Assignment 6 Design Document

Nicholas Chu

February 2022

1 Purpose

The purpose of this assignment is to implement Huffman coding which is used encoding and compressing information. To do this the program will need to use nodes, priority queues,
which are implemented in the functions below.

2 Node

In order to implement Huffman encoding it is important to create an abstract data type for a node. Nodes will be used to track the frequency of all the characters of a given input and will be used in a priority queue to construct a Huffman tree to determine the number of bits used to represent each symbol when encoding. The Node struct will hold four values: pointers to the left and right node, the symbol inside the node, and the frequency of the symbol within the node.

```
typedef struct Node Node;

struct Node {
    Node *left; //Points to left node
    Node *right; //Points to right node
    uint8_t symbol; //Symbol of node
    uint64_t frequency //Frequency of symbol of node inside message
}
```

In order to create, delete, and manipulate these nodes throughout the program it is necessary to create a few functions.

2.1 Node *node_create(uint8_t symbol, uint64_t frequency)

This function will be used to create an instance of a node. First initialize a new node then set the symbol and frequency of the node to the uint_8t's with their respective names specified by the parameters.

```
Node n
n.symbol = symbol
n.frequency = frequency
```

2.2 void node_delete(Node **n)

This to do delete a node. To do this you first free the left and right nodes, then the node itself. Set the pointer to the node to NULL as well.

2.2.1 Node *node_join(Node *left, Node *right)

Joins two nodes with each other by creating a new node. This is used specifically when enqueuing and dequeuing nodes when building the Huffman tree. This new node will point to the left and right nodes specified in the parameters and will have the symbol "$". Its frequency will be the sum of the frequency values of the left and right nodes.

```
Node *n
n->left = left
n->right = right
n->symbol = "$"
n->frequency = left.frequency + right.frequency
return n
```

2.3 void node_print(Node *n)

A debug function that can be used to confirm if nodes are being created and generated properly. Should simply just print current node's symbol and frequency and if left and right exist, print their symbols and frequencies as well.

3 Queue

The nodes are queued based on the frequency of their symbols, where nodes with smallest frequencies are higher priority, meaning they are more to the left. The queue is meant to enqueue nodes given their frequencies then:

- Dequeue two nodes from the queue
- Join the two nodes into another new node using node_join
- Queue the joined node back into the queue

And repeat until there is only one node left which will be used as the root for the Huffman tree. You will first need a struct for the priority queue:

```
typedef struct PriorityQueue PriorityQueue;

struct PriorityQueue {
    uint32_t head;
    uint32_t tail;
    uint32_t capacity;
    Node **queue;
}
```

Before you define the functions for the prioirty queue that will be included into the header file to be used in encode and decode you first will need to write some helper functions to simulate a heap data structure so you can dequeue and enqueue nodes easier.

3.1 void swap(Node *x, Node *y)

This function simply just swap nodes and will be used when implementing the heap.

```
Node tmp = *x
*x = *y
*y = tmp
```

3.2 uint32_t min_child(PriorityQueue *q, uint32_t first, uint32_t last)

Finds and returns the index of the child node with the smallest frequency of the node at the index given by first.

```
left = 2 * first
right = left + 1
if right < last frequency and right's frequency is lager:
    return right
else:
return left
```

3.3 PriorityQueue *q, uint32_t first, uint32_t last)

Fixes the heap, used for organizing nodes so they follow the conventions of the heap data structure from the given index first.

```
found = false
parent = first
child = min_child(q, parent, last)
while parent < last // 2 and found is false:
    if node frequency at index parent is larger than the child's:
        swap the nodes at parent and child
        child = min_child(q, parent, last)
    else:
        found = true
```

3.4 void build_heap(PriorityQueue *q, uint32_t first, uint32_t last)

This function build a heap using the queue from a specified range of indexes given by first and last.

```
for (parent = last/2, parent > first-1, i-=1):
    fix_heap(q, parent, last)
```

3.5 PriorityQueue *pq_create(uint32_t capacity)

This function is a constructor that creates a new queue and set its values. Its head and tail should both be set to 1 and capacity set to the number specified by the parameter, they are set to 1 to simulate 1 base indexing for heap. It should also create and allocate the appropriate memory for an array of type uint32_t's with a size based off capacity. Once done it should return the newly created priority queue.

```
PriorityQueue p
p.head = 1
p.tail = 1
p.capacity = capacity
p.queue = calloc(capacity, sizeof(uint32_t))
return p
```

3.6 void pq_delete(PriorityQueue **q)

This function is to clean the priority queue. Should free the memory allocated for the queue (array) inside q and set the pointer for q to NULL.

3.7 bool pq_empty(PriorityQueue *q)

This function returns true when the queue is empty and true false if not. To do this you would check if the head of the queue is pointing to the index at 0.

```
if q.head equals q.tail:
    return true
else:
    return false
```

3.8 bool pq_full(PriorityQueue *q)

The function checks if the queue is full and will return true when it is and false when not. This is done by comparing the head to capacity.

```
if q.head == capacity:
    return true
else:
    return false
```

3.9 uint32_t pq_size(PriorityQueue *q)

This should return the number of current items within the queue. This is done simply by returning the value of the head of the queue.

3.10 bool enqueue(PriorityQueue *q, Node *n)

This function takes a node n and enqueues it into the priority queue q. To do this you need to add the node to the array and then build it into a heap. In the code below it first adds the node from the parameter to the queue's array then uses an insertion sort to sort the array from nodes with the greatest frequency to the lowest.

```
if q is full:
    return false
set top of queue to node n
build_heap(q, q.tail, q.head)
q.head += 1
return true
```

3.11 bool dequeue(PriorityQueue *q, Node **n)

This dequeues a node from the priority q and saves it to n and returns true when it is able to dequeue and node or false when when the queue is empty. You set the node to top of the queue, swap it with the tail, and finally rebuild the heap. This will takes the the node that the tail points to and save it to n**, then increment the tail.

```
if q is empty:
    return false
n = q.queue[q.head - 2]
swap head and tail nodes
build_heap(q, q.tail, q.head-1)
q.head -=1
return true
```

3.12 void pq_print(PriorityQueue *q)

This is a debug function that will print the specified priority queue. Should print the head, tail, and array of the priority queue.

4 Code

The Code is an abstract data type which represents a stack of bits which will later be used to represent a symbol of a node. You also need to define a few macros before hand:

```
#define BLOCK 4096
#define ALPHABET 256
#define MAGIC 0xBEEFBBAD
#define MAX_CODE_SIZE (ALPHABET / 8)
#define MAX_TREE_SIZE (3 * ALPHABET - 1)

typedef struct {
    uint32_t top;
    uint8_t bits[MAX_CODE_SIZE ];
} Code;
```

4.1 Code code_init(void)

A constructor function, which will create a new instance of Code and return it. It will set top to 0 and zero out the array of bits titled bits.

```
Code c
c.top = 0
c.bits[MAX_CODE_SIZE] = {0}
```

4.2 uint32_t code_size(Code *c)

All this just does is return the number of bits pushed onto code.

```
return c.top
```

4.3 bool code_empty(Code *c)

Returns true if Code is empty, else it returns false.

```
if c.top == 0:
    return true
else:
    return false
```

4.4 bool code_full(Code *c)

If the Code is full then it returns true, else it returns false.

```
if sizeof(code.bits) == MAX_CODE_SIZE:
    return true
else:
    return false
```

4.5 bool code_set_bit(Code *c, uint32_t i)

Sets the bit at index i in the bit array of Code c to 1. If i is in range and is successfully set then return true, otherwise return false. Since bits is an array of of bytes you will need to use a bit vector to set the bit at a specified index.

```
if i not in range of c.bits:
    return false
c->bits[i / 8] |= (0x1 << i % 8)
return true
```

4.6 bool code_clr_bit(Code *c, uint32_t i)

Sets the bit at index i in the bit array of Code c to 0. If i is in range and is successfully cleared then return true, otherwise return false.

```
if i not in range of c->bits:
    return false
c->bits[i / 8] &= ~(0x1 << i % 8);
return true
```

4.7 bool code_get_bit(Code *c, uint32_t i)

Gets the bit at specified index from Code. If i is out of the range or the bit at the index is 0 it will return false, else if the specified index has a bit of 1 then return true.

```
if i not in range of c->bits:
    return false
if c->bits[i / 8] & (0x1 << i % 8):
    return true
else:
    return false
```

4.7.1 bool code_push_bit(Code *c, uint8_t bit)

Pushes a bit to Code. To do this you setting the top of the bits array to the specified bit, then you increment top. It should return false when Code is already full and true when the bit is successfully pushed.

```
if c.bits full:
    return false
c.bits[i] = bit
return true
```

4.8 bool code_pop_bit(Code *c, uint8_t *bit)

Pops a bit off of code and saves it to *bit. To do this first save the bit at top and decrementing top by 1. If top is at 0 then return false, else if it decrements successfully then return true.

```
if c->bits empty:
    return false
bit = c->bits[i]
return true
```

4.9 void code_print(Code *c)

Is just a debug function and should print top and the array of bits of c.

5 I/O

The functions below are going to be used to read, write, open, and close input and output files using syscalls. These functions will also keep track of statistics like bytes_read and bytes_written.

5.1 int read_bytes(int infile, uint8_t *buf, int nbytes)

This function will loop through an input file reading bytes and saving them to a buffer until the buffer is full or it reaches the end of a file. Then it will return the number of bytes read.

```
while unread bytes still in infile and *buf is not full:
    read byte from infile
    bytes_read += number of bytes from read
return bytes_read
```

5.2 int write_bytes(int outfile, uint8_t *buf, int nbytes)

This function writes nbytes from the buffer or until there is no more bytes left to print from the buffer. To do this it will need to loop until either the end of the buffer is reached or nbytes is reached.

```
i = 0
write byte to file and clear buf
while written bytes < nbytes and still bytes in buffer:
    write byte and clear buffer
    nbytes += 1
return bytes_written
```

5.3 bool read_bit(int infile, uint8_t *bit)

The function is fore readings bits from a file. Since it is not possible to read a single bit from a file the work around is to read a byte from a file, save that byte to a buffer, and access bits from the buffer one by one. To do this the function will use static variables to keep track of which byte from the buffer it is reading and what bit from that byte will be read. It will save a bit to *bit through bit shifting.

```
static buf[BLOCK]
static byte_index
static bit_index
read bytes from infile and save to buffer:
if (read_buf[bit_index/8] & (1 << bit_index % 8)):
    bit = 1
    bit_index += 1
    return true
 else:
    bit = 0
    bit_index += 1
    return true
return false
```

5.4 void write_code(int outfile, Code *c)

This function writes bits to an outfile and should print out BLOCK bytes. It should first take the bits from c and save them to a buffer of BLOCK bytes in size. Then it should write them to print the contents of that buffer to the outfile.

```
for every bit in code_buf:
    if bit is 1:
        push 0 to code_buffer
    if bit is 0:
        push 1 to code_buffer
if buf is full:
    write buf to outfile
```

5.5 void flush_codes(int outfile)

Prints and left over bits from the buffer from write_code after the file has been completely read.

```
if buf is not empty:
    write remaining bytes in buf to file
    bit_index = 0
    clear buf
```

6 Stacks

In order for the decoder to reconstruct a Huffman tree it will need to use stacks. The stacks struct is defined as:

```
struct Stack {
    uint32_t top;
    uint32_t capacity;
    Node **items;
};
```

6.1 Stack *stack_create(uint32_t capacity)

Constructor that creates a stack of size capacity and returns it.

```
Stack s
s.capacity = capacity
s->items = calloc(capcity, sizeof(unit32_t))
return s
```

6.2 void stack_delete(Stack **s)

Destructor for a stack, should free all the allocated memory and sets the pointer to the stack to NULL.

```
free(s->items)
s* = NULL
```

6.3 bool stack_empty(Stack *s)

Returns true if stack if empty else it returns false.

```
if 0 elements in s->items:
    return true
else
    return false
```

6.4 bool stack_full(Stack *s)

Returns true if stack if full and false when anything else.

```
if elements in s->items is equal to capacity:
    return true
else:
    return false
```

6.5 uint32_t stack_size(Stack *s)

Returns number of nodes in the stack, or in other words the number of elements saved into the stack. One way to find this is by just getting the value for top.

```
return s.top-1
```

6.6 bool stack_push(Stack *s, Node *n)

Pushes a node onto the stack. This is done by setting the element at the top of the stack to the node then increment top. The function should return true when a the push is completed.

```
if s is full:
    return false
s->items[top] = n
s.top += 1
return true
```

6.7 bool stack_pop(Stack *s, Node **n)

Pops an element from the stack. This should take the top element of the stack and save it to n, while also decrementing top, and returning true or false on if it was able to successfully pop.

```
if s is empty:
    return false
n = s->items[top]
s.top -= 1
return true
```

6.8 void stack_print(Stack *s)

Is a debugging function that should print the contents of stack which should include top, capacity, and items.

7 Huffman Coding

This module contains the functions that will be implementing Huffman Coding. These functions will be used when encoding and decoding later.

7.1 Node *build_tree(uint64_t hist[static ALPHABET])

Takes a computed histogram and builds a Huffman tree from it. You first create nodes based on the information from the histogram which is added to a priority queue. Then using the priority queue, dequeue the two nodes at a time then join them. Repeat this until there is only one node left in the queue. That will be the root nodefor the Huffman tree and what will be returned.

```
for element in hist:
    if hist[element] != 0:
        create node with symbol element and frequency of hist[element]
        add the new node to queue
while there are at least 2 nodes in the queue:
    dequeue 2 nodes, one will be left and the other right
    node = node_join(n, left, right)
    add node back into queue
return root node
```

7.2 void build_codes(Node *root, Code table[static ALPHABET])

Using the Huffman tree, you build the codes that will represent each symbol and save those codes to a table. To do this you walk though the tree, in which you push a 0 to the code when going down a left node and 1 when going down a right node. You continue this until you have reached a leaf node, and it will need to be done recursively. This is the code that will be used to represent that symbol.

```
if root is leaf:
     save code c to leaf
 else:
     push 0 to c
     build_codes(left node, table)
     pop bit from c

     push 1 to c
     build_codes(right node, table)
     pop bit from c
```

7.3 void dump_tree(int outfile, Node *root)

Prints the tree to outfile in the in post order traversal to an outfile. This is done recursively by going down each node and writing "L" if the node is a leaf node and 'I' if it is an interior node. Below is what post order traversal through the tree would look like:

```
if node exists:
    dump_tree(outfile, node.left)
    dump_tree(outfile, node.left)

    if node has no leaf nodes:
        write "L" to outfile
        write node.symbol
    else:
        write "P" to outfile
```

7.4 Node *rebuild_tree(uint16_t nbytes, uint8_t tree_dump[static nbytes])

Constructs a Huffman tree from its post order dump created tree_dump and should it should be the length of nbytes. It should return the root node. It will read the dumped tree if it reads an L that means the next item will in the array is the symbol for a leaf node. You take that symbol and make it a node, frequency does not matter in this case, then enqueue it. If you read a P from the dump then deqeue two nodes, join the two nodes, and enqueue the joined node. When you have finished going though the tree dump then there should only be left a single node left in the queue which should be the root node.

```
for item in tree_dump:
    if item is L:
        //Take next item and use it for the symbol for node
        item += 1
        n = node_create(item, 0)
        enqueue n
    if item is P:
        n1 = pop from enqueue
        n2 = pop from enqueue
        n3 = node_join(n1,n2)
        enqueue n3
return root node in queue
```

7.5 void delete_tree(Node **root)

Goes through the nodes of the Huffman tree via post order traversal and uses node_delete() to delete each one.

```
if node exists:
    dump_tree(outfile, node.left)
    dump_tree(outfile, node.left)

    if node has no leaf nodes:
        delete_node(node)
```

8 Encoder

Used for taking bytes and compressing them into a smaller representation. The program will create a histogram from the read contents of the infile, creates a Huffman tree using said histogram, and prints the Huffman tree to an outfile. The steps for this would be:

- First use read_bytes, then find each individual symbol in the buffer and its frequency, make them into nodes and save them into a histogram.

  ```
  read_bytes(infile, buf, nbytes)
  hist[256]
  for each byte in buf:
      if byte is not a node in hist:
          make new byte with byte as symbol
      if byte already has a node:
          node.frequency += 1
  ```

  The histogram will need to have at least 2 elements for it to work properly, specifically when building a tree.

- Call build_tree(hist) using the created histogram and save the root node of that tree.
- Create an array of 256 (ALPHABET) Codes and filling it by using root node and the created array as parameters for build_codes().
- Write the code for each symbol using write_code() and flush any buffered codes using flush_codes.
- Create a header struct:

  ```
  typedef struct {
     uint32_t magic;
     uint16_t permissions;
     uint16_t tree_size;
     uint64_t file_size;
  } Header;
  ```

  For magic save the macro MAGIC, permissions should save the infile permissions which is got through fstat(), tree_size is the Huffman tree size, and file_size is the number of bytes of the infile which is basically just the number of bytes read. Once done initializing it print it to the outfile.

- Write the constructed Huffman tree to an outfile using dump_tree().
- Close files and free/delete/clear anything that was dynamically allocated memory.

9 Decoder

Used for undoing the encoder. It takes a compressed infile and decompresses it, placing the decompressed output to outfile. It will need to:

- Read the header and verify the magic number, and if it was unable to verify quit.
- Set the permissions field of the outfile to that specified in the header.
- Read the dumped tree and then use rebuild_tree() to reconstruct the tree and use tree_size from the header for its size.
- Read and translate the contents of the infile by:
  1. Reading a bit and going down, starting at the root, the reconstructed Huffman tree's left leaf if the bit read was 0 or right leaf if 1 was read and continue doing this until you hit a leaf node with a symbol in it. Write that symbol to the outfile.
  2. Start over at the root and repeat until all bytes have been read.
- Close any opened files and free/delete/clear anything that was dynamically allocated memory.