Assignment 7 Design Doc

In this assignment the main goal is to create an encoder that encodes files using huffman encoding and a decoder that decodes a file that has already been encoded.

node.c

```
node_create(symbol, frequency):
        Node n = allocate memory for Node
        n.symbol = symbol
        n.frequency = frequency'
        return n
node_delete(**n):
        free(*n)
        n = NULL
node_join(left, right):
        Node n = allocate memory for Node
        n.symbol = '$'
        n.frequency = left.frequency + right.frequency
        n.left = left
        n.right = right
        return n
node_print(n):
        print n.symbol, and n.frequency
node_cmp(n1, n2):
        if (n1.frequency > n2.frequency): return True
        return False
node_print_sym(n):
        print n.symbol
```

pq.c

```
pq_create(capacity):
        PriorityQueue pq = allocate memory for PriorityQueue
        pq.capacity = capacity
        pq.size = 0
        pq.queue = allocate enough memory for capacity number of nodes
        return pq
pq_delete(**q):
        free(*q)
        q = NULL
pq_empty(q):
```

```
        if (q.size == 0): return True
        return False
pq_full(q):
        if (pq.size == pq.capacity): return True
        return False
pq_size(q):
        return pq.size
enqueue(q, n):
        if (pq_full(q)): return False
        q.size += 1
        if (pq_empty(q)):
                q.queue[0] = n
                return True
        for i from 0 to q.size-1:
                if q.queue[i].frequency > n.frequency:
                        continue
                break
        for j from q.size-1 to i:
                q.queue[j+1] = q.queue[j]
        q.queue[i] = n
        return True
dequeue(q, n):
        if (pq_empty(q)): return False
        q.size -= 1
        n  = q.queue[q.size]
        return True
pq_print(q):
        for i from q.size to 0:
                node_print(q.queue[i])
```

code.c

```
code_init():
        Code c
        c.top = 0
        for i from 0 to MAX_CODE_SIZE:
                c.bits[i] = 0
        return c
code_size(c):
        return c.top
code_empty(c):
        if (c.top == 0): return True
        return False
code_full(c):
```

```
        if (c.top == 255): return True
        return False
code_set_bit(c, i):
        if i > 255 or i < 0: return False
        c.bits[i/8] bitwise or with 1 left shifted i%8 times
        return True
code_clr_bit(c, i):
        if i > 255 or i < 0: return False
        c.bits[i/8] bitwise and with inverse of 1 left shifted i%8 times
        return True
code_get_bit(c, i):
        if i > 255 or i < 0: return False
        if (c.bits[i/8] bitwise and with 1 left shifted i%8 times right shifted i%8 times): return True
        return False
code_push_bit(c, bit):
        if (code_full(c)): return False
        if (bit):
                code_set_bit(c, c.top)
        else:
                code_clr_bit(c, c.top)
        c.top += 1
        return True
code_pop_bit(c, bit):
        if (code_empty(c)): return False
        c.top -= 1
        if (code_get_bit(c, c.top)): bit = 1
        else: bit = 0
        code_clr_bit(c, c.top)
        return True
code_print(c):
        for i from 0 to MAX_CODE_SIZE:
                print code_get_bit(c, i)
```

io.c

```
read_bytes(infile, buf, nbytes):
        total = 0
        while nbytes != total:
                result = read(infile, buf, nbytes-total)
                total += result
                if result == 0: break
        bytes_read += total
        return total
write_bytes(outfile, buf, nbytes):
```

```
        total = 0
        while nbytes != total:
                result = write(infile, buf, nbytes-total)
                total += result
                if result == 0: break
        bytes_written += total
        return total
read_bit(infile, bit):
        if (index == 0):
                bytes = read_bytes(infile, buffer, BLOCK)
        bit = buffer[index/8] bitwise and with 1 left shifted index%8 times right shifted index%8
times
        if (index == BLOCK*8):
                index = 0
                return True
        if (index == bytes*8):
                return False
        index += 1
        return True
write_code(outfile, c):
        i = 0
        for j from 0 to code_size(c):
                if (code_get_bit(c, i)):
                        buffer[i/8] bitwise or with 1 left shifted i%8 times
                else:
                        buffer[i/8] bitwise and with inverse of 1 left shifted i%8 times
                i += 1
                if (j % BLOCK*8 == 0):
                        write_bytes(outfile, buffer, BLOCK)
                        i = 0
                        for k from 0 to BLOCK:
                                buffer[k] = 0
flush_codes(outfile):
        write_bytes(outfile, buffer, BLOCK)
```

stack.c

```
stack_create(capacity):
        Stack s = allocate memory for stack
        s.capacity = capacity
        s.top = 0
stack_delete(**s):
        free(*s)
        s = NULL
```

```
stack_empty(s):
	if (s.top == 0): return True
	return False
stack_full(s):
	if (s.top == s.capacity): return True
	return False
stack_size(s):
	return s.top
stack_push(s, n):
	if (stack_full(s)): return False
	s.items[top] = n
	top += 1
	return True
stack_pop(s, n):
	if (stack_empty(s)): return False
	n = s.items[top-1]
	top -= 1
	return True
stack_print(s):
	for i from 0 to top:
		node_print(s.items[i])
```

huffman.c

```
build_tree(hist):
	create PriorityQueue with hist
	while pq_size(q) > 1:
		left
		dequeue(q, left)
		right
		dequeue(q, right)
		parent = node_join(left, right)
		enqueue(q, parent)
	root = dequeue(q)
	return root
c = code_init()
build_codes(root, table):
	if root != NULL:
		if not root.left and not node.right:
			table[node.symbol] = c
		else:
			push_bit(c, 0)
			built(root.left, table)
			pop_bit(c)
```

```
                push_bit(c, 1)
                built(root.right, table)
                pop_bit(c)
dumb_tree(outfile, root):
    if root:
        dump(outfile, root.left)
        dump(outfile, root.right)
        if not root.left and not root.right:
            write('L')
            write(node.symbol)
        else:
            write('I')
rebuild_tree(nbytes, tree_dump):
    Stack s = stack_create(nbytes)
    for i from 0 to length of tree_dump:
        if tree_dump[i] = 'L':
            i += 1
            stack_push(s, tree_dump[i])
        if tree_dump[i] = 'I':
            Node right
            stack_pop(s, right)
            Node left
            stack_pop(s, left)
            joined = node_join(left, right)
            stack_push(joined)
    Node root
    stack_pop(s, root)
    return root
delete_tree(**root):
    delete_tree(&(*root.left))
    delete_tree(&(*root.right))
    node_delete(*root)
    root = NULL
```

encode.c

infile = stdin
outfile = stdout
stats = False
take command arguments and run associated code:
-h: print help message
-i: infile = optarg
-o: outfile = optarg
-v: stats = True

```
histogram[256]
read through infile:
	histogram[symbol] += 1
if histogram[0] = 0:
	histogram[0] = 1
if histogram[1] = 0:
	histogram[1] = 1
root = build_tree(histogram)
table[]
build_codes(root, table)
Header h
h.magic = MAGIC
h.permissions = get permissions with fstats
fchmod(outfile, h.permissions)
h.tree_size = (3 * number of unique symbols) - 1
h.file_size = get number of bytes with fstats
write header to outfile
dump_tree(outfile, root)
for each symbol in infile:
	c = code for symbol
	write_code(outfile, c)
flush_codes()
if (stats): print statistics
close files
```

decode.c

```
infile = stdin
outfile = stdout
stats = False
take command arguments and run associated code:
-h: print help message
-i: infile = optarg
-o: outfile = optarg
-v: stats = True

read header from infile
if h.magic != MAGIC:
	print(“invalid file”)
	exit
fchmod(outfile, h.permissions)
read into array tree_dumb h.tree_size bytes long
root = rebuild_tree(h.tree_size, tree_dumb)
```

run through infile with read_bit():
    if 0 traverse down tree to left
    if 1 traverse down tree to right
    if node is leaf write symbol to outfile and set current node back to root
close files