Assignment 7

A (Huffman Coding) Tree Grows in Santa Cruz

Description of Assignment:

In this assignment, we will implement a Huffman encoder and decoder. The Huffman encoder will read an input file, find its Huffman encoding, and use the encoding to compress the file. The decoder will take in a compressed file and decompress it, bringing it back to its original size. We will also create 4 abstract data types(ADT): nodes, priority queues, codes, and stacks(similar to the last assignment). Huffman trees are made of nodes, and each node contains a pointer to its left and right child, a symbol, and the frequency of the symbol. In the encoder, we will make a priority queue of nodes. A priority queue is similar to a regular queue but it assigns each element a priority so that elements with higher priority are dequeued first. After creating the tree, we will traverse the tree in order to create a code for each symbol. The Code ADT will represent a stack of bits. Furthermore, we will create an I/O module to implement low-level system calls like read(), write(), open(), and close(). The final ADT will be a stack since we need a stack of nodes in our decoder to reconstruct a Huffman tree. Lastly, there will also be a Huffman Coding Module which will build the tree and codes, dump the tree, and rebuild the tree.

Files to be included:

1. encode.c: This file will contain your implementation of the Huffman encoder.
2. decode.c: This file will contain your implementation of the Huffman decoder.
3. defines.h: This file will contain the macro definitions used throughout the assignment.
4. header.h: This will contain the struct definition for a file header.
5. node.h: This file will contain the node ADT interface.
6. node.c: This file will contain your implementation of the node ADT.
7. pq.h: This file will contain the priority queue ADT interface.
8. pq.c: This file will contain your implementation of the priority queue ADT.
9. code.h: This file will contain the code ADT interface. This file will be provided.
10. code.c: This file will contain your implementation of the code ADT.
11. io.h: This file will contain the I/O module interface. This file will be provided.
12. io.c: This file will contain your implementation of the I/O module.
13. Stack.h: This file will contain the stack ADT interface.
14. Stack.c: This file will contain your implementation of the stack ADT.
15. huffman.h: This file will contain the Huffman coding module interface.
16. huffman.c: This file will contain your implementation of the Huffman coding module interface

Pseudocode/Structure:

Node.c:

node_create(symbol, frequency)
Allocate memory for node
Initialize node symbol to symbol
Initialize node frequency to frequency
Initialize left node to NULL
Initialize right node to NULL

node_delete(**n)
Free memory allocated for node
Set node pointer to NULL

node_join(left, right)
Create a parent node which has symbol $ and whose frequency is sum of left and right node
Set the parent left node to left
Set the parent right node to right

node_print(n)
Print out the node symbol and frequency
Node symbol is printed as a hex if it is not control character and not printable

node_cmp(n, m)
Return true if the frequency of the first node is greater than second node
Else, return false

node_print_sym(n)
Print the node symbol

Code.c:

Code code_init(void)
Set top to 0. This is the top of the array of bits
Loop through the code size
    Initialize each bit to 0

code_size(c)
Return the top of code

code_empty(c)
If the top of code is 0
    Return true
Else, return false

code_full(c)
If the top of code is 256(ALPHABET)
    Return true
Else,
    Return false

code_set_bit(c, i)
If i is over 256
    Return false because i is out of range
Else,
    Perform OR operation between the bit left shifted by 1 and the byte

code_clr_bit(c, i)
If i is over 256
    Return false because i is out of range
Else,
    Perform AND operation between the NOT of bit left shifted by 1 and the byte

code_get_bit(c, i)
If i is over 256
    Return false because i is out of range
Else,
    Perform AND operation between 1 and bit right shifted with the byte and set it to a int
    If the int is 1
        Return true
    Else
        Return false

code_push_bit(c, bit)
If the code is already full
    Return false
Else
    If the bit is 1 then we push it and set it
        Set the bit at the top to bit. Can call code set bit
    Increment top by 1
    Return true

code_pop_bit(c, *bit)
If the code is empty
    Return false
Else
    Set the pointer to bit to the bit at the top. Can call code get bit

Clear the bit at the top
Decrement top by 1
Return true

code_print(c)
Print out each bit in code

Stack.c:

stack_create(capacity)
Allocate memory for stack
Initialize top of stack to 0
Initialize capacity of stack to capacity
Allocate memory for items the stack can hold based on the capacity

stack_delete(s)
Free memory allocated for stack
Set pointer to stack to NULL

stack_empty(s)
If stack top is equal to 0
Return true
Else,
Return false

stack_full(s)
If stack top is equal to capacity
Return true
Else,
Return false

stack_size(s)
Return stack top

stack_push(Stack s, Node n)
If stack is full
Return false
Else,
Set the top of stack items to n
Increment top by 1

Return true

stack_pop(Stack s, Node *n)
If stack is empty
    Return false
Else,
    Decrement top by 1
    Set the pointer to n to the top of stack items
    Return true

stack_print(Stack s)
Print the stack

Pq.c:

pq_create(capacity)
Allocate memory for priority queue
Initialize priority queue capacity to capacity
Initialize priority queue size to 0
Allocate memory for items in priority queue based on capacity

pq_delete(pq)
Free memory allocated for priority queue
Set pointer to priority queue to NULL

pq_empty(pq)
If pq size is equal to 0
    Return true
Else,
    Return false

pq_full(pq)
If pq size is equal to pq capacity
    Return true
Else,
    Return false

pq_size(pq)
Return pq size

swap(node x, node y)
Set temp node to node x
Set node x to node y

Set node y to the temp node

l_child(n)
    Return (2 * n) + 1

r_child(n)
    Return (2 * n) + 2

parent(n)
    Return (n - 1) / 2

up_heap(Node items, n)
Keep looping while n is greater than 0 and the nth element is less than its parent node
    Swap the nth element with its parent node
    Set the nth element as the parent node

down_heap(Node items, heap_size)
Set a variable n to 0
    Loop while the left child is less than the size of the entire heap
        If there is no right child
            Set the left child as the smaller one
        Otherwise, if the left child is less than the right child
            Set the left child as the smaller one
        If the left child is not less than the right child
            Set the right child as the smaller one
        If the nth element is greater than the smaller element
            Stop looping
        Swap the nth element with the smaller element
        Set variable n to the smaller element

enqueue(pq, Node n)
If pq full is true
    Return false
Else,
    Set the top of the list of items to node n
    Call up_heap on list of nodes and top of pq
    Increment top of q
    Return true

dequeue(pq, Node *n)
If pq empty is true
    Return false
Else,
    Set node pointer to first index of queue, the top of queue
    Set the top of queue to the last node in the queue
    Call down_heap on list of nodes and q size
    Decrement q size
    Return true

io.c:

Define bytes_read
Define bytes_written

int read_bytes(int infile, uint8_t *buf, int nbytes)
Declare a variable to track the bytes read from read()
Declare a variable to track total bytes read
    Read in BLOCK bytes from infile into the buffer
    Add the bytes read to total read
    If total read equals nbytes then break
    Keep looping while read() doesn't return 0

Add total bytes read to bytes_read
Return total read

int write_bytes(int outfile, uint8_t *buf, int nbytes)
Declare a variable to track the bytes wrote from write()
Declare a variable to track total bytes wrote
    write in BLOCK bytes from buffer into outfile
    Add the bytes wrote to total wrote
    If total wrote equals nbytes then break
    Keep looping while write() doesn't return 0

Add total bytes wrote to bytes_written
Return total wrote

int read_bit(int infile, uint8_t *bit)
Declare a static buffer for the bytes read
Declare a static int variable for index into the buffer. Set index to BLOCK * 8
Declare a variable to track size of buffer that is read

If index all bits in the buffer have been visited
    Fill up the buffer again with BLOCK
    Reset index to 0
If index reached end of bytes to read
    Return false because there are no more bytes to read

Set pointer to bit by getting bit at the index
Increment index
Increment index by 1
Return true

void write_code(int outfile, Code *c)
Loop through code_size
Get the bit
If the bit is equal to 1
Set the bit in the buffer
Else
Clear the bit

If index is equal to BLOCK * 8
Write the buffer to outfile with either write_bytes
0 out the buffer
Reset index to 0

void flush_codes(int outfile)
If index is greater than 0
Convert index into bytes
Write leftover bytes to outfile using write_bytes

Huffman.c:

Node *build_tree(uint64_t hist[static ALPHABET])
PYTHON PSUEDOCODE GIVEN TO BUILD TREE

```
def construct(q):
    while len(q) > 1:
        left = dequeue(q)
        right = dequeue(q)
        parent = join(left, right)
        enqueue(q, parent)
    root = dequeue(q)
    return root
```

Create a priority queue
Loop through histogram
If the frequency is greater than 0
Create a node
Enqueue node to priority queue
While length of priority queue is greater than 1
Dequeue the left node
Dequeue the right node

Join the two nodes
Enqueue the joined node to priority queue

There will only be one node left in the priority queue
Dequeue the node and return it

void build_codes(Node *root, Code table[static ALPHABET])
PYTHON PSUEDOCODE GIVEN TO BUILD CODE

```
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)
```

void dump_tree(int outfile, Node *root)
PYTHON PSUEDOCODE GIVEN DUMP TREE

```
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')
```

Node *rebuild_tree(uint16_t nbytes, uint8_t tree[static nbytes])
Iterate over tree_dump from 0 to nbytes
If element is L

Node_create with next element since it is a symbol
Push created node to stack
If element is I
Pop stack to get right child
Pop stack to get left child
node_join(left, right)
Push joined node to stack

<u>void delete_tree(Node **root)</u>
Free all nodes
Free all allocated memory
Set pointer to NULL

<u>Huffman Encoder:</u>

While command line options given
Case ‘h’: print help message
Case ‘i’: takes input file to encode
Case ‘o’: takes output file to write compressed input into
Case ‘v’: prints compression statistics to stderr
Create a histogram to store 256 uin64_t’s
Loop through histogram and 0 out the frequencies
Create a buffer of size BLOCK
Loop through buffer and 0 out the bits
Read in BLOCK to buffer
Loop through bytes read in
Set buffer contains the symbol for histogram and increment the frequency of the symbol

After going through infile, reset the cursor to start of infile
Set the zeroth index of histogram to 1
Set the first index of histogram to 1
Call build tree to create a huffman tree
Initialize a code table to hold 256 codes
Call build code to populate code table with Huffman tree
Construct a header, which is given in header.h
Set header magic to MAGIC
Set header permissions to permission of infile
Set header tree size to number of bytes in huffman tree
Set header file size to infile size
Write the header to outfile
Call dump_tree
Read in BLOCK into buffer
Loop through the BLOCK read in
Write the codes from code table to outfile

Call flush code after writing all symbols

Print compression statistics

Close infile and outfile

Huffman Decoder:

While command line options given
- Case ‘h’: print help message
- Case ‘i’: takes input file to decode
- Case ‘o’: takes output file to write decompressed input to
- Case ‘v’: prints decompression statistics to stderr

Declare a Header struct
Read in the header from infile with read_bytes
If the MAGIC number does not equal MAGIC
    Return error

Set permission of outfile to match header permissions
Create an array that is tree_size long to read in dumped tree
Read in dumped tree from infile into array
Call rebuild_tree() to rebuild Huffman tree

Declare a variable to track symbols written
Declare a variable to hold bit value
Set a Node to the node returned by rebuild_tree(). This will be node used to traverse tree
Keep looping while decoded symbol does not match file size
    If code_get_bit is equal to 0
        Go to the left child of the current node
    Else,
        Go to the right child of the node

    If the node has not children
        Write the symbol of node to outfile
        Increment symbol by 1
        Reset current node back to root node

Print decompression statistics

Close infile and outfile