Afzal Khan
afakhan@ucsc.edu
05/11/2021

CSE 13s Spring 2021
Assignment 6: Huffman Coding
Design Document

Overview:

The objective of this is to implement the Huffman coding algorithm in order to compress a given file. Using Hamming codes, the programs must be able to encode an inputted file and decode it. Three new ADTs will be used in this lab. The first is the Node ADT. This structure is responsible for keeping track of the frequencies of each symbol in the file. These nodes will make up the Huffman Tree. The second ADT is the Priority Queue. The priority queue will be used to hold all the nodes when calculating the leaves, parents, and root nodes of the tree. The Code data structure is the last ADT. Code is responsible for creating the unique code for each symbol. This is done through the traversal of the Huffman Tree in which each branch holds the value of either 1 or 0, respective to the right or left branch. The path taken to reach each symbol is the code for that symbol. Both encode and decode must support a set of command line arguments for both decode and encode.

Encoder:

- -h: Prints out a help message
- -i [infile]: Specify the input file path containing data to encode using Huffman Codes
- -o [outfile]: Specify the output file path to write the encoded data
- -v: prints out the statistics of the process to stderr

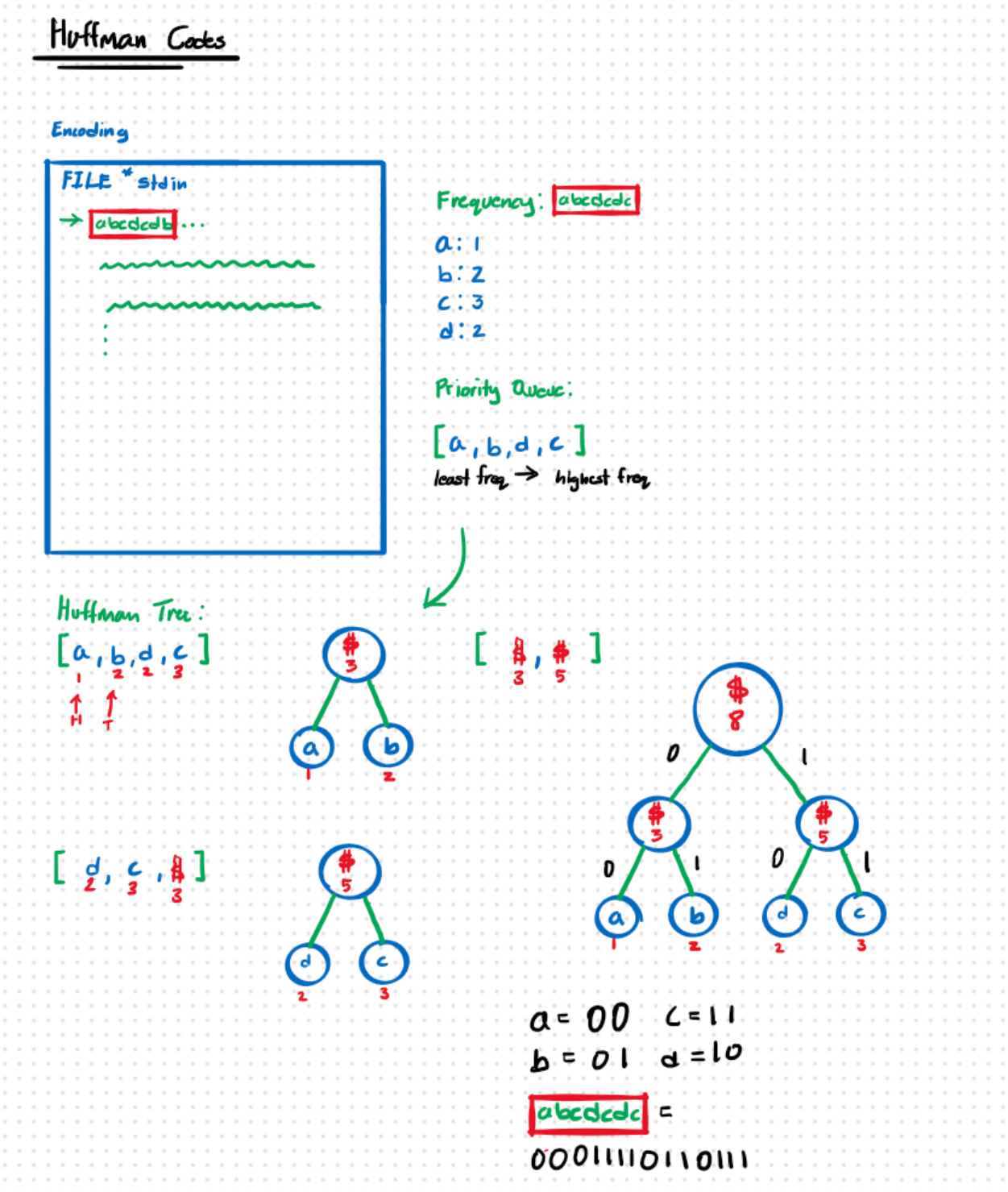
Decoder:

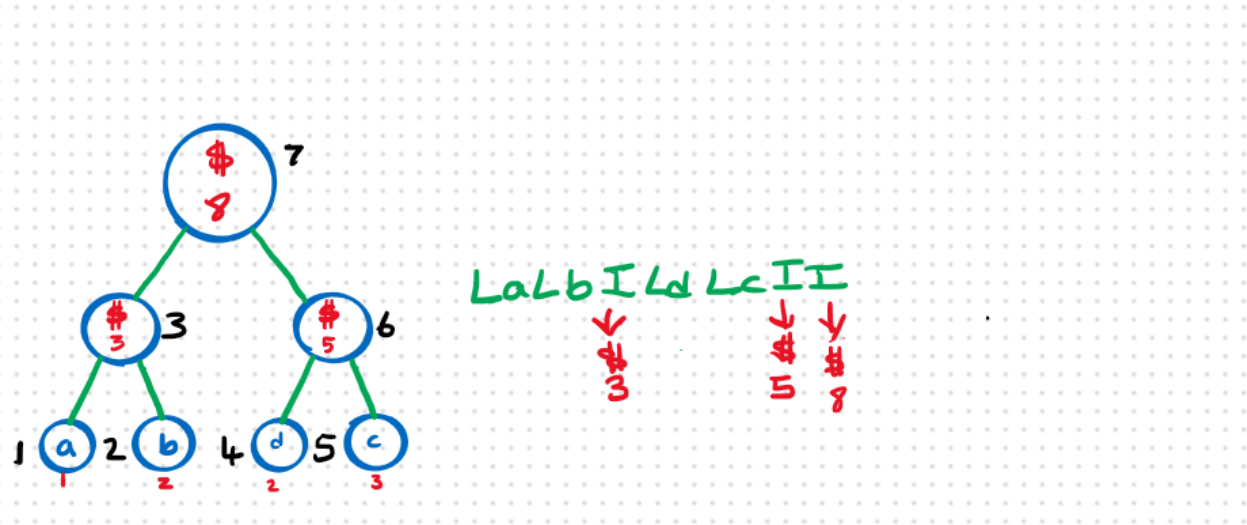
- -h: Prints out a help message
- -i [infile]: Specify the input file path containing Hamming codes to decode
- -o [outfile]: Specify the output file path to write the decoded Hamming codes to
- -v: prints statistics of the decoding process to stderr

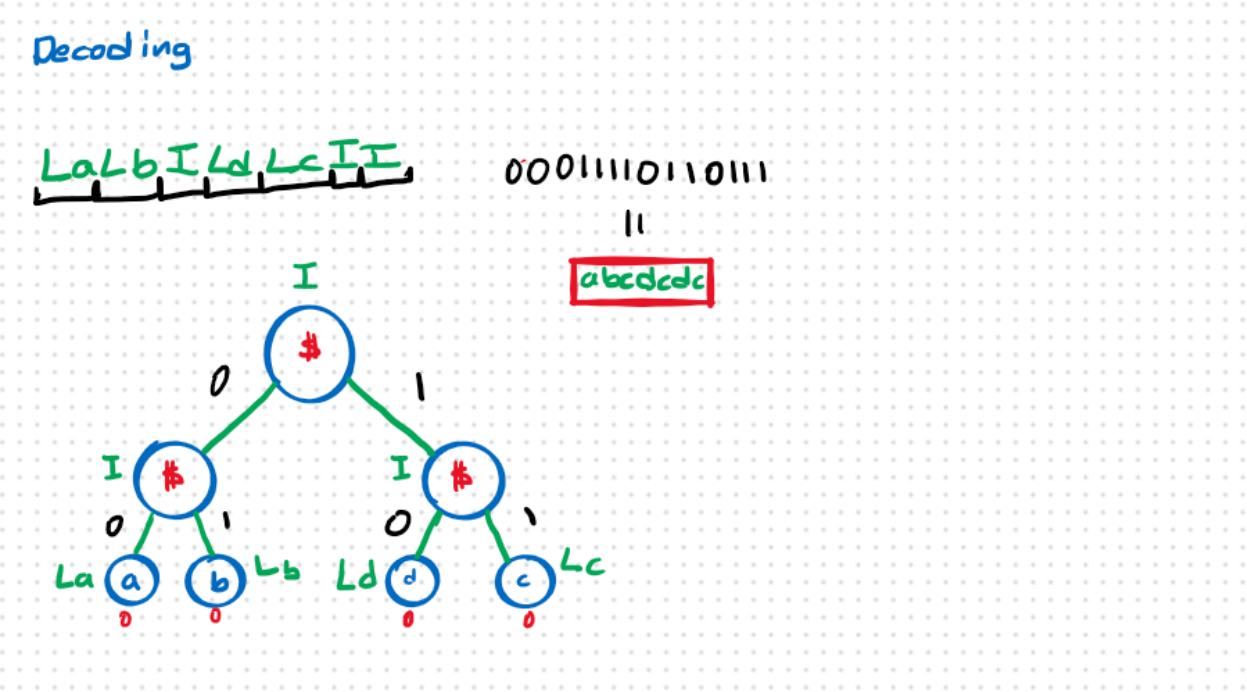
(Sourced from lab pdf)

Approach:

The following covers a generalized approach for what the Huffing code process entails. All drawing and understanding are drawn at thought through by me. The approach covers both encode and the tree dump as well as decoding.







It is important to note that when decoding, the tree dump only passes on the information of whether a node is a leaf or an internal (parent) node. This means that the frequency of the node is irrelevant. In this process, a stack will be utilized to push the information of the tree dump. Every time an “I” occurs, two values are popped, corresponding to the right and left children.

Pseudocode/ Explanation:

Code

```
code initialize:
    Code c;
    top = 0
    bits[MAX_CODE_SIZE] = {0}

code_size():
    return top of code

code_empty():
    return if top of code == 0

code_push_bit(bit):
    if code is full:
        return false
    if bit == 1 :
        set bit at top to 1

    top = top + 1
    return true for success

code_pop_bit(bit):
    if code is empty:
        return false

    decrement the top pointer
    bit = bit at top
    bit at top = 0;
    return true for success

code_print():
    for i in 256:
        print bit at index i in code
```

This pseudocode is for the Code ADT. Unlike traditional ADTs we have implemented in the past, the Code ADT is unique because it does not require the allocation of memory. Rather it uses the heap already available to the computer. In addition, the structure is completely transparent. This means that it does not contain private representations specific to that structure. The structure makes use of the pointer to top as a regular stack would. However, when initializing the top pointer, it is defined using string literals. The Code's push and pop logic is almost the exact same as that of a Stack ADT. However, since Code deals with individual bits, pushing and popping bits requires the manipulations of the stack using bitwise operators.

ANDing and ORing bits is the way to push, clear and pop the bits from the top index. The implementations id very reminiscent on that of the BitVector data structure.

Node

```
node_create():
    allocate memory for node structure
    node symbol = parameter passed symbol
    node frequency = parameter passed frequency
    node left = NULL
    node right = NULL

node_delete():
    delete memory for node
    set pointer to NULL

node_join(node left, node right):
    create a new node with parent node symbol
    frequency of new node = frequency of left + frequency of right
    left child of new node = node left
    right child of new node = node right
    return new node

node_print():
    print symbol + frequency
```

This pseudocode outlines the Node ADT. This structure is very short simple. It is used to keep track of the unique elements of an encoded file. The nodes will later be used to implement a tree based of the respective frequencies. Each node is created with a left and right child. Its default value is *NULL*. When the node_join function is used, there are two nodes passed in as parameters. These are considered the left and right nodes. In order to join the node properly, a new node is created with the symbol *$* representing a parent node. The parent node has a frequency equivalent to the sum of the left and right nodes. Finally, the new node is assigned to two children, the left node that was passed in, as well as the right node that was passed in.

Priority Queue

```
declare the queue structure:
    head pointer
    tail pointer
    size of stack
    max capacity
    array of nodes

creating_queue():
    allocate memory for the pq
    head pointer = start of queue
    tail pointer = start of queue
    size of queue = initial is 0
    capacity = size of node

delete_queue():
    if queue exists and has items:
        deallocate array of node
        deallocate queue

priorityqueue _empty():
    return if pointer to tail = pointer to head

priorityqueue _size():
    return current size

priorityqueue_full():
    return if index in front of pointer to tail = pointer to head

enqueue(x):
    if queue is full:
        return false - queue has no space
    else:
        size += 1
        index of tail pointer = node
        insertion sort after addition of node
        increment tail pointer
        return true if successful
```

```
dequeue(x):
    if queue is empty:
        return false - queue has nothing to pop
    else:
        size -= 1
        node  = node pointed by head pointer
        increment head pointer
        return true if successful
```

The Priority Queue data structure is a vital part of the Huffman encoding algorithm. In order for the nodes read to be of use, they need to be formed into a tree. This can only be done by using a queue that can organize the nodes based on their frequency. As a result, a Priority Queue is needed. Unlike the tradition enqueue and dequeue functions of a queue, this enqueue function uses an insertion sort to organize the nodes based on their frequency.

```
def insertion(arr, node):
    arr.append(node)
    for i in range(1,len(arr)):
        index = i
        value = arr[i]
        while(index > 0 and arr[index - 1] > value):
            arr[index] = arr[index - 1]
            index = index - 1

        arr[index] = value
```

This is a python implementation of the insertion sort used in the C code. Essentially, the program loops through all the nodes in the queue. If there is a node in which its frequency is less than the node before it, then a swap will occur between the nodes. The swap will continue until the nodes is no longer in a position where its frequency is larger than its predecessor. Once all the nodes are set and no more swaps can occur, the queue is said to be sorted. This addition of an insertion sort converts a regular queue into a priority queue.

Stack

```
declare the stack structure:
    top pointer
    capacity of stack
    array for items

creating_stack:
    top pointer = start index 0
    stack capacity = capacity defined by user
    allocate memory for items of x capacity
    if no items:
        deallocate stack

stack _empty:
    return if top pointer = start index

stack _size:
    return index of top pointer

stack_full:
    return if the stack size = capacity

stack_push(x):
    if stack is full:
        return false - stack has no space
    else:
        index of top pointer = value of x
        increment top pointer
        return true if successful

stack_pop(x):
    if stack is empty:
        return false - stack has nothing to pop
    else:
        decrement the top pointer
        value of x = value pointed by top pointer
        return true if successful
```

In the Huffman algorithm, the stack will be used when rebuilding tree after its tree dump. The declaration and initialization of the stack was included in the lab document. Implementing the smaller features of the stack such us popping and pushing is relatively simple. The return type of the push and pop functions of the stack are boolean. Since they do not need to return a specific value, instead they return *true* or *false* to tell if the function use was successful. It is important to note that in a stack, the top pointer is directed at the index one above the last item in the stack.

This means that when pushing items on the stack, the item is pushed to the top pointers location and then the top pointer is incremented. Whereas, when popping items from the stack, the top pointer is decremented first, then the item is removed.

I/O

```
read_bytes(infile, buffer, # of bytes):
    total = o
    bytes read = 0
    while ( bytes = read(infile) > 0) :
        if error while reading:
            return current total and exit
        total = total + bytes read
        buffer = buffer + bytes read
        if ( total = # of bytes):
            return total
    return total

write_bytes(outfile, buffer, # of bytes):
    total = o
    bytes written = 0
    while ( bytes = write(outfile) > 0) :
        if error while writing:
            return current total and exit
        total = total + bytes written
        buffer = buffer + bytes written
        if ( total = # of bytes):
            return total
    return total

read_bit(infile, bit):
    if (buffer index == 0 or buffer is full):
        reset buffer index
        number of bytes read = read_bytes()
        if number of bytes read < BLOCK:
            end of buffer = number of bytes read * 8 + 1
    bit = bit and buffer index
    buffer index += 1
    if buffer index == end of buffer
        return false
    return true
```

```
write_code(outfile, c):
    for bits in code:
        if bit is == 1:
            add bit to buffer at index
        else if bit == 0:
            add o bit to buffer at index
        increment buffer index

        if buffer gets full:
            write bytes(buffer)
            buf index = 0

flush_code(outfile):
    if buf index > 0:
        write bytes remaining in buffer
```

The above code outlines the input and output functions used to read in the Huffman algorithm. Since the read() function in C can sometimes stop reading into a buffer for whatever reason, the read_bytes implementation essentially makes it so that read_bytes is continuously called until the buffer is full. Then the number of bytes read is returned to verify. The write_bytes function works in almost the exact same way, except it writes to the outfile instead. The read_bit function is responsible for for reading into a buffer and returning each bit one at a time for as long as there are bits left to read. Once the number of bits has been written out it terminates. Finally, the write_code and flush_code function work together. The write_code function adds to a buffer each individual bit from the code. Once the buffer gets full, it printed to an outfile. When all the code bits have been read in, the function finishes. Any left over bits in the buffer that did not complete the buffer to be able to print out, are printed using the flush_codes function.

Huffman

```
build_tree(histogram[]):
    create a priority queue
    loop through the index i of the hist
        if the elment at i > 0:
            create node of hist[i]
            enqueue the node

    while (there are more than 1 node in the pq):
        dequeue left node
        dequeue right node
        join left and right node
        enqueue joined node

    root = dequeue the remaining node in the pq
    return root
```

```
build(root, code table[], c):
    if the children of root are NULL:
        table[root symbol] = code c
    else:
        if left child != NULL:
            push bit 0 to c
            recusrive call on build
            pop bit in c
        if right child != NULL:
            push bit 1 to c
            recursive call on build
            pop bit in c

build_codes(root, code table[]):
    create a code c
    call function build(root, table, c)

rebuild_tree(size of tree in bytes, tree dump[]):
    create a stack
    for i in size of tree:
        if tree[i] == char L:
            create a node with next char after L
            push to stack
            increment index i
        else if tree[i] == char I:
            pop right node from stack
            pop left node from stack
            join left and right nodes
            push joined node to stack

    root = pop last remaining node in stack
    return root

delete_tree(root):
    if children of root != NULL:
        delete node of root
    else:
        recusrive call delete tree on left child
        recursive call delete tree on righ child
```

The Huffman file contain the main functions for the algorithm. The build_tree function creates a node tree using a histogram that is passed in. The histogram is simply a list of all the characters available in the ASCII table. When a particular symbol is present, the count for it is increased. The build tree function takes this data to determine which symbols to turn int o leaf nodes and are joined to create a node tree. The root of the tree is then returned.

The build_codes function is similar in way. Instead of a histogram, the function takes in an empty table. Using the passed root node, the tree is traversed. If the left branch is taken, a 0 is added to the code, if a right branch is taken, a 1 is added to the code. Eventually when a leaf node is reached, the built code is assigned to its corresponding leaf node.

Rebuild_tree is required in order to build a tree from the tree dump. The tree dump is read into a buffer. While iterating over the buffer, if an 'L' is encountered, the code knows that the next char is a leaf node symbol and the node is added to a stack. If it encounters a 'I' the code determines that the node is a parent node and two nodes are popped out of the stack to join. Eventually only one node will remain, the rebuilt parent node.

Encode

The encoding process is the combination of all these functions put together. First a histogram must be created by reading the infile. Once the histogram is created, it is passed to build_tree in order to create the node tree and return the root node. The root node is then used in build_codes to create a code table. When the code table is successfully built, the header file is used to assign permission, pass the tree size, pass the file size, and ensure only the encoded file get decoded. The header is written out to the outfile. This soon followed by the writing of the tree dump. A post order traversal is used to efficiently do the dump. Finally, the infile is read again, and translated byte by byte using write_codes.

Decode

The decoding steps work backward to the encoding process. The header information is the first thing that is read. Once the file permissions are set and the data about the tree size and file size is received, the tree dump is constructed into a tree using rebuild_tree. This is followed by the reading of the codes bit by bit to travers the rebuilt tree and write out the nods that correspond to each. Once the number of bytes decoded matched the original file size. The reading process terminates, and the decoding process is completed.

Optimization:
There are possible rooms for optimization in my code. The biggest one is probably the conversion from an insertion sort to a min heap in order to implement a priority queue. This will indirectly result in a better tree formation as well, resulting in smaller codes for encoding and a better compression ratio.

Outcomes:

- Learned how to use low level I/O calls
- How to manipulate bits in order to create a coding scheme
- Implementing a tree using nodes
- How to make a priority queue
- ASCII representation in C
- Post order traversals
- Header files and how to deal with permissions