asgn5: Huffman Coding

Student name: *Serjo Barron*

Course: *CSE13S* – Professor: *Prof. Long*
Due date: *10/28/21*

1 Program

This program implements "Huffman Coding", which is often associated with compressing data so that it can be sent/downloaded in smaller sizes (of bytes), which is the encoding portion. Which then after, the compressed data can then be decoded back to its original contents, and original size. Compressing data is quite applicable to the real world and is seen when sending images, zipped files, or anything that would regularly require more kbytes or bytes of data. Though there are other forms of more efficient and complex data compression, this particular form of data compressing that will be applied in this assignment is Huffman Coding. Huffman Coding being one of the first forms of data compression invented by David Albert Huffman all the way back in 1952.

2 Required Files

1. encode.c
   - This file will be one of the two Test Harnesses of Huffman Coding. This file in particular will contain everything needed to encode a message.
2. decode.c
   - This file is the second of the two Test Harnesses of Huffman Coding. This file in particular will contain everything needed to decode a message.
3. defines.h
   - This header file will include defined macros used throughout the rest of the ADT's and with the encoder/decoder.
4. header.h
   - This header file will include a defined struct which will be the format of containing and writing the header of each encoded message.
5. node.h
   - This is the header file includes the struct definition and function prototypes of the node ADT.

6. node.c
    - This file contains the implementation of the node ADT.
7. pq.h
    - This is the header file includes the struct definition and function prototypes of the priority queue ADT.
8. pq.c
    - This file contains the implementation of the priority queue ADT.
9. code.h
    - This is the header file includes the struct definition and function prototypes of the code ADT.
10. code.c
    - This file contains the implementation of the code ADT.
11. io.h
    - This is the header file includes extern variables for statistic gathering and function prototypes of the I/O ADT.
12. io.c
    - This file contains the implementation of the I/O ADT.
13. stack.h
    - This is the header file includes the function prototypes of the stack ADT.
14. stack.c
    - This file contains the implementation of the stack ADT.
15. huffman.h
    - This header file includes the function prototypes of the huffman ADT.
16. huffman.c
    - This file contains the implementation of the huffman ADT.
17. Makefile
    - This file is used to link all the required files (ADT's) with the encoder and decoder, and will produce their corresponding executables.
18. README.md

- A file in markdown syntax which contains a program description and instructions on how to use and run the program (encoder/decoder).

19. DESIGN.pdf

    - The design process and structure of the program (this document).

3 Psuedocode and Function Descriptions

3.1 Nodes

This function will be our constructor for a node.

```
define node_create(symbol, frequency):
        malloc memory for node (n)
        set symbol as symbol
        set frequency as frequency
        set right child to NULL
        set left child to NULL
        return node
```

This function will be our destructor of a node.

```
define node_delete(n):
        free node
        set node pointer to NULL
        return
```

This function joins a right and light child node, which will result in a parent node. The parent node will set its left child as **left** and set its right child as **right**. The parent node will be differentiated with the '$' as its symbol and its frequency will be a sum of its left and right child's frequency. Meaning that if the left child is a parent node, then '$' is a summation of all its successive nodes frequencies.

```
define node_join(left, right):
        add left and right freq (frequency)
        set summed freq as parent freq
        set parent symbol to '$'
        set left child as left
        set right child as right
        return (pointer to) parent node
```

A debug function that will be used to verify/check that the nodes are created and joined properly.

```
define node_print(n):
        print node symbol, its frequency
        as well as its left child symbol & frequency
        and its right child symbol & frequency.
```

3.2 Priority Queue

3.2.1 Min Heap

Before going on to the actual priority queue, here are a few helper functions (similar to Heapsort in asgn3, except we don't sort) to ensure the nodes can be ordered properly by using a Min Heap, meaning that element (node) with the lowest value, in this case that will be frequencies, will be the parent node. Then all successive nodes after will have a higher frequency than its parent node.

To do this, first we must determine the smallest node to place at the top, then work downwards from there. To do this, we will make use of a min_ child function which will compute a left node defined as $L = 2 \times first$, and a right node defined as $R = L + 1$, which we will then compare to return the smallest of the two, which will serve as a pivot value. The next helper function is fix_ heap, which will rebuild the heap so that the smallest element is the root. Lastly, the function build_ heap will iteratively call fix_ heap to construct the min heap.

```
define min_child(A, first ,last):
        left=2*first
        right=left+1
        if (right<=last and A[right-1] < A[left=1]):
                return right
        return left

define fix_heap(A, first ,last):
        found=false
        mother=first
        least=min_child(A,mother, last)
        while (mother<=last/2 and not found):
                if (A[mother-1] > A[least-1]):
                        swap(A[mother-1],A[least-1])
                        mother=least
                        least=min_child(A,mother, last)
                else:
                        found=true
        return
```

The struct elements for the priority queue.

```
Priority Queue struct:
        head (front of the queue)
        tail (back of the queue)
        capacity (size of queue)
        nodes (array of nodes)
```

3.2.2 Rest of Priority Queue

This function will be our constructor of the priority queue. Like a regular queue, this will work in a LIFO (last in first out) basis, but will each node in the queue will have an associated priority (frequency) attached to it. Nodes with a lower frequency will be placed higher in the queue, while nodes with higher frequencies will be placed lower in the queue. This queue will act similarly to that of a min heap (where each parent node is a lower or equal to its children nodes).

```
define pq_create(capacity):
        allocate memory for pq struct (q is priority queue)
        if (q):
                head=0
                tail=0
                capacity=capacity
                allocate (capacity) memory for nodes
                if allocation fails:
                        free q
                        set q ptr to NULL
        return q
```

This function will be the destructor of the priority queue.

```
define pq_delete(q):
        if (q and nodes):
                free nodes
                free q
                set q ptr to NULL
        return
```

This function checks whether the priority queue is empty, and returns true if so, and false otherwise.

```
define pq_empty(q):
        if tail==0:
                return true
        return false
```

This function checks whether the priority queue is full, and returns true if so, and false otherwise.

```
define pq_full(q):
        if tail==capacity:
                return true
        return false
```

This function returns the current size of the priority queue.

```
define pq_size(q):
        return tail
```

This function will enqueue a node into the priority queue, and if it is full prior to enqueueing the node return false, and true otherwise. (enqueue means to add an item in a queue)

```
define enqueue(q,n):
        if q is full:
                return false
        nodes[tail]=n (node)
        tail+=1
        k=tail
        while(k>1):
                parent=k/2(floor)
                fix_heap(nodes,parent,tail)
                k=parent
        return true
```

This function will dequeue a node from the priority queue, and "return" that dequeued item by passing it to a double pointer n. The node dequeued should be the one with the highest priority, meaning it is at the top of a min heap. Return false if the queue is empty prior to dequeueing a node, and true otherwise.

```
define dequeue(q,n):
        if q is empty:
                return false
        *n=nodes[head] (dereference n)
        tail-=1
        swap node[0] and node[tail]
        k=tail
        while(k>1):
                parent=k/2(floor)
                fix_heap(nodes,parent,tail)
                k=parent
        return true
```

A debug function for the priority queue.

```
define pq_print(q):
        for i in range(0,tail):
                print nodes[i] symbol and its frequency
        return
```

3.3 Codes

This function serves as a constructor, but unlike other constructors this does not have a corresponding destructor. As such, to initiate the code, all we need to do is create an array of length MAX_ CODE_ SIZE (a macro defined in defines.h).

```
define code_init():
        Code code
        for i in range(0,MAX_CODE_SIZE):
                bits[i]=0
        set top as 0
        return code
```

This function will return the size of the current code.

```
define code_size(c):
        return top
```

This function will check whether the code is empty, and return boolean value.

```
define code_empty(c):
        if top==0:
                return true
        return false
```

This function will check whether the code is full, and return a boolean value.

```
define code_full(c):
        if top==MAX_CODE_SIZE*8:
                return true
        return false
```

This function will set the bit at index i of the code to 1.

```
define code_set_bit(c,i):
        if (i<0 or i>=MAX_CODE_SIZE*8): (checks if i is out of range)
                return false
        bits[i/8] bitwise-or 0x1 left-shift by i%8 bits
        return true
```

This function will clear the bit at index i of the code, meaning changing that bit to 0.

```
define code_clr_bit(c,i):
        if (i<0 or i>=MAX_CODE_SIZE*8):
                return false
        bits[i/8] bitwise-and not(0x1 left-shift by i%8 bits)
        return true
```

This function will get (return) the bit (0 or 1) at index i of the code.

```
define code_get_bit(c, i):
        if (((bits[i/8] right-shift by i%8 bits) bitwise-and 0x1)==1):
                (means the bit at index i of the byte is a 1)
                return true
        (otherwise the bit at index i is not 1)
        return false
```

This function will push a bit (0 or 1) to the current top of the code, and then move the top up by 1.

```
define code_push_bit(c, bit):
        if code_full(c):
                return false
        if (bit==1):
                code_set_bit(c,top)
        else: (bit has to be 0, assuming that is only 0 or 1)
                code_clr_bit(c,top)
        top+=1
        return true
```

This function will pop a bit (0 or 1) from the current top of the code, and then move the top down by 1.

```
define code_pop_bit(c, bit):
        if code_empty(c):
                return false
        top-=1
        if (code_get_bit(c,top)==true):
                *bit=1
        else:
                *bit=0
        return true
```

A function for debugging purposes.

```
define code_print(c):
        iterate over bit indices (0 to code_size(c)):
        if code_get_bit is true:
                print a 1
        else:
                print a 0
        return
```

3.4 I/O

This will be the buffer "buf" referenced throughout the functions included in I/O, where BLOCK is 4KB or 4096 bytes, which is defined in defines.h.

```
uint8_t buf[BLOCK] (our buffer)
```

This function will mimick fread, a function included inside the stdio library. But rather than using that, the function read_ bytes will use syscalls to read a file in chunks of 4KB (4096 bytes), and keep track of the amount of bytes read. The function will read the infile until it has read the specified amount of nbytes, and then return the amount of bytes read.

```
define read_bytes(infile,buf,nbytes):
        bytes_read=0
        do:
                bytes=read(infile,buf+bytes_read,nbytes-bytes_read)
                bytes_read+=bytes
        while(bytes>0)
        return bytes_read
```

This function will be the opposite of read_ bytes, and by that write_ bytes will take the contents of the buffer (buf), and write them to an outfile in chunks of 4KB (4096 bytes), and keep track of the amount of bytes being written to the outfile. The function will then return the amount of bytes written to outfile.

```
define write_bytes(outfile,buf,nbytes):
        bytes_write=0
        do:
                bytes=write(outfile,buf+bytes_write,nbytes-bytes_write)
                bytes_write+=bytes
        while(bytes>0)
        return bytes_write
```

This function will perform a read_ bit, meaning that it will read a bit on the current byte being read in a buffer of BLOCK size (4096 bytes), and will continue to read each bytes bits until it has run out of bits to read.

```
(static buffer to read bytes)
static buf[BLOCK]
(use a static index to keep track of which bit is currently being read
in the byte)
static bit_index=0

define read_bit(infile,bit):
        if bit_index==0:
                (then read a new set of bytes into the static buffer)
                read=read_bytes(infile,buf,BLOCK)
                if read<BLOCK:
```

```
                end=read * 8 + 1 (set the end bit)
                (we add 1 since, we have to account for
                the fact that it is being read similar to an array)

        (dereference bit to store the bit retrieved at index
        specified by bit_index)
        *bit=(buf[bit_index/8] right shift by bit_index%8 bits) & 0x1
        bit_index+=1 (move onto next bit)
        if bit_index/8==BLOCK:
                reset bit_index to 0
        return bit_index/8!=end (bool evaluation)
```

This function will take the contents of a code and load its bits into a static buffer. When that buffer is filled with bytes, the contents of that buffer will be written to an outfile.

```
(static buffer to hold bits inside code)
static code_buf[BLOCK]

(static index to keep track of which bit is being written)
static index=0 (code index)

define write_code(outfile,code):
        for i in range(i,code_size(c)):
                if (code_get_bit(c,i)==true: (set bit to 1)
                        code_buf[index/8] then apply a
                        bitwise-or 0x1 left-shift by index%8 bits
                else: (set bit to 0)
                        code_buf[index/8]  then apply a
                        bitwise-and not(0x1 left-shift by index%8 bits)
                index+=1
        if (index/8==BLOCK):
                write_bytes(outfile,code_buf,BLOCK)
                index=0 (reset index to 0)
        return
```

This function will write out the codes to outfile if write_ code() did not end up filling a BUFFER (4096 bytes) with bits. flush_ codes() also accounts for extra bits that do not make up a full byte, the bits after the last bit are then zeroed out until a full byte is met.

```
define flush_codes(outfile):
        if index>0: (shared index b/t write_code and flush_codes)
                while index%8!=0:
                        code_buf[index/8] then a apply bitwise-and
                        with a not(0x1 left-shift by index%8 bits)
                        index+=1
                write_bytes(outfile,code_buf,index/8)
        return
```

3.5 Stacks

This function is a constructor for a stack. It allocates memory for the stack struct and initializes a stack (items) which will hold the nodes.

```
define stack_create(capacity):
        allocate memory for stack struct (malloc)
        if allocated properly:
                set top to 0
                set capacity to capacity
                allocate memory for items (calloc)
                if allocation to items failed:
                        free(s)
                        set s ptr to NULL
        return s
```

This function is a destructor for a stack. It will free the allocates memory for the items stack, and free the pointer to the stack.

```
define stack_delete(s):
        if s and items exist:
                free(items)
                free(s)
                set s ptr to NULL
```

This function checks whether the stack is empty.

```
define stack_empty(s):
        if top==0:
                return true
        return false
```

This function checks whether the stack is full.

```
define stack_full(s):
        if top==capacity:
                return true
        return false
```

This function returns the current size of the stack of nodes.

```
define stack_size(s):
        return top
```

This function pushes a node on to the items stack, and then will increment the top by 1.

```
define stack_push(s,n):
        if stack_full(s):
                return false
        items[top]=n (node)
        top+=1
        return true
```

This function decrements the top by 1, and then retrieves (pops) the node at the top of the items stack.

```
define stack_pop(s,n)
        if stack_empty(s):
                return false
        top-=1
        *n=items[top] (dereference n to store node)
        return true
```

This function will serve for debugging purposes (checking whether nodes get pushed/popped off the stack properly).

```
define stack_print(s)
```

3.6 Huffman Coding

This function will create a priority queue, and take the given histogram to populate that priority queue. A node will be created for each element in the priority queue whose frequency is greater than 0. After the priority queue has been made and populated, we will not dequeue 2 nodes, join them, and enqueue back the combined node until there is only 1 node left in the queue. The remaining node is the root of the huffman tree, which is returned.

```
define build_tree(hist[ALPHABET]):
        create pq of ALPHABET capacity

        (populate the pq with nodes)
        for i in range(0,255): (ALPHABET indices)
                if hist[i]>0: (freq. > than 0)
                        (symbol is i, in this case 8 bit unsigned
                        in place of a char)
                        node_create(symbol, hist[i])
                        enqueue(q, hist[i])

        (take pq and construct huffman tree)
        while size of pq>1:
                left=dequeue(q)
                right=dequeue(q)
                parent=node_join(left,right)
                enqueue(q,parent)
        root=dequeue(q)
        return root node
```

This function will create corresponding binary codes for each symbol of the Huffman tree. This will be done by traversing the Huffman tree, and if we encounter a node that has no left or right child, then we have reached a leaf node. If a leaf node is encountered the symbol of that node will be given a corresponding code. The codes are determined based on which direction you go down the tree. If you go right, you push a 1 to the code, and if you go left, you push a 0 onto the code.

```
define build_codes(root,code table):
        (have a code initialized once)
        if on leaf node:
                table[symbol]=c (code)
        else:
                code_push_bit(c,0)
                build_codes(left ,table)
                code_pop_bit(c,bit)

                code_push_bit(c,1)
                build_codes(right ,table)
                code_pop_bit(right ,table)
return
```

This function will also traverse the Huffman tree, but will use a post-order traversal to traverse the tree. If a leaf node is encountered (meaning that the node has no children), then we add an L to representing a leaf node, and the node corresponding symbol to the buffer. Otherwise, we add an I to the buffer. Once the index is equal to the tree_ size (calculated with 3 × number of unique symbols - 1), then we write out the contents of the buffer to outfile.

```
(create a buffer to contain the tree dump)
dump_buf[BLOCK]
(use a static index to index the bytes into the buffer)
index

define dump_tree(outfile ,root):
        if root: (node exists)
                dump_tree(outfile ,left)
                dump_tree(outfile ,right)
                if on leaf node:
                        dump_buf[index]=L (leaf node)
                        index+=1
                        dump_buf[index]=symbol
                        index+=1
                else:
                        dump_buf[index]=I (interior node)
                        index+=1
        if the index == tree_size:
                (write out the tree dump)
                write_bytes(outfile ,dump_buf,index)
        return
```

This function given the tree_ dump in a post-order fashion, will make use of a stack to reconstruct the Huffman tree. Given a tree_ dump and the size of that dump, we will iterate over nbytes to read the contents of the tree_ dump. If an 'L' is encountered we skip to the next index which must be a symbol. Otherwise, we encountered an 'I' which is an interior node. Whenever an 'L' is encountered we push a created node for the symbol after it onto the stack. However, in the else case, we pop the stack twice (right first, left second), and then join the 2 popped nodes into a parent and push that onto the stack. Once finished iterating over the dump, there should be one node left inside of the stack, which we pop and return as the root of the reconstructed Huffman tree.

```
define rebuild_tree(nbytes,tree_dump):
        stack_create()
        iterate over tree_dump
        if you encounter an 'L' the character after it is a 'symbol':
                push(symbol)
        else you encounter an 'I', an interior:
                (pop the stack twice)
                stack_pop(right)
                stack_pop(left)
                parent=join(left ,right)
                push(parent)
        return the root of the reconstructed tree
```

This function conducts a post-order traversal of the Huffman tree just like dump_ tree, but instead this function is used to deallocate the memory of all the nodes of the tree.

```
define delete_tree(root):
        if root exists:
                delete_tree(left)
                delete_tree(right)
                if node has no children:
                        node_delete(root)
```

4 Encode

The encoder will be one of the two executables needed to perform a complete encoding and decoding of a message using Huffman Coding. This file in particular (encode.c) will take an infile which includes the desired message to be encoded, and will return the encoded message to outfile. The encoded message will include 3 major things. The first being the header which includes information needed to decode the message in the decoder, the second being the tree dump which will be used later on to reconstruct the Huffman Tree, and the third being an emitted binary which will be used to decode symbol by symbol.

It should be noted however that encoding does indeed prefer larger messages/files and a fair amount of unique symbols to be the most effective. Encoding smaller files such as one including the word "banana", would indeed work. However, when thinking about the size of the encoded message (in bytes) the practicality of encoding such a message seems more pointless. It is important to remember that the point of encoding is to compress our data to a smaller size, not increase it.

```
(a function used to write header to outfile)
define dump_header(outfile ,header):
        write the conents of header to outfile

begin getopt loop:
        switch(opt)
        case 'h':
                set bool to print synopsis msg to true
        case 'i':
                set bool for stdin to false
                open infile and take perm with fstat()
        case 'o':
                set bool for stdout to false
                open outfile and set perm to infile
                with fchmod()
        case 'v':
                set bool for verbose stats to true
        default:
                set bool to print synopsis msg to true

if specified:
        print synopsis message
        end program

if specified:
        set infile to read from stdin
if specified:
        set outfile to write to stdout

buf[BLOCK]

create histogram array of ALPHABET size

(increment index 0 & 255 to account for
an empty file edge case)
hist[0]+=1
hist[255]+=1

populate histogram:
while bytes>0:
        bytes=read_bytes(infile,buf,BLOCK)
        for i in range(0,bytes):
         hist[buf[i]]+=1

build_tree with histogram

create code table of ALPHABET size

build_codes with root returned by build_tree
and the code table

iterate over histogram to count number of unique symbols

construct header:
        magic=MAGIC
        permissions=perm of infile
        tree_size=(3*symbols)-1
        file_size=size of infile (in bytes)

dump_header(outfile,header)

use lseek to read from start on file

(write the codes by iterating over input from infile)
while x>0:
        x=read_bytes(infile,buf,BLOCK)
        for i in range(0,x):
                write_code(outfile,table[buf[i]])

flush_codes(outfile)

delete_tree(root)
close(infile)
close(outfile)
end program
```

5 Decode

The decoder is the second of the two executables need to properly implement Huffman Coding. The decoder will take an infile containing the header, tree dump, and emitted binary (the outfile produced by the encoder) and use such information to first read the header and gain appropriate information before proceeding with the decoder and to verify that the file to be decoded is valid. To which we then reconstruct the Huffman tree by using the tree dump. Lastly, once the tree has been reconstructed we can then traverse the tree by iterating over the bits of the emitted binary to decode the message one symbol at a time.

If the decoding process works successfully, the decoded message will be of the same file size (in bytes) as the original file which was encoded. And as mentioned in the encoder, the efficiency of data compression (the amount of saved space) sees greater results when encoding and decoding files of larger sizes and large amounts of unique symbols.

```
define load_header(infile ,header):
        this function will take the header of the infile
        and read its contents into the header struct

define write_bin(infile ,file_size ,binary_buf,root):
        current=root
        while decoded_symbols!=file_size:
                read_bit()
                if bit==0:
                        (traverse to left child)
                        current->left
                else:
                        (traverse to right child)
                        current->right
                if on a leaf node:
                        (buffer the symbol)
                        binary_buf[decoded_symbols]=current->symbol
                        current=root (reset to root node)
                        decoded_symbols+=1
        return

begin getopt loop:
        switch(opt)
        case 'h':
                set bool to print synopsis msg to true
        case 'i':
                set bool for stdin to false
                open infile
        case 'o':
                set bool for stdout to false
                open outfile
        case 'v':
                set bool for verbose stats to true
        default:
                set bool to print synopsis msg to true

if specified:
        print synopsis message
        end program

if specified:
        set infile to read from stdin
```

```
if specified:
        set outfile to write to stdout

(read the header of the infile)
load_header(infile)

if the magic number is not MAGIC:
        print error msg
        exit program

(set outfile perms based on what was
read from the infile using fchmod())

dumped_tree[MAX_TREE_SIZE]

(read the tree dump into buffer)
read_bytes(infile ,dumped_tree, tree_size)

root=rebuild_tree(tree_size ,dumped_tree)

(buffer to hold the decoded message)
binary_buf[file_size]

(iterate over emitted binary to decode the msg)
write_bin(infile , file_size ,binary_buf, root)
write_bytes(outfile ,binary_buf, file_size)

delete_tree(root)
close(infile)
close(outfile)
end program
```

6 Error Handling

If the magic number read from the infile of the decoder is not the magic number, print an error message and end the program. This is also true if the header of the read infile is not complete, meaning it is missing information (magic, permissions, tree_ size, file_ size). The magic number in particular is a method of assigning which files can be decoded, and which files cannot.

Another thing to mention is regarding the creationg of the histogram in the encoder. As written in the pseudocode/structure of the encoder the 0th and 255th indexes are incremented by 1. The reason behind this is account of the possibility of encoding empty files. Though the produced Huffman tree will be slightly unoptimal, doing so will ensure that there are always 2 symbols to be encoded regardless if the file is empty.

7 Credit

1. Particular function designs pertaining to "io.c" have been inspired by Eugene Chou's implementation of them during his 10/28/21 and 11/2/21 in-person sections. As well as the initial draft for the design process behind Encode and Decode have been helped by the 10/26/21 Zoom section.

2. Function designs (mainly pertaining to Huffman Coding section) have been based off of/inspired by the provided pseudocode in the corresponding assignment document/specifications by Prof. Long

3. The write_ bin function in dequeue was designed with the help of the tutor Jason Yang, whom gave advice on how to approach it without using recursion. Since beforehand, I had implemented a recursive version (which worked) but was somewhat inefficient due to the vast amount of recursive calls needed to be done, which in the end complicated things.

4. I attended Miles Glapa-Grossklag's tutoring session on 11/5/21 where he pointed a minor but important aspect of read_ bit which I overlooked. Specifically it was in regards with the end bit, which beforehand the way I wrote it interpreted as the end byte.