Assignment 6: *Design*

Sriramya Prayaga

March 2, 2022

Abstract

In this document, I will be describing my design process for Assignment 6. Assignment 6 is essentially implementing file compression using Huffman's algorithm. The files we are supposed to create for this assignment are node.c, pq.c, code.c, io.c, stack.c huffman.c, encode.c, decode.c (described in more detail below.)

1 Pseudocode/Structure

1.1 node.c

The first file we are supposed to implement –node.c– creates a data structure that keeps track of a symbol and its frequency. The definition of the Node structure as well as 4 functions are in this file: Node *node_create(uint8_t symbol, uint64_t frequency), void node_delete(Node **n), Node *node_join(Node *left, Node *right), void node_print(Node *n). The pseudocode is included down below. node_create() creates the node, node_delete() frees node memory, node_join() joins two nodes together, and print_node() checks to see if everything worked the way it should.

```
Node *node_create(uint8_t symbol, uint64_t frequency) {
         allocate memory for a Node node

         if node pointer exists {
           set node symbol to symbol
           set node left to NULL
           set node right to NULL
         }
         return node
}

void node_delete(Node **n) {
    free n
    set n to NULL
}

Node *node_join(Node *left, Node *right) {
    set Node *parent_node to node_create('$', (left->frequency + right->frequency))
    set parent_node left to left
    set parent_node right to right
    return parent_node
}

void node_print(Node *n) {
    print(n symbol and n frequency)
    print(n left symbol and frequency)
    print(n right symbol and n right frequency)
}
```

1.2 pq.c

The next file contains a data structure and 8 functions to create a priority queue. The data structure consists of two head and tail nodes, a uint32_t size value, an array of node pointers, and a uint32_t capacity value. The priority queue is used to de-queue the node with the smallest frequency first (highest priority nodes are the nodes with the smallest frequency, and they will be located at the tail.) The functions used in this file are pq_create(), pq_delete(), pq_empty(), pq_full(), pq_size(), pq_enqueue(), pq_dequeue(), pq_print(). pq_create() creates and initializes a priority queue, pq_delete() frees the memory used for the priority queue, pq_empty() checks if the queue is empty, pq_full() checks if the queue is full, pq_size() gives the size of the queue, pq_enqueue() enqueues a node in the correct position based on its frequency (using insertion), pq_dequeue() dequeues the node at the tail, and pq_print() is a de-bugger function. The pseudocode is given below.

```
PriorityQueue *pq_create(uint32_t capacity) {
   allocate memory to create a priority queue called pq that can hold
   capacity number of nodes
   if pq was created properly {
       set pq head to 0
       set pq tail to 0
       set pq size to 0
       set pq capacity to capacity
       dynamically allocate memory for an array of node pointers
       set pq items to dynamically allocated array
       return pq
   } else {
       return a NULL pointer
   }
}

void pq_delete(PriorityQueue **q) {
   if *q exists {
      free *q->items
      set *q->items to NULL
      free(*q)
      set (*q) = NULL
   }
}

bool pq_empty(PriorityQueue *q) {
   if pq has no nodes {
       return true
   } else {
       return false
   }
}

bool pq_full(PriorityQueue *q) {
   if pq has capacity number of items {
      return true
   } else {
      return false
   }
}

uint32_t pq_size(PriorityQueue *q) {
   returns pq size
}

bool enqueue(PriorityQueue *q, Node *n) {
  set Node *temp = n
  set j = pq_size(q) +q->tail
  set k = pq->size
  if n is NULL {
     return false
  }
  if pq is full {
     return false
  }
  if pq is empty {
     set q->items[q->tail] = temp
  } else {
     while (temp’s frequency < q->items[j-1]->frequency) {
        set q->items[j] = q->items[j - 1]
        decrease j,k by 1
     }
  }
  increase q->size by 1
  set q->head = q->head + 1
  return true
}
```

```
bool dequeue(PriorityQueue *q, Node **n) {
   if q is empty {
      return false
   }
   set *n to q->items[q->tail]
   increment q-> tail by 1
   decrease q->size by 1
   return true
}
void pq_print(PriorityQueue *q) {
   print all values in priority queue tree
}
```

1.3 code.c

The next file includes a structure called code, and 10 functions: code_init(), code_size(), code_empty(), code_full(), code_set_bit(), code_clr_bit(), code_get_bit(), code_push_bit(), code_pop_bit(), void code_print(). This file creates codes for each symbol in the input text file. The structure for a code consists of a uint32_t top value, and a uint8_t bits value. This file constructs a code for each symbol in the text. code_init() initializes the values in the code structure, code_size() gives the size of a code, code_set_bit() and code_clear_bit() set and clear a bit at an index, code_push_bit pushes a bit, code_pop_bit pops a bit at an index and return the value. code_print() is a debugger function. The pseudocode for this file is defined below:

```
Code code_init(void) {
   create a Code code
   set top to 0
   make values in bits[] equal 0
   return code
}

uint32_t code_size(Code *c) {
   return number of bits pushed to to bits[]
}

bool code_empty(Code *c) {
   if code_size(c) is 0 {
     return true
   } else {
     return false
   }
}

bool code_full(Code *c) {
   if code_size(c) is 256 {
     return true
   } else {
     return false
   }
}

bool code_clr_bit(Code *c, uint32_t i) {
    if i > 255 {
       return false
    }
    take a Code with all bits but the bit at index i set to 1
    (done through left-shifting the value)
    AND that value with c
    return true
}

bool code_set_bit(Code *c, uint32_t i) {
    if i > 255 {
       return false
    }
    take a Code with all bits but bit at index i set to 0
    (done through left-shifting the value)
    OR that Code value with c
    return true
}

bool code_get_bit(Code *c, uint32_t i) {
    if i > 255 {
       return false
    }
    take a Code with all bits but bit at index i set to 0
    (through left-shifting the value)
    AND that Code value with c
    right-shifting that value and OR it with 1
    return true
}

bool code_push_bit(Code *c, uint8_t bit) {
    if code is full {
      return false
    }
    set c->bit[s->top] = bit
    set top += 1
    return true
}
```

```
bool code_pop_bit(Code *c, uint8_t *bit) {
   if code is empty {
      return false
   }
   set top -= 1
   set *bit = bit[s->top]
   return true
}

void code_print(Code *c) {
   print(value of pushed bits, and popped bits)
}
```

1.4 io.c

In this file there will be five functions, and two external variables declared. The five functions are int read_bytes(), write_bytes(), read_bit(), write_code(), flush_codes(). The two extern variables are bytes_read, and bytes_written. read_bytes() reads nbytes number of bites from a file into a buffer, write_bytes() writes bytes in a buffer to a file, read_bit reads one bit from a file and returns it. write_code() writes a code to a file, and flush_codes() checks to see if there are any remaining bytes in a buffer that need to be written. This file implements functions that read from and write to files for various purposes. The pseudocode for this file is below:

```
create a static buffer[block]
create a static index

int read_bytes(int infile, uint8_t *buf, int nbytes) {
    do {
       set bytes =  read(infile, buf + offset, nbytes - offset)
       if (bytes ==-1) {
        exit loop
       }
    } while (number of bytes left to read are greater than 0 or bytes read < nbytes)
    return number of bytes read from infile
}

int write_bytes(int outfile, uint8_t *buf, int nbytes) {
    do {
       set bytes = write(infile, buf, nbytes)
       if (bytes ==-1) {
        exit loop
       }
    } while (number of bytes left to write are greater than 0 or bytes != 0)
    return number of bytes written to outfile
}
```

```
bool read_bit(int infile, uint8_t *bit) {
    if (bytes_read = 0) {
        set bytes = read_bytes(infile, buffer2, 4096)
    }
    set file_end = (BLOCK * 8)
    if (bytes < BLOCK) {
        file_end = number of bits read in from the file
    }
    get bit at particular global index
    increment index

    if index != file_end {
       return true
    } else {
       return false
    }
}

void write_code(int outfile, Code *c) {
    create a buffer2[] to keep track of all bytes in a file
    set c->bits = buffer2
    create local index = 0 to keep track of bits

    if (index == 8 * sizeof(buffer) {
        set index = 0
    }
    do {
        get a bit from the code
        set the bit at the corresponding index
        increment local, global indexes
    } while (index != sizeof(buffer) and local index < c->top)

    if (index = 8 *sizeof(buffer) ) {
        call flush_codes(outfile)
        index = 0
    }

}

void flush_codes(int outfile) {
    if there are more bits in buffer {
        clear the excess bits in the last byte
        call write_bytes(outfile, buffer, bytes left to print)
    }
}
```

1.5 stacks.c

The next file implements a stack module (consisting of a stack structure, and 8 functions). The stack structure consists of two uint32_t top and capacity values, as well as a double pointer to a Node items. This file implements a stack that will be used for reconstructing a Huffman tree. The functions used are stack_create(), stack_delete(), stack_empty(), stack_full(), stack_size(), stack_push(), stack_pop(), stack_print(). stack_create() creates and initializes a stack structure, stack_delete() frees memory used by a stack. stack_empty(), and stack_full returns if a stack is empty or full, respectively. stack_push() and stack_pop() push and pop from a stack, respectively. stack_size() returns the size of a stack. The pseudocode/structure for this file is as shown below:

```
Stack *stack_create(uint32_t capacity) {
    dynamically allocate memory for a Stack called stack with a
    capacity number of nodes
    if stack is not NULL {
        set top of stack to equal 0
        set stack capacity to equal capacity
        dynamically allocate memory for an array of node pointers
        set stack items equal to array of node pointers
        return stack
    }
    else {
        return NULL pointer
    }
}

void stack_delete(Stack **s) {
     if *s is not NULL {
        free stack items array
        set items array to NULL
        free stack
        set stack to NULL
     }
}

bool stack_empty(Stack *s) {
     if top of the stack = 0 {
        return true
     } else {
        return false
     }
}
```

```
bool stack_full(Stack *s) {
     if top of the stack = capacity {
        return true
     } else {
        return false
     }
}

uint32_t stack_size(Stack *s) {
      return s -> top
}

bool stack_push(Stack *s, Node *n) {
      if top of stack = capacity {
         return false
       }
      set s->items[s->top] to n
      set s->top += 1
      return true
}

bool stack_pop(Stack *s, Node **n) {
      if stack is empty {
         return false
      }
      set s->top -= 1
      set n = s->items[s->top]
      return true
}

void stack_print(Stack *s) {
     prints(all data items in the stack)
}
```

1.6 huffman.c

In this file, A Huffman coding module is implemented. It has five functions: build_tree(), build_codes(), dump_tree(), rebuild_tree(), and delete_tree(). build_tree() builds a Huffman tree, build_codes() builds a code table, dump_tree() dumps a tree, rebuild_tree() reconstructs a Huffman tree, and delete_tree() frees the memory used by a Huffman tree. The pseudocode for this file is given below:

```
Node *build_tree(uint64_t hist[static ALPHABET]) {
    while length of hist > 1 {
       set left = dequeue(hist)
       set right = dequeue(hist)
```

```
        set parent = node_join(left,right)
        call enqueue(hist, parent)
    }
    set root = dequeue(hist)
    return root
}

void build_codes(Node *root, Code table[static ALPHABET]) {
    create uint8_t *bit, *bit2
    if root doesn’t equal NULL {
        if node->left and node->right are NULL {
           set table[node->symbol] =  to value of code_init()
        } else {
           call code_push_bit(c, 0)
           call build_codes(node->left, table);
           call code_pop_bit(c, bit)

           call code_push_bit(c, 1)
           call build_code(node->right, table)
           call code_pop_bit(c, bit2)
        }
    }
}

void dump_tree(int outfile, Node *root) {
    if root is not NULL {
         dump_tree(outfile, root->left)
         dump_tree(outfile, root->left)

         if root->left and root-> right are not NULL {
             set buffer[0] = ’L’
             set buffer[1] = root->symbol
             call write_bytes(outfile, buffer, 1)
             call write_bytes(outfile, buffer, 1)
         } else {
             set buffer[0] = ’I’
             call write_bytes{outfile, buffer, 1}
         }
    }
}

Node *rebuild_tree(uint16_t nbytes, uint8_t tree[static nbytes]) {
    set s = stack_create(nbytes)
    for all values in tree {
        if tree[value] = ’L’ {
            set node = node_create(tree[value + 1], tree[value + 2])
            stack_push(s, node)
        } else if tree[value] = 'I' {
            call stack_pop(s, node1)
            call stack_pop(s, node2)
            call set nodep = node_join(node2, node1)
            call stack_push (s, nodep)
    }
    }
    stack_pop(s, root)
    return root of the tree
}

void delete_tree(Node **root) {
   if (root is not NULL) {
    call delete_tree(&(root)->left)
    call delete_tree(&(root)->right)
    free root
    set root to NULL
   }

}
```

1.7 encode.c

This file contains the Huffman encoder, and it compresses the file using the functions and files above. The pseudocode for this file is as shown below.

```
int main(int argc, char ** argv} {
    initialize default variables
    create a table array

    parse command-line options with getopt() and switch() statements
    to determine verbose mode, input and output files, and usage message
            display usage message if option -h was chosen

    if given infile is not seekable {
        create a temporary file
        copy contents of infile to temporary file by calling read_bytes
        and write_bytes in a while-loop
        close infile
        set infile = temporary file
    }
    call read_bytes(infile, buf, file_length)
    create a histogram called hist[] to store frequency of each character
    to infile
    increment index 0 and 255 in hist[] by 1

    set tree = build_tree(hist[ALPHABET])
    call build_codes(root, table)

    allocate memory for a header struct
    set header->magic to MAGIC
    set header->permissions to permissions of infile
    set header->tree_size to 3 * (unique histogram symbols) - 1
    set header->file_size to value of fstat

    call write_bytes(outfile, (uint8_t) pointer to header, size of header)

    call dump_tree(outfile, root)

    for all values in the code table {
        call write_codes(outfile, table[code index])
    }

    call flush_codes(outfile)

    close infile, outfile

}
```

1.8 decode.c

This file decompresses a compressed infile, and writes the original file contents to outfile. The pseudocode for this file is as shown below.

```
int main(int argc, char ** argv} {
    initialize default variables
    create an array called tree

    parse command-line options with getopt() and switch() statements
    to determine verbose mode, input and output files, and usage message
            display usage message if option -h was chosen

    call read_bytes(infile, (uint8_t) pointer to header, size of header)
    to read in header

    if magic number is incorrect {
       print error message and exit program
    }
    call rebuild_tree(header.tree_size, tree)
```

```
while decoded symbols != file size {
 call read_bit(infile, a pointer to a bit) for all bits in file
 if read bit value is 0 {
   go to current node's left child
 } else {
   go to current node's right child
 }

 if there is a leaf node {
   call write_bytes() to write node's symbol
   go back to root
 }
}

close infile and outfile
```

2 Summary

There are 8 files we needed to complete to accomplish Huffman's file compression: node.c, pq.c, code.c, io.c, stack.c, huffman.c, encode.c,and decode.c. We were given header files for all of these files except for encode.c and decode.c. We were also given a defines.h file that included the various definitions needed in this assignment such as the BLOCK size, ALPHABET size, MAGIC number value, MAX_CODE_SIZE value, and MAX_TREE_SIZE value.