Assignment 6: Huffman encoding

Aniket Pratap

February 2022

1 Introduction

The purpose of this assignment is to compress data using the Huffman encoding method. This method is special because it uses a technique that allocates a fewer number of bits to frequent appearing symbols, and more bits to less frequent symbols. The end goal is to compress a message by using the least amount of bits as possible and by also not losing any information.

2 Process

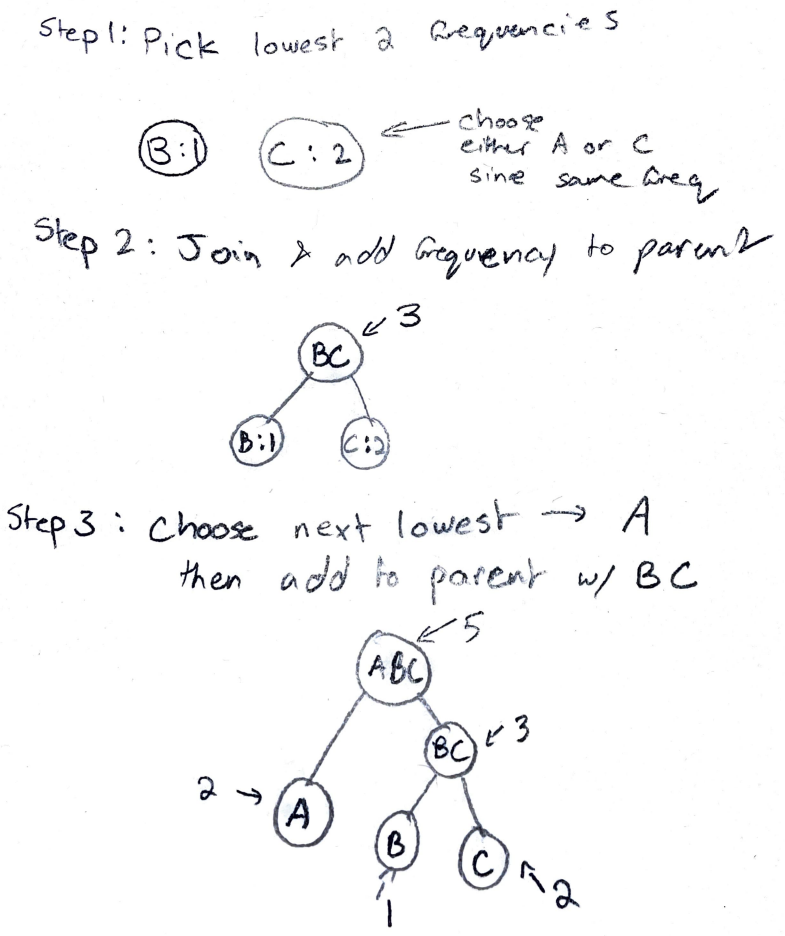
Step 1: reading in message

Lets say we have a message to compress: ccaba. The first thing to doing Huffman compression is to for a histogram of all the symbols involved in the message. In this case, the histogram would be something life:

- A: 2
- B: 1
- C: 2

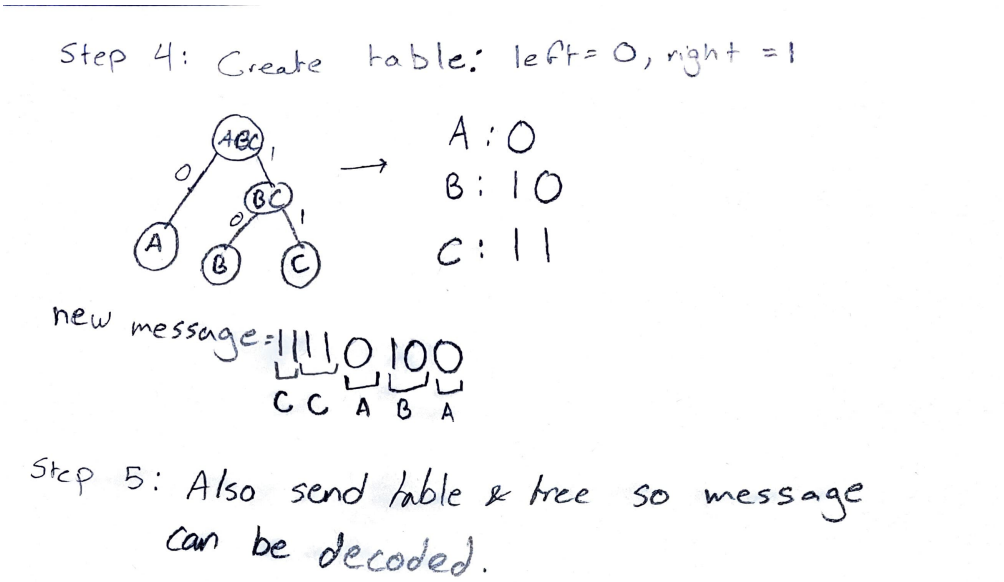
Step 2: Creating Tree

The next step is to create a Huffman tree. This can be done by taking the 2 fewest frequencies and joining them to create parent frequency that is the sum of both added frequencies. This process is then repeated until all the symbols are part of a tree. This process looks something like this:



Step 3: Create code and compress

The final step is to create the code by traversing down the tree and compressing the data along with the tree and code. If one takes the left branch, then the code will represent a 0, while the right branch represents a 1. This process will look something like this:



3 Pseudo-code

This file creates the nodes we need to use for each symbol

3.1 node.c

nodeCreate(symbol, freq)
creates a new node with a symbol and frequency

nodeDelete(node)
deletes node and sets to NULL

nodeJoin(left, right)
joins a left and right node to create parent node with a arbitrary symbol
frequency of parent = left freq + right freq

nodePrint(node)
prints node

3.2 pq.c

struct PriorityQueue
holds the capacity, head, tail, current size and Node array Q

maxChild(**Q, first, last)
gets the maxchild to create a max heap so it can be sorted at end

fixHeap(**Q, first, last)
fixes heap to max heap fashion

heapSort()
sorts maxheap so that larger numbers at end of array and smaller at front

pqCreate()
creates priority queue
initializes tail and head to 0
sets capacity of queue
allocates memory of size node capacity times

pqDelete()
deletes each node from priority queue

pqEmpty()
returns true if queue empty and false otherwise

pqFull()
returns true if queue full and false otherwise pqSize()
returns current size of queue

enqueue()
add element to queue
increases current size by 1
increase head by 1
if queue full, don't enqueue

dequeue()
remove highest priority element from queue
search for smallest element and swap with tail then remove tail
increase tail by 1
decerease current size by 1
if queue empty, don't remove

pqPrint()
debugging function suing print statements

4 code.c

codeInit()
initializes code

codeSize()
size is equal to top

codeEmpty()
empty if top is 0

codeFull()
full if top equal capacity

codeSetBit()
sets bit at index i
fails if i > 255

codeClrBit()
clear bit at index i
fails if i > 255

codeGetBit()
gets bit at index i
fails if i > 255

codePushBit()
pushes code to top
of code stack
increase size

codePopBit()
pop code from top
of code stack
decrease size

codePrint()
debugging function for code
prints in bytes
then use xxd on file

5 io.c

readBytes()
reads bytes from infile and return total bytes read
make this using read()
stores infile into a uint8t buffer

writeBytes()
similar to readBytes
get stuff from buffer and write to out file
write until all bytes done

readBit()
reads from the byte bit by bit using left shift
returns if bit is 0 or 1
use division by 8 for index of buffer
use mod 8 for shifting

writeCode()
stores code to buffer and flushes once buffer full
use same division by 8 and mod 8 logic from readBit()
This time left shift beccause want to || bit with buffer

flushCodes()
once write buffer full, get min bytes required to print then flush
don't need to write whole buffer

6 stack.c

This has a similar concept to code.c
struct Stack
creates the things neccessay for stack:
top, capacity, Node array

stackCreate()
initializes struct vars and allocates memory to stack
also allocates memory to Node array

stackDelete()
deletes stack and node array

stackEmpty()
empty if top == 0

stackFull()
full if stack == capacity

stackPrint()
simple debug function

prints eaech node in stack

7 huffman.c

buildTree()
creates priority queue

goes through histogram and creates nodes for each symbol
joins nodes and enqueue until 1 thing left in pq
dequeue the final node

buildCodes()
traverse tree to create code
push 0 then use recursion to travel to left node until leaf
back track and pop as you go
push 1 and travel to right
back track until all codes built

dumpTree()
traverse tree in post order and write all leafs and parents
if leaf, write L and symbol
if parent, write I
this uses recursion

rebuildTree()
traverse down tree dump until L or I
skip the symbol that comes after the L and move on to the next leaf
if L, then push symbol after it to stack
if I, pop first 2 and create parent by joining
repeat until 1 thing left in stack
the first thing popped should be right leaf because order pushed

deleteTree()
follows post order traversal to delete nodes
delete leafs first, then parents
use recursion for this

8 encode.c

usage()
same as previous assignments
output help

isSeekable()
checks if user used stdin for input (credit to Eugene)

hackyMkstemp()
if user used stdin
create temp file and set equal to default

main()

getopt()
searches fro user input specified by assignment
-h for help
-v for verbose statistics
-i for input (default stdin)
-o for output (default stdout)

define vars for encode.c
use for loop to create histogram for ascii
set 0th and 255th index to 1 for error handling

build tree based on computed histogram
build codes and store in code table while traversing through tree
use loop to count unique symbols in histogram
use fstat to get input file statistics
collect all info in a header typed Header (found in header.h)
write header to outfile file using writeBytes
dump tree to outfile using dumpTree
write codes using for loop and traversing code table and write code
flush codes
have verbose option for getopt
delete things allocated

9 decode.c

implemented same getopt, isSeekable, and hackyMkstemp from encode
read in header, and if magic number of encoded file doesn't match, stop
read in dumped tree and store in tree buffer
rebuild the tree using rebuildTree
read the codes; if 0, traverse left, if 1, traverse right
if leaf, write to outfile
things for verbose stats
delete allocated things