otherwise, the top of the code is decremented by 1 and the value of the bit at the top is passed to the parameter bit using the function get_bit(). Finally, true is returned, signalling a successful pop.

io.c:

(read_bytes/write_bytes went over in Eugene's section)

(read_bit/write_code/flush_codes went over in Eric's tutoring section)

```
bytes_read = 0
bytes_written = 0

static int bit_index = 0
static int bit_buf = { 0 }

static int code_index = 0
static int code_buf = { 0 }
```

The IO file will be used to access and then print input from designated input files to output files. Two extern variables, bytes_read and bytes_written, are included in the io header file and will be updated as needed within the necessary functions. I also declare and set to 0 two indexing variables and buffers: bit_index and bit_buf to be used for reading and writing bytes and code_index and code_buf to be used for the use of codes. The buffer has a size of BLOCK, because a block is read each time and looped for the reading of bytes.

```
int read_bytes (int infile, int buf, int nbytes):
    int bytes = -1
    int total_bytes = 0
    while (total_bytes != nbytes):
        set bytes to read(infile, buf, nbytes)
        add bytes to total bytes
        add bytes to buf
    # end while
    add total_bytes to bytes_read
    return total_bytes

int write_bytes (int outfile, int bit_buf, int nbytes):
    int bytes = -1
    int total_bytes = 0
    while (total_bytes != nbytes):
        set bytes to write(outfile, buf, nbytes)
        add bytes to total bytes
        add bytes to buf
    # end while
    add total_bytes to bytes_written
    return total_bytes
```

The structure of read_bytes and write_bytes are exactly the same, except for the fact that read_bytes uses the read function and takes input from an input file, and write_bytes uses the write function and writes output to an output file. Both return an integer, being the total number of bytes_read or written respectively. The function read_bytes takes as parameters a file descriptor named infile, a buffer (of unsigned 8-bit numbers) and an integer nbytes, the specific number of bytes to read. The function write_ bytes takes as parameters a file descriptor named outfile, a buffer (of unsigned 8-bit numbers) and an integer nbytes, the specific number of bytes to write. In both functions, I declare integer variables bytes (set to -1) and total_bytes (set to 0). Then, I use a while loop which iterates while total_bytes is not equal to nbytes. Within the loop, I set bytes equal to read(infile, buf, nbytes), and set it to write(outfile, buf, nbytes) within write_bytes. Then, I add the integer bytes to total_bytes, updating its value, and I add bytes to buf (using pointer arithmetic) because the buffer also has to be updated. At the conclusion of the while loop, I add total_bytes to bytes_read (and bytes_written, respectively) because they are stats that need to be updated with each function call. Finally in each function, I return the integer total_bytes.

```
bool read bit (int infile, int ptr bit):
    int end = -1
    if bit_index == 0:
        int bytes = read_bytes(infile, bit_buf, BLOCK)
        if bytes < BLOCK:
            end = bytes += 1
        #end if
    #end if
    ptr bit = (bit_buf[bit_index/8] >> (bit_index % 8)) & 1
    bit_index = (bit_index + 1) % (BLOCK * 8)
    return (bit_index == (8 * BLOCK))

void write_code (int outfile, Code ptr c):
    for i in range(0, code_size(c), 1):
        if bit in c.bits[i] == 1:
            set bit at code_buf[code_index] to 1
    #end for
    code_index = (code_index+1) % (8 * BLOCK)
    if (code_index == 0):
        write_bytes(outfile, code_buf, BLOCK)
        memset(code_buf, 0, BLOCK)

void flush_codes (int outfile):
    int bytes_to_write = (code_index / 8) if (code_index % 8) == 0
    otherwise bytes_to_write = (code_index / 8 + 1
    write_bytes(outfile, code_buf, bytes_to_write)
```

The function read_bit is used to read bits, which will be done when decoding a file. Its return type is a boolean, to signal that the bits were read. Its parameters are the input file to be read from and a pointer to a bit, which will be given a value. I use a static integer end to be the end of the buffer. Then if the index in the buffer is 0, read_bytes() is called to read from the input file, using the necessary buffer. If bytes is less than BLOCK, less than a block has been read, then the end of the buffer is updated to the index one after the last valid byte was read. Then the bit value is passed to the parameter bit using bitwise operations (like the buffer is a bit vector). Then the index within the buffer is reset with the BLOCK size and the fact that the buffer is made of

8-bit unsigned ints. Finally, the boolean that results from the statement that the bit_index is at the right value (all bits have been read) is returned, and should be true.

The function write_code writes all of the codes to the output file. It buffers a code, and the buffer is written when it’s full. Its return type is void, and its parameters are an integer file descriptor and a pointer to a code c. I iterate through the code, using code_size(), and if the bit in the code’s bits array is 1, I set the bit at the corresponding index in the buffer to be 1 using bitwise operations. Then the code’s index is reset. If it has been reset to 0 (the buffer is full), then the values of the buffer are written to the outfile using read_bytes(). Then the function memset() is used to zero out the buffer for the codes.

The final function in io.c, flush_codes, will write any remaining bits after the use of write_code() (because we are writing BLOCKs at a time). Its return type is void, and it takes in an integer file descriptor for the output file. The number of bytes left to write is determined. If the index in the code buffer is divisible by 8, then the number of bytes to write will be the index / 8, otherwise, it will be the index / 8 + 1, accounting for an extra bit. Finally, the remaining bits are written using write_bytes(), passing the number of remaining bytes as a parameter.

encode.c:

```
int main(int argc, char double ptr argv):
    bool verbose = false
    bool help = false
    int infile = STDIN
    int outfile = STDOUT

    while there is input to command line:
        switch():
            case 'v':
                verbose = true
            case 'h':
                help = true
            case 'i':
                infile = input file
            case 'o':
                infile = output file
            default:
                break
        #end switch

        int hist[ALPHABET] = {0}
        hist[0] += 1
        hist[ALPHABET-1] += 1
        int symbols = 0

        while (bytes_read = read_bytes(infile, h_buffer, BLOCK)) > 0):
            for i in range(0, bytes_read, 1):
                if hist[h_buffer[i]] == 0:
                    symbols += 1
                #end if
                hist[h_buffer[i]] += 1
            #end for
        #end while



        Node ptr root = build_tree(hist)
        Code table[ALPHABET] = {0}
        build_codes(root, table)

        set header values (magic, permissions, tree_size, file_size)
        write header to outfile with write_bytes

        reset to beginning of file with lseek(infile, 0, SEEK_SET)
        while bytes are left to read in file:
            for i in range(0, bytes_read, 1):
                write_code(outfile, table[i])
        #end while
        flush_codes(outfile)

        if verbose:
            print compression stats to stderr

        if help:
            print help message to stderr

        delete_tree(root)
        close infile and outfile
        return 0
```

This file is responsible for the parsing of command line options and the subsequent encoding. Within the main function, I start by setting necessary booleans and the file descriptors (which will by default be the stdin/stdout). Then, I use a switch statement to parse the options. If -v or -h are given, the corresponding boolean is set to true. If -i or -o are given, the file descriptors are set to be the given input or output file. Then after that, the histogram is built. The first and last index of the histogram array (an array of 64-bit unsigned ints) are incremented. Then using a buffer, I read through the input file using read_bytes(), filling the histogram at each index every time a new symbol is encountered. I also increment a variable I use to track the number of unique symbols. Then, I build the Huffman tree and the code table, using build_tree() and write_codes. Then, I construct a header using the struct we were given in the file header.h. Each member of the header struct is set using the various methods. The permissions and file_size are set using fstat(). Then, I write the header to the output file. After that, I dump the tree using the root of the constructed Huffman tree. Then, I go back and read through the input file again using lseek() in order to write the codes to the output file using write_codes(). Then I use flush_codes to write any remaining bits. Finally, I print the compression statistics (if -v was a given command line option) and delete any allocated memory using delete_tree. Then, I close the input and output files and return 0, for the end of the program.

decode.c:

```
int main(int argc, char double ptr argv):
    bool verbose = false
    bool help = false
    int infile = STDIN
    int outfile = STDOUT

    while there is input to command line:
        switch():
            case 'v':
                verbose = true
            case 'h':
                help = true
            case 'i':
                infile = input file
            case 'o':
                infile = output file
            default:
                break
        #end switch

    read all value into header (magic, permissions, tree_size, file_size)

    int tree_dump = [tree_size]
    read_bytes(infile, tree_dump, tree_size)


    Node ptr root = rebuild_tree(tree_size, tree_dump)
    int decoded_syms = 0
    int bit = NULL

    Node ptr curr = root
    while decoded_syms != file size:
        read_bit(infile, bit)
        if bit == 0:
            curr = curr.left
        if bit == 1:
            curr = curr.right
        if (c.left and c.right == NULL):
            write_bytes(outfile, curr.symbol, 1)
            decoded_syms += 1
            curr = root
    #end while

    if verbose:
        print compression statistics to stderr

    if help:
        print help message to stderr

    node_delete(root)
    close infile and outfile
    return 0
```

This file is responsible for the parsing of command line options and the subsequent decoding. Within the main function, I start by setting necessary booleans and the file descriptors (which will by default be the stdin/stdout). Then, I use a switch statement to parse the options. If -v or -h are given, the corresponding boolean is set to true. If -i or -o are given, the file descriptors are set to be the given input or output file. Then, I read the necessary values into the header struct using read_bytes(). Then I create an 8-bit unsigned int array to store the values of the tree dump into an array and read the dump into the array using read_bytes(). Then, I rebuild the tree using rebuild_tree, passing the tree dump array in as one of the parameters. Then, I read through the input file one bit at a time and walk through the constructed Huffman tree. Until the end of the file is reached, I walk through the tree. When I get to a leaf node (going left/right using the values of 0/1), I print the symbol of the current node to the output file using write_bytes(). This is done until all the decoded symbols are printed out. Then, compression statistics are printed if necessary and I delete the root node of the constructed Huffman tree. Finally, I close the input and output file and return 0, signaling the end of the program.