Assignment 5: Huffman Coding

By Derfel Terciano

Background

Everyday we always use some form of data compression. Whether through networking, or using zip files, data compression is used everyday. In this assignment, we will be using the Huffman Coding algorithm in order to compress data. As a note, remember that everything will be read in bytes and bits and not in a char type.

We will go over the Huffman algorithm later for now, we will first cover the abstract data types (ADTs) that will be used in order to make the algorithm easier to understand.

ADTs:

- Nodes
- Priority Queues
- Stacks
- Codes
- I/O
- Huffman Coding Module

Nodes

```
1 typedef struct Node Node;
2
3 struct Node {
4     Node *left;          // Pointer to left child.
5     Node *right;         // Pointer to right child.
6     uint8_t symbol;      // Node's symbol.
7     uint64_t frequency; // Frequency of symbol.
8 };
```

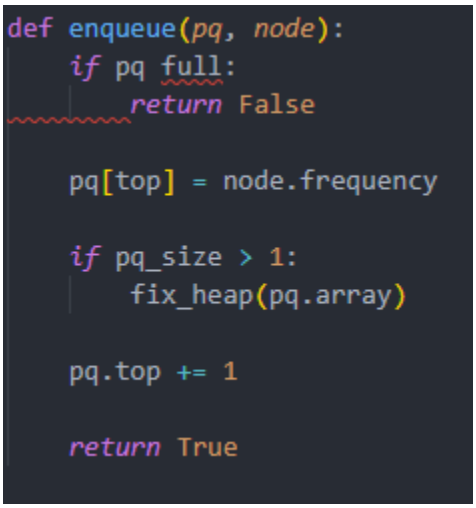
Above is the struct definition of the Node ADT.

- Node *node_create(uint8_t symbol, uint64_t frequency)
    - This is just the constructor for the Nodes.
    - No pseudo code will be needed here since we will just be calling malloc() and setting symbol to symbol and frequency to frequency.
- void node_delete(Node **n)
    - This is the destructor for the Node.
    - No pseudocode is needed here since we will just be calling free() and setting the Node pointer to NULL.
- Node *node_join(Node *left, Node *right)
    - This is a simple function that just combines the left and right nodes together. In addition, it sets the frequency as the sum of the left and right nodes
    - We don't need a thorough pseudocode for this since we will just be doing the following:
        - Symbol = $
        - Frequency = left + right
        - Return node pointer
- void node_print(Node *n)
    - This just prints whether or not the nodes have been created successfully and whether they have been joined or not

Priority Queues

Since most of the Huffman algorithm uses a Huffman tree, we would need to order certain characters based on their frequency. With that being said, we will use priority queues with the help of fix_heap() [from asgn3] in order to prioritize certain elements in the queue.

- Since the struct is not specified in the assignment doc, I will be using the following fields:
    - Top: the top of the queue
    - Tail: the front of the queue
    - *items: the array of the queue
- PriorityQueue *pq_create(uint32_t capacity)
    - This is the constructor function
    - We will be doing the following steps to initialize pq:
        - tTop = 0
        - Calloc *capacity* amount of 8 bit integer into the *items array
- void pq_delete(PriorityQueue **q)
    - This is the destructor function.
    - We will free the array, then the PriorityQueue pointer, then we will seth the pointer to NULL
- bool pq_empty(PriorityQueue *q)
    - This function checks if the queue is empty.
    - For this all we need to do is the following:
        - If top == 0 : return true
    - If the top is 0 then that means there is nothing currently in the array
- bool pq_full(PriorityQueue *q)
    - This checks if the queue is full. We can do this by doing the following short check:
        - if top == capacity: return true
    - This just says that if the tail is at the capacity, then we are at full capacity
- uint32_t pq_size(PriorityQueue *q)

    - This tells us the number of items in the queue.
    - This can be done by returning the following:
        - Return top
- bool enqueue(PriorityQueue *q, Node *n)
    - This function enques a node into the queue.
    - This function fails if the queue is already full.
    - We can implement this function using the following pseudocode I made:
        - 
```
def enqueue(pq, node):
    if pq full:
        return False

    pq[top] = node.frequency

    if pq_size > 1:
        fix_heap(pq.array)

    pq.top += 1

    return True
```
    - Since this is a priority queue, we must fix a min heap on the array so that the highest priority is always in index 0 of the array
- bool dequeue(PriorityQueue *q, Node **n)
    - This function dequeues the highest priority item in the queue
    - REMEMBER: queues always uses the FIFO rule so we must pop everything fro element 0.
    - Below is the pseudo code as to how to approach this

```
def dequeue(pq, POINTER_node):
    if pq empty:
        return False

    POINTER_node = pq[0]

    for i in range(0,cpapcity -1): #shift the elements down by 1 element
        pq[i] = pq[i+1]

    pq.top -= 1
    fix_heap(pq.array)

    return true
```

- We need to shift all the elements down due to the fact that we always want the smallest element or the root of min heap to always be the first element in the queue

- void pq_print(PriorityQueue *q)
  - This is a print function that prints out the priority queue
  - This is helpful for debugging
  - To implement this, all we need to do is have a loop that prints out the elements of the array.

Stacks

In order to decode an encoded message, we would need to use a stack ADT. Since, we have already dealt with stacks in asgn 4, we can just use the same pseudocode and concepts from asgn 4.

- Stack *stack_create(uint32_t capacity)
  - This is the constructor for the stack.
  - Refer to asgn4 doc to find full code for the constructor stack.
  - NOTE: instead of having an array of ints we will be using an array of Nodes
- void stack_delete(Stack **s)
  - This is destructor for the stack
  - Refer to asgn4 doc to find full code for the destructor stack.

- bool stack_empty(Stack *s)
    - This checks if the stack is empty
    - In order to check if the stack is empty, all we need to do is:
        - If (stack top == 0): return true
- bool stack_full(Stack *s)
    - This function checks if the stack is full.
    - This can be simply implemented by saying:
        - If (stack top == stack capacity): return true
- uint32_t stack_size(Stack *s)
    - This function returns the number of nodes in the stack.
    - This can also be simply implemented by saying:
        - Return the current stack top
- bool stack_push(Stack *s, Node *n)
    - This function pushes items onto the stack.
    - The following pseudocode shows us how to push to the stack:
        -
```
def stack_push(STACK, item):
    if stack_is_full:
        return False
    STACK[top] = item
    top += 1
    return True
```
- bool stack_pop(Stack *s, Node **n)
    - This function pop items from the stack and passes the popped item into a pointer.
    - The following pseudocode shows us how it's done:
        -
```
def stack_pop(STACK, pointerV):
    if stack_is_empty:
        return False
    #pass to pointerV the following: STACK[top]
    top -= 1
```
- void stack_print(Stack *s)

    - This just prints the stack
    - In order to implement this, we can just use a for loop that iterates throughout the array and print out each element

Code

- Code code_init(void)
    - This is a special constructor that does not use any dynamic memory allocation. Instead, we will just initialize the top to be 0 and zero out the array of bits.
- uint32_t code_size(Code *c)
    - This returns the size of the code which is the equivalent to the number of bits pushed.
    - We can determine this by returning the value of top.
- bool code_empty(Code *c)
    - This just returns true if the code is empty
    - We can implement this by checking whether or not the top is 0.
- bool code_full(Code *c)
    - This checks whether or not the code is full
    - This can be implemented by checking whether or not the top field is equal to the ALPHABET macros defined in defines.h
- bool code_set_bit(Code *c, uint32_t i)
    - We set the bit at index i to 1.
    - We can implement this by bitwise shifting the number 1 by i indices and then using bitwise OR on the code
    - Refer to Eugene's section video to find out how to do this
    - Also refer to the code comments repository
- bool code_clr_bit(Code *c, uint32_t i)
    - This clears the bit at a certain index
    - To implement this, we will probably need to take the NOT of a bit and use the bitwise AND in order to clear the bit at *i*.

    - Refer to the code comments repo for more help.
- bool code_get_bit(Code *c, uint32_t i)
    - This gets the bit at index *i*.
    - In order to implement this, we would just just need to shift the index up *i* times and use the bitwise & in order to get the bit at the index
- bool code_push_bit(Code *c, uint8_t bit)
    - Since we have an array of bits, we will just treat this array as a stack.
    - For the implementation of this, just refer to stack_push from above
- bool code_pop_bit(Code *c, uint8_t *bit)
    - This pops bits from the array.
    - Refer to stack_pop from above in order to implement this function
- void code_print(Code *c)
    - This just prints the array of bits
    - We can just use a for loop to print out the array

A Huffman Coding Module

Before we use the encoding and decoding modules we will create an interface of all the necessary steps to make implementing the Huffman algorithm easier.

- Node *build_tree(uint64_t hist[static ALPHABET])
    - This builds a Huffman tree based on the inputted histogram.
    - The pseudo code for this was given to us in the assignment doc:
    -
```
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
```
- void build_codes(Node *root, Code table[static ALPHABET])

- This function fills in the coding table. This would build the code for each symbol in the Huffman tree.
- The following pseudocode is provided to us from the assignment doc:

```
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)
```

- Here, we are using a recursive call of build_codes

- void dump_tree(int outfile, Node *root)
  - This function uses a post-order traversal algorithm of the Huffman tree startting at the root and writing it to the outfile. This function uses 'L' to signify leaf and 'I' to signify the interior nodes.
  - The following pseudocode is given to us in the assignment pdf:

```
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_dump[static nbytes])

    - This function creates a Huffman tree out of the tree dump that was created in the previous function.
    - The implementation would be the same as build_tree() however, we need to take into account the tree_dump instead of the hist[] array.
- void delete_tree(Node **root)
    - This is a destructor function for the trees. Since we are calling a bunch of functions that dynamically allocates memory, we would need to use a post order traversal of the tree and free up all the nodes used.
    - We would just need to re-implement dump_tree but instead we would be freeing all the nodes we created.

IO

For this assignment we are not able to use <stdio.h> since we need to deal with raw bytes and not characters. With that being said, we would need to create our own version of stdio.h that can handle raw bytes.

- int read_bytes(int infile, uint8_t *buf, int nbytes)
    - This function reads a file and reads up to *nbytes* of data and stores it in a buffer.
    - To implement this, all we need to do is run the read() function and check to make sure we got the right amount of bytes inputted
    - If not, we keep looping to make sure we reach the end of the file or we reach up to nbytes of data.
- int write_bytes(int outfile, uint8_t *buf, int nbytes)
    - This function is the same read_bytes except we are writing to an outfile
    - To implement this, we would just need to do the same exact thing as read_bytes() expect, we keep looping write until we reach EOF or until we reach *nbytes*.
- bool read_bit(int infile, uint8_t *bit)
    - This function reads the infile and reads the bytes from read_bytes and stores each block into a bit buffer.

    - In order to implement this, all we need to do is store each bit in each byte onto a buffer.
- void write_code(int outfile, Code *c)
    - This function does the same thing as read_bit except it will write bits to a buffer.
    - To implement this, we can just use a loop to put each bit in Code, and place it in the buffer
- void flush_codes(int outfile)
    - This function just flushes the leftover bits in the buffer out. After using write_code, we will be having some leftover code so we must zero out any extra bits in the **last byte**.
    - To implement this, we can just perform some bitwise operations to zero out the last bits.

The Encoder

The encoder uses the Huffman algorithm to create a Huffman tree and dump out the tree to a code which will represent the Huffman code that will be used to decode later on.

The details and implementation pseudo code for the encoder is found on the assignment pdf which should be enough for me to understand how to implement the program.

The Decoder

The decoder takes in the dumped Huffman tree and reconstructs the tree. The tree then gets traversed down and then they are inputted back onto the stack. Once the stack has been made, we can then emit a binary code. From the binary code, we can then traverse down the tree again and output the final clear text output.

The details and implementation of the decoder can be found on the assignment pdf