Assignment 6 Design Document

Sean Siddens

CSE13s - Spring 2021

---

encode.c

`encode` encodes an input file into its corresponding Huffman codes. An input or an output file name can be provided, but by default it reads from STDIN and outputs to STDOUT. An additional argument can be provided to indicate to the program to print the compression statistics. If a named output file is provided, it should match the file permissions of the input file, if also provided.

After processing the command line arguments and changing the output file permissions, an 256 item array is initialized to act as a histogram to store the counts of each byte which appears in the input file. The 0th and the 255th index of the array is incremented once in order to have a minimum of two items in our tree. After the histogram array is initialized, we read in each byte of the input file and increment the index of the histogram corresponding to the byte we read in. We rely on the `read_bytes` function, which is defined in `io.c`. We also use this initial pass through the file to keep track of the input file size.

We then iterate through the histogram and count how many non-zero elements there are. This gives us the total number of unique symbols in the input file (a count of zero means the symbol never occurred in the file).

The histogram is passed to the function `build_tree`, defined in `huffman.c`, and will return the pointer to the root of the Huffman tree created from our histogram. Next, an array of codes is initialized to store the code for each of our symbols, and this array is populated by the function `buid_codes`, also defined in `huffman.c`.

Next, we create a header struct, and assign the members for the magic number (defined in defines.h), the file permissions, the size of the tree, and the size of the file. The header is written to the outfile using `write_bytes`, defined in `io.c`.

Next, a buffer is created to store the tree, which is filled by the function `tree_dump`. This function encodes our Huffman tree in a way that the decoding file can reconstruct it. It does this by doing a post-order traversal of the tree, adding an ‘L’ and a symbol to the buffer when reaching a leaf node, and adding an ‘I’ to the buffer when reaching an internal node. A static

variable is used to track the index of where we are in the tree dump buffer. After our tree dump buffer is filled, it's contents are written out to the output file.

Next, we seek back to the beginning of the input file, and read in each byte, fetch it's corresponding code from the code table, and write the code out using `write_code`, defined in `io.c`. An additional function, `flush_codes`, is also used in order to write out any codes left in our buffer.

Finally, if the user specified, the compression statistics are printed out: the uncompressed file size, the compressed file size, and the space saving percentage. After all data is freed and deallocated, the program ends.

```
main(argc, argv):
      HELP, VERBOSE = false
      infile_name, outfile_name = NULL
      infile = STDIN
      outfile = STDOUT

      while ((arg = get_arg(argc, argv)) != -1):
            if arg == ‘h’:
                  HELP = true
            if arg == ‘v’:
                  VERBOSE = true
            if arg == ‘i’:
                  infile_name = name provided
            if arg == ‘o’:
                  outfile_name = name provided

      if (HELP):
            Print help and program usage
            return 0

      if (infile_name != NULL):
            infile = open(infile_name)

      if (outfile_name != NULL):
            outfile = open(outfile_name)
            outfile.permissions = infile.permissions

      histogram[ALPHABET] =  { 0, 0, …, 0} // ALPHABET = 256
      histogram[0] += 1
      histogram[255] += 1

      buffer[BLOCK] = { 0, 0, …, 0} // BLOCK = 4096
```

```
      while ((bytes = read_bytes(infile, buffer, BLOCK)) != 0):
            for (i = 0; i < bytes; i++):
                  histogram[buffer[i]] += 1

      uncompressed_file_size = bytes_read // External var defined in io.c

      unique_symbols = 0
      for (i = 0; i < ALPHABET; i++):
            if (histogram[i] != 0):
                  unique_symbols += 1

      root = build_tree(histogram)
      code_table[ALPHABET] = {0, 0, …, 0}
      build_codes(root, code_table)

      header.magic = MAGIC // MAGIC = 0xDEADBEEF
      header.permissions = infile.permissions
      header.tree_size = (3 * unique_symbols) - 1
      header.file_size = bytes_read

      write_bytes(outfile, header)

      tree_buf = allocate tree_size # of bytes
      tree_dump(root, tree_buf)
      write_bytes(outfile, tree_buf)

      seek to beginning of file
      while ((bytes = read_bytes(infile, buffer, BLOCK)) != 0):
            for (i =  0; i < bytes; i++):
                  c = code_table[buffer[i]]
                  write_code(outfile, c)
      flush_codes(outfile)

      uncompressed_file_size = bytes_written // External var defined in io.c

      if (VERBOSE):
            print uncompressed file size
            print compressed file size
            space_saving = 1.0 -
(compressed_file_size/uncompressed_file_size)
            Print space saving
```

```
      Deallocate buffers
      delete_tree(root)
      Close files
      Return 0


tree_dump(root, tree_buf):
      static index
      if (root.left == NULL AND root.right == NULL):
            tree_buf[index] = ‘L’
            tree_buf[index+1] = root.symbol
            index += 2
            return
      else:
            tree_dump(root.left, tree_buf)
            tree_dump(root.right, tree_buf)
            tree_buf[index] = ‘I’
            index += 1
            return
```

---

huffman.c

`huffman.c` contains the functions used by the encoding and decoding programs for initially constructing the Huffman tree, building the codes using the Huffman tree, rebuilding the tree, and deleting the tree. `build_tree` relies on a priority queue and `rebuild_tree` relies on a stack.

build_tree constructs the Huffman tree given a computed histogram of byte frequencies. First, it creates leaf nodes out of every byte in the histogram with a non-zero frequency, using it’s index of the histogram as it’s symbol, getting its frequency in the histogram. The priority queue ensures a total ordering such that nodes with the least frequency are dequeued first.

After we have enqueued all of the leaf nodes, we then dequeue twice, rejoin, and enqueue the newly joined nodes, until only one node is left in the priority queue. This process will construct our Huffman tree, ensuring that leaf nodes with higher frequencies are closer to the root of the tree than leaf nodes with lower frequencies. The root node of the newly constructed tree is returned.

```
build_tree(histogram[ALPHABET]):
      pq = pq_create(ALPHABET)
```

```
    for (i = 0; i < ALPHABET; i++):
        if (histogram[i] != 0):
        enqueue(pq, node_create(i, histogram[i]))

    while (pq_size(pq) > 1):
        dequeue(pq, left)
        dequeue(pq, right)
        enqueue(pq, node_join(left, right))

    dequeue(pq, root)
    pq_delete(pq)
    return root
```

`build_codes` constructs each symbol's corresponding Huffman code by traversing the tree and utilizing a stack of code bits to remember it's path through the tree. When it reaches a leaf node, it's current code stack is added to the code table corresponding to the symbol of the leaf node. If it doesn't hit a leaf node, it pushes a 0 onto the code stack, travels left, then pops the 0 back off after returning. It then pushes a 1, travels right, then pops the 1 off after returning.

A code stack is preserved between calls using a static Code, which is zero initialized in C by default.

```
build_codes(root, table[ALPHABET]):
    static Code curr_code;
    if (root.left == NULL && root.right == NULL):
        table[root.symbol] = curr_code
        return
    else:
        code_push_bit(curr_code, 0)
        build_codes(root.left, table)
        code_pop_bit(curr_code)

        code_push_bit(curr_code, 1)
        build_codes(root.right, table)
        code_pop_bit(curr_code)

        return
```

`rebuild_tree` is used by the decoding program to rebuild the Huffman tree from the tree dump outputted by the encoding program. It iterates over the contents of the tree dump array read in from the input file and reconstructs the tree using a stack of nodes. When it encounters an 'L', it pushes a leaf node to the stack. When it encounters an 'I', it pops twice from the stack,

rejoins the two popped nodes, and pushes this joined node back to the stack. The root of the reconstructed tree will be the final node in the stack.

```
rebuild_tree(nbytes, tree[nbytes]):
      s = stack_create(nbytes)
      for (i = 0; i < nbytes; i++):
            if (tree[i] == ‘L’):
                  stack_push(s, node_create(tree[i +1]))
                  i += 1
            else if (tree[i] == ‘I’):
                  stack_pop(s, right)
                  stack_pop(s, left)
                  stack_push(s, node_join(left, right))

      stack_pop(s, root)
      stack_delete(s)
      return root
```

`delete_tree` deletes every node in our tree by doing a post-order traversal, and calling each node’s destructor.

```
delete_tree(root):
      if (root.left == NULL && root.right == NULL):
            node_delete(root)
            return
      else:
            delete_tree(root.left)
            delete(tree(root.right)
            node_delete(root)
            return
```

---

pq.c

We use a priority queue to initially construct the Huffman tree. The priority queue is a queue of Node pointers, and ensures an ordering such that nodes which are dequeued have the lowest frequency. It uses an insertion sort when enquing to ensure this ordering.

The priority queue has a field for it’s head, it’s tail, it’s size, it’s capacity, and a pointer to it’s array of node pointers. The head tracks the index of where we dequeue from. The size is the

current size of the priority queue, and the capacity is the maximum number of node pointers we can enqueue.

```
struct PriorityQueue {
      head, tail, size, capacity
      Node ** nodes
};
```

We have a constructor function, `pq_create,` to initialize the struct members and allocate the array for storing the node pointers. The function returns a pointer to the newly created priority queue. The capacity field is set to capacity + 1 because we use up one spot in the queue to check whether the queue is full or not

```
pq_create(capacity):
      pq = malloc(sizeof(PriorityQueue))
      pq.head, pq.tail, pq.size = 0
      pq.capacity = capacity + 1
      pq.nodes = calloc(pq.capacity, sizeof(Node pointer))

      return pq
```

We have a destructor function, pq_delete, to delete the priority queue, and free up all of it's memory.

```
pq_delete(pq):
      free(q.nodes)
      free(q)
      q = NULL
      return
```

We use three functions for accessing the size of the priority queue, as well as determining whether it is full or empty. We also use two functions for determining the next item and the previous item in the queue.

```
pq_empty(pq):
      if (pq.size > 0):
            false
      else:
            return true

pq_full(pq):
      // If the next position of the tail is the head, queue is full
      if (succ(pq.tail, pq.capacity) == pq.head):
```

```
            return true
      else:
            return false

pq_size(pq):
      return pq.size

succ(pos, capacity):
      return ((pos + 1) % capacity)

prev(pos, capacity):
      ((pos + capacity - 1) % capacity)
```

The enqueue function ensures that items which are dequeued from the priority queue have the lowest frequency. It returns false if the queue is full and we can't enque anything, and true otherwise. If the queue is empty, we simply enqueue at the tail, get the next index of the tail, and increment the size. If the queue is not empty, we perform an insertion sort to find the correct slot to enqueue at. We first begin at the tail. If our current slot is the head, we insert at the current slot. If the previous slot from our's has a larger frequency, then we shift that item to our current slot, and shift our current slot to that previous slot. If the previous slot doesn't have a larger frequency, we simply insert where we currently are.

```
enqueue(pq, n):
      if (pq_full(pq)):
            return false

      if (pq_empty(pq)):
            pq.nodes[pq.tail] = n
            pq.tail = succ(pq.tail, pq.capacity)
            pq.size += 1
      else:
            slot = pq.tail


            while (true):
                  if (slot == pq.head):
                        pq.nodes[slot] = n
                        pq.tail = succ(pq.tail, pq.capacity)
                        pq.size += 1
                        break
                  else if (pq.nodes[prev(slot, pq.capacity)].freq > n.freq):
                        pq.nodes[slot] = pq.nodes[prev(slot, pq.capacity)]
                        slot = prev(slot, pq.capacity)
```

```
                else:
                        pq.nodes[slot] = n
                        pq.tail = succ(pq.tail, pq.capacity)
                        pq.size += 1
                        break
        return true
```

The dequeue function simply dequeues from the head from the head of the queue. If the queue is empty, we return false. Otherwise, we return true, and pass back the popped node pointer through the pointer n. We then get the next position of the head, and decrement the size of the queue.

```
dequeue(pq, n):
        if (pq_empty(pq)):
                return false
        n = pq.nodes[pq.head]
        pq.head = succ(pq.head, pq.capacity)
        pq.size -= 1
        return true
```

The print function prints the priority queue in dequeue order by starting at the head, and printing each item until we reach the tail.

```
pq_print(pq):
        i = pq.head
        while (i != pq.tail):
                node_print(pq.nodes[i])
                i = succ(i, pq.capacity)
```

---

node.c

Our Huffman tree is composed of a Node ADT. Each node contains a pointer to it's left and right children, it's symbol, and it's frequency.

The constructor for a node allocates the memory, sets its children nodes to NULL, and sets it's frequency and symbol to the values given it. A pointer to the newly created node is returned.

```
node_create(symbol, frequency):
	n = calloc(1, sizeof(Node))
	if (n):
		n.left, n.right = NULL
		n.symbol = symbol
		n.frequency = frequency
```

The destructor deallocates the memory and sets the pointer to the node to NULL. The children nodes are not freed, but the function which deletes a tree ensures that children are always deleted before the parent.

```
node_delete(n):
	n.left, n.right = NULL
	free(n)
	n = NULL
	return
```

We use a function which takes in two pointers to nodes, and creates a new node which is a parent to both of these nodes. The symbol of this newly created parent node is always '$', and it's frequency is the sum of the two children's frequencies.

```
node_join(left, right):
	n = node_create('$', left.frequency + right.frequency)
	n.left = left
	n.right = right
	return n
```

The print function simply prints the node (if it exists) and it's symbol and frequency. It also prints its children nodes, if it has any.

```
node_print(n):
	if (n):
		print("(" + n.symbol + ", " + n.frequency + ")")
	else:
		print("NULL")
	print(" --> L: ")
	if (n.left):
		print("(" + n.left.symbol + ", " + n.left.frequency + ")")
	else:
		print("NULL")
	print(" --> R: ")
	if (n.right):
		print("(" + n.right.symbol + ", " + n.right.frequency + ")")
	else:
		print("NULL")
```

code.c

In order to construct each symbol's Huffman code, we need to traverse the Huffman tree and maintain a stack of bits representing the path taken while traversing the tree. The Code ADT has a bit vector representing the stack of bits, as well as an index to the top of the stack.

We use three helper functions for setting, clearing, and getting a specific bit of the bit vector stack.

```
code_set_bit(c, i):
      c.bits[i / 8] |= 0x1 << (i % 8)

code_clr_bit(c, i):
      c.bits[i / 8] &= ~(0x1 << (i % 8))

code_get_bit(c, i)
      return 1 & (c.bits[i / 8] >> (i % 8))
```

Our construct does not dynamically allocate any memory, and the bit vector is simply set to a predetermined maximum size. This allows us to pass the code struct by value, and also prevents us from needing any sort of destructor function.

```
code_init():
      Code c
      c.top = 0
      for (i = 0; i < MAX_CODE_SIZE; i++):
            c.bits[i] = 0

      return c
```

We use three functions for accessing the size of the code and determining whether the code is empty or full. The top field of the struct represents an index to a specific bit, so the size is measured in bits.

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

```
code_empty(c):
      if (c.top > 0):
            return false
      else:
            return true

code_full(c):
      if (c.top == ALPHABET):
            return true
      else:
            return false
```

The push function will return false if the stack is full, and true otherwise. It either sets or clears the bit at the top of the stack depending on the value of the bit we wish to push onto the stack.

```
code_push_bit(c, bit):
      if (code_full(c)):
            return false
      else:
            if (bit):
                  code_set_bit(c, c.top)
            else:
                  code_clr_bit(c, c.top)
            c.top += 1
            return true
```

The pop function returns false if the stack is empty, and true otherwise. It passes back the value of the popped bit back through the pointer bit.

```
code_pop_bit(c, bit):
      if (code_empty(c)):
            return false
      else:
            bit = code_get_bit(c, c.top - 1)
            c.top -= 1
            return true
```

The print function prints the bits of the code stack in reverse pop order. This ordering is the Huffman code, and represents the path you'd take when traversing the tree to arrive at a symbol. We start at the bottom of the stack, and print each bit until we reach the top

```
code_print(c):
      if (code_empty(c)):
            return
      else:
            for (i = 0; i < c.top; i++):
                  print(code_get_bit(c, i))
      return
```

The stack used while rebuilding the tree is implemented in a very similar way to the Code ADT, but instead of a stack of bits, it is a stack of node pointers. The only difference between the two is that the stack is dynamically allocated, so it needs to be destroyed with a destructor.

---

io.c

For reading and writing from files, we use the low-level system calls open(), read(), and write(). However, since these functions don't guarantee all of the bytes are written/read, we write wrapper functions to loop calls until we ensure all bytes are written/read out. We also implement a function to write the individual bits of our Huffman codes out. This file also has two external variables, bytes_read and bytes_written which we use to track file sizes (total bytes written/read).

The read_bytes function takes in an input file descriptor, a buffer to write to, and the number of bytes of the buffer. It loops calls to read() until it knows that it's read in the specified number of bytes. It also increments the bytes_read external variable. The function returns the number of bytes it reads. This value will be less than the number provided if it reaches the end of the input file.

```
read_bytes(infile, buffer, nybytes):
      current_bytes_read, bytes = 0
      while(true):
            if ((bytes=read(infile, buffer+current_bytes_read, nbytes))<=0):
                  break

            current_bytes_read += bytes
            bytes_read += bytes

            if (current_bytes_read == nbytes):
                  break

      return current_bytes_read
```

The write_bytes function takes in a buffer and it's size, and writes it's contents to the provided output file, once again using looped calls to write() to ensure everything is written out. It returns the number of bytes it wrote out, and increments the external variable bytes_written.

```
write_bytes(outfile, buffer, nbytes):
	current_bytes_written, bytes = 0
	while(true):
		if ((bytes=write(outfile, buf+current_bytes_written,
nbytes))<=0):
			break

		current_bytes_written += bytes
		bytes_written += bytes

		if (current_bytes_written == nbytes):
			break
	return current_bytes_written
```

The read_bit function will read in a single bit of the input file and return its value back through the pointer bit. The function returns false if there are no more bits to be read in, and false otherwise. The function buffers the bits in a static buffer, and doles out the bit values of this buffer until it is empty. When it is empty, the buffer is refilled. A static index is also used to keep track of where we are in the buffer. This index is an index to a bit.

```
read_bit(infile, bit):
	if (index == 0):
		if (read_bytes(infile, buffer, BLOCK) == 0):
			return false

	bit = buffer_bet_bit(index)
	index += 1

	if (index == BLOCK * 8):
		index = 0

	return true
```

Since our buffer is a bit vector, we also need functions for setting, clearing, and getting individual bits of the buffer.

```
buffer_set_bit(i):
	buffer[i / 8] |= 0x1 << (i % 8)
```

```
buffer_clr_bit(i):
      buffer[i / 8] &= ~(0x1 << (i % 8))

buffer_get_bit(i):
      return 1 & (buffer[i / 8] >> (i % 8))
```

The write_code function functions in a similar way to the read_bit function. It accumulates the bits of the code we wish to write out in a buffer. When the buffer is full, it writes out it's contents with write_bytes. We also need a function, flush_codes, to write out any remaining bits left in our buffer, since the write_code function only writes out it's contents when it is full.

```
write_code(outfile, c):
      i = 0
      while (i < code_size(c)):
            bit = 1 & (c.bits[i / 8] >> (i % 8))
            if (bit):
                  buffer_set_bit(index)
            else:
                  buffer_clr_bit(index)

            i += 1
            index += 1

            if (index >= BLOCK * 8):
                  write_bytes(outfile, buffer, BLOCK)
                  index = 0


      return

flush_codes(outfile):
      byte_num = (index - 1) % 8 == 0 ? (index - 1) / 8: ((index - 1) / 8) +
1
      write_bytes(outfile, buffer, byte_num)
      index = 0
      return
```

---

decode.c

The decoding program will read in the file produced by the encoding function, reconstruct the Huffman tree, and traverse the tree in order to decode the Huffman codes into the original message.

The program has arguments to specify a path to an input file and an output file. By default, we read/write from STDIN and STDOUT. There is also an option for printing the decompression statistics, including the compressed file size, the decompressed file size, and the space saving percentage.

Just as in the encoding program, if an infile and an outfile is provided, the permissions of the output file should match that of the input.

After processing the command line arguments, we first verify that we can actually decode the file. This is done by reading in the header, and checking the header magic number. If it does not match, we exit the program.

Next, we read in the tree dump, first storing it in an array, and then passing this array to the reconstruct_tree function, which will return the root of the newly created tree.

Next, we read in each bit left in the input file, traversing the tree depending on this bit. If we read in a 0, we traverse left. If we read in a 1, we traverse right. If we reach a leaf, we write out the symbol of this leaf, and set the current node back to the root. The write calls are also buffered in an array. This process ends once we've written out the number of bytes of the original file size, specified in the header we read in.

Then, if the verbose flag was supplied, we print the statistics, deallocate all of the memory, and close the files.

```
main(argc, argv):
      HELP, VERBOSE = false
      infile_name, outfile_name = NULL
      infile = STDIN
      outfile = STDOUT

      while ((arg = get_arg(argc, argv)) != -1):
            if arg == ‘h’:
                  HELP = true
            if arg == ‘v’:
                  VERBOSE = true
            if arg == ‘i’:
                  infile_name = name provided
            if arg == ‘o’:
```

```
            outfile_name = name provided

if (HELP):
      Print help and program usage
      return 0

if (infile_name != NULL):
      infile = open(infile_name)

if (outfile_name != NULL):
      outfile = open(outfile_name)
      outfile.permissions = infile.permissions

read_bytes(infile, header, sizeof(Header))
if (header.magic != MAGIC):
      print("Invalid magic number!")
      exit

tree_dump = calloc(header.tree_size)
read_bytes(infile, tree_dump, header.tree_size)
root = rebuild_tree(header.tree_size, tree_dump)

out_buf = calloc(header.file_size)

curr_node = root
symbols_written = 0
while (true):
      read_bit(infile, bit)

      if (bit):
            curr_node = curr_node.left
      else:
            curr_node = curr_node.right

      if ((curr_node.left == NULL) || (curr_node.right == NULL)):
            out_buf[symbols_written] = curr_node.symbol
            symbols_written += 1
            curr_node = root

      if (symbols_written == header.file_size):
            break

write_bytes(outfile, out_buf, header.file_size)
```

```
if (VERBOSE):
      print compressed file size
      print decompressed file size
      print space saving
return
```