Dmitrius Agoston
Prof. Darrell Long
7 November 2021

CSE 13S Fall 2021
Assignment 5: Huffman Coding
Design Document

Description of Program:
The purpose of this program is to serve two main functions. Those functions being both encoding and decoding a file. The encoding of this program will first read an input file to be encoded. It will then find the Huffman encoding of its contents in order to compress the contents of the file. The decoding of this program has to receive a compressed file first that it is tasked with decoding. Once received, the decoder will take the file and decompress it, returning the file to its original size.

Layout/Pseudocode:

- node.c
  This file contains the necessary functions for creating and defining nodes and their elements that will be used in the other processes in this program. In particular the nodes are used when making Huffman trees. Huffman trees are composed of different nodes that contain pointers to a right and left child as well as the nodes symbol and frequency. In this file there are functions for creating and deleting nodes as well as joining them to find the parent node. There is also a print function to visualize this process.

  Create typedef struct Node Node

  Define node create with symbol and frequency
      Allocate memory for node
      Set symbol to symbol
      Set frequency to frequency
      Return node

  Define node delete with node
      Free memory allocated to node
      Set node equal to null

  Define node join with left and right
      Variable for parent node
      Set parent symbol to ‘$’

Set parent frequency to left + right
Return parent node

Define node print with node
Print contents of node

- pq.c
  This file contains the functions involved with the process of creating and maintaining a priority queue for this program. The priority queue that is being made is one filled with nodes. Unlike a normal stack, the elements in a priority queue all hold a value that determines when they should be dequeued. The actual process behind the functioning of the priority queue is implemented through a minimum heap sort. This way all elements can maintain their priority while also being efficiently searched through. This file also has functions capable of creating and deleting new queues along with checking the fullness of the quere. Functions for queueing and dequeuing and functions for getting the size as well as printing the actual queue are also added to help maintain its functioning throughout this process.

  Create struct priority queue (pq)
  Variable for nodes
  Variable for capacity
  Variable for tail

  Define swap with node x, node y
  Temporary node set to node x
  Set x node equal to node y
  Set y node equal to temporary node

  Define min_child with priority queue, first
  Variable for left set to 2 * first
  Variable for right set to left + 1
  If right is less than the tail and node right - 1 is less than node left - 1
  Return right
  Return left

  Define fix_heap_de with priority queue
  If tail is less than 2
  Return
  Variable for found set to false

```
        Variable for mother set to 1
        Variable for min set to the min child of mother
        Loop while mother is less than the tail / 2 and found is false
            If node mother - 1 is greater than node min - 1
                Swap mother and min
                Min set to min child of mother
            Else
                Found is true

Define fix_heap_en with priority queue
        If tail is less than 2
            Return
        Variable for found set to false
        Variable for child set to tail
        Loop while child is greater than 1 and found is false
            If node child - 1 is less than node child / 2 - 1
                Swap the two nodes
                Child set to half of itself
            Else
                Found is true

Define pq create with capacity
        Allocate memory for q
        Set capacity to capacity
        Allocate memory for nodes
            If this failed
                Free the queue
                Set queue to NULL
        Return q

Define pq delete with q
        Free memory allocated to nodes
        Free memory allocated to pq
        Set pq to null

Define pq empty with q
        If current size of q is equal to 0
            Return true
        Return false
```

```
Define pq full with q
    If current size of q equals capacity
        Return true
    Return false

Define enqueue with q and n
    If q is empty
        Enqueue node n
    If q is not full
        Add n to q
        Increment tail
        Fix heap
        Return true
    Return false

Define dequeue with q and n
    If q is not empty
        Decrement tail
        Set n equal to node zero
        Remove n from q
        Fix heap
        Return true
    Return false

Define pq print with q
    Print all the contents of q
```

- code.c

  This file contains the functions used in maintaining a stack whilst searching through a Huffman tree. The stack in particular is responsible for keeping bits that are then used to make a code. The code being created is meant to represent a stack of bits and functions in a way almost like a bit vector. That being said, this file is able to get the size of the stack as well as the fullness along with pushing and popping the stack. There are also functions that are capable of clearing, setting, and emptying the bit being used. A function for printing the code is also available to help identify if the files functions are working correctly.

```
Create struct code
    Variable for top
    Variable for bits with size of max code
```

```
Define code init
        Create new code
        Set top to 0
        For the size of max code size
                Set each position to 0

Define code size with c
        Return top value

Define code empty with c
        If top of c is equal to 0
                Return true
        Return false

Define code full with c
        If top of c is equal to size of bits
                Return true
        Return false

Define code set bit with c and i
        If i is less than ALPHABET
                Set temp variable to i mod 8
                Variable bit set to 1
                Bit shift bit by temp value
                Codes byte position “or”ed with bit
                Return true
        Return false

Define code clr bit with c and i
        If i is less than ALPHABET
                Set temp variable to i mod 8
                Variable bit set to 1
                Bit shift bit by temp
                Flip the bits of bit
                Codes byte position “and”ed with bit
                Return true
        Return false

Define code get bit with c and i
```

```
        If i is less than ALPHABET
            Set temp variable to i mod 8
            Variable bit set to 1
            Bit shift bit by temp
            And byte position with bit
            If that value is true
                Return true
        Return false

    Define code push bit with c and bit
        If c is not full
            If bit is equal to 0
                Clear bit
            Else
                Set bit
            Increment top
            Return true
        Return false

    Define code pop bit with c and bit
        If c is not empty
            Decrement top
            If get bit of top is 0
                Bit set to 0
            Else
                Bit set to 1
            Clear bit
            Return true
        Return false

    Define code print with c
        Print all the contents of c
```

- io.c

  This file contains the functions used to implement the analysis and editing of the given file to compress/decompress. The purpose of this file is to provide wrapper functions that make the process go along more smoothly and accurately. The functions in this file are responsible for the reading and writing for both bytes and codes. There are also variables that are defined in this file that are later used for

displaying the statistics of these processes. For example, the amount of bytes that were read and written.

```
Variable for bytes_read = 0;
Variable for bytes_written = 0;
Static variable buf_read[BLOCK] = { 0 };
Static variable index_read = BLOCK * 8;
Static variable buf_written[BLOCK] = { 0 };
Static variable index_written = 0;
Static variable bytes = 0;

Define read bytes with infile, buf, and nbytes
        Variable for reads
        Loop through infile and break if reads equals nbytes
                If reading the infile fails
                        Break
                Increment reads
        Add reads to bytes_read
        Return reads

Define write bytes with outfile, buf, and nbytes
        Variable for written
        Loop through outfile and break if written equals nbytes
                If writing to the outfile fails
                        Break
                Increment written
        Add written to bytes_written
        Return written

Define read bit with infile and bit
        If the index read is equal to BLOCK * 8
                Bytes set equal to read bytes of infile with block size and buf read
                Index read set to 0
        If index read is less than bytes * 8
                Variable curbit is set to 1
                Curbit is left shifted index read mod 8 times
                Curbits value is given to bit
                Index read is incremented
                Return true
        Else
```

Return false
Return true

Define write code with outfile and c
Variable beg set to index written
Loop through code size + beg starting from beg
If index written is greater than BLOCK * 8
Set index written to 0
Clear all values in buffer
Index set to i mod BLOCK * 8 / 8
If code get bit of c, i - beg
Buf written “or”ed with 1 bit shifted left i mod 8
Else
Buf written “and”ed with 1 bif shifted and flipped left i mod 8
Index written incremented

Define flush codes with outfile
Write out any leftover bits to outfile
Set any extra bits in last byte to 0

- stack.c
  This file contains the functions necessary for creating a stack. For this program in specific the stack is tasked with reconstructing a Hoffman tree for the deconstructor. This is done by storing nodes in the stack that are used in the reconstruction of the tree. Like a normal stack, this file is capable of creating and deleting new stacks, along with determining the size and fullness of the stacks created. It is also capable of pushing and popping nodes onto and off the stack. There is also a function used to print the stack and visualize the contents of the stack.

  Create struct stack
  Variable for top
  Variable for capacity
  Variable for items

  Define stack create with capacity
  Stack top set to 0
  Stack capacity set to capacity
  Stack items allocated dynamic memory
  If a stack is not available

```
            Free the stack
            Set stack equal to null
        Return stack

Define stack delete with stack
        If there is a stack with items
            Free the stack
            Set stack equal to null
        Return

Define stack empty with stack
        If the stack is empty
            Return true
        Return false

Define stack full with stack
        If the stack if full
            Return true
        Return false

Define stack size with stack
        Return stack size

Define stack push with stack and n
        If the stack isn't full
            Set the top of the stack equal to x
            Increment top
            Return true
        Return false

Define stack pop with stack and n
        If the stack isn't empty
            Set x = to the top of the stack
            Empty previous top value
            Decrement top
            Return true
        Return false

Define stack print with stack
        Loop through stack
```

Print each element in stack
Print new line

- huffman.c
  This file contains the functions necessary for building the Huffman trees that will be used for encoding and decoding. There is a function for building a Huffman tree that intakes a computed histogram and after being completely built, will return the root of the tree created. Another function implemented is building code that is used for the symbols in the huffman tree. There are also functions for editing a built tree, including, dump tree, rebuild tree, and delete tree. All three of these functions required a completely built tree in order to function.

  Define build tree with hist of size ALPHABET
  Create priority queue pq with the size of ALPHABET
  Create node root
  Loop ALPHABET times
  If current histogram position is greater than 0
  Create new node
  Enque new node
  Loop through priority queue until only 1 left
  Nodes for left and write
  Dequeue left first then right
  Create parent node by joining left and right
  Enqueue parent
  Dequeue last in priority queue
  Delete priority queue
  Return root

  Create static Code code

  Define build codes with root and code table of size ALPHABET
  If root left and root right are not made
  Table at root symbol set to code
  Else
  Temp = 0
  Push bit code 0
  Build codes for root left and table
  Pop bit at code temp
  Push bit code 1

Build codes root right and table
Pop bit at code temp
return

Define dump tree with outfile and root
8bit L set to L
8bit I set to I
If root
Dump the tree for left and right
If root left and write don't exist
Write bytes for L and symbol
Else
Write Bytes for I
return

Define rebuild tree with nbytes and tree dump of size nbytes
Create stack size of ALPHABET
Loop through nbytes times
If the tree dump at index is L
Create a node leaf
Push leaf onto stack
Increment index
Else if the tree dump at index is I
Create node left and right
Pop right and then left from the stack
Create parent node joining left and write
Push parent to stack
Create node root
Pop the last value on stack to root
Delete stack
Return root

Define delete tree with root
If root
Delete tree with root left and root right
Then delete node root
return

- encode.c

This file contains the functions used to encode and compress a given file. If the user requests this file will output a help message with given instructions on the usage of this part of the program. The user also has the option to set a designated location for both the infile and the outfile, defined in the command line. If this option is not chosen the user will have to use stdin to give the file and will receive the compressed result through stdout. There is also an option for the user to receive the statistics behind the process of encoding the given file and the bytes used to compress it. The actual encoding starts by creating a histogram of the file by counting the number of unique symbols that occurred in the file. Next, a Huffman tree will be created with the help of a priority queue. After the tree is built, a code table must be created in order to represent the symbols used in the tree and to give them values. Once these steps are executed, the file can start to be encoded and in the end will be printed to the desired outfile.

```
Define help
    Print help statement

Define main
    Variable for opt
    Variable for infile set to 0
    Variable for outfile set to 1
    Boolean for stats
    Parse through command line
    If h entered
        Print help statement
        End program
    If v entered
        Enable printing stats
    If i entered
        If opening the file does not equal null
            Set file to infile
        Else
            Print error and end program
    If o entered
        If opening the file does not equal null
            Set file to outfile
        else
        Print error and end program
    Compute histogram of infile
        Do so by counting number of occurrences of each symbol
```

Make sure to add 1 to first and last element
Create root node by building huffman tree
Create code table the size of ALPHABET
Build the codes given the tree and table
Loop through the histogram and collect the number of unique symbols
Assign correct values to all of the header values
Write the header to the outfile
Dump the tree to the outfile and write it
Re read through the infile and write corresponding codes
Flush codes for leftover
If statistics are desired print statistics
Close files and free memory used
Return 0

- decode.c
  This file contains the functions used to decode and decompress a given compressed file. Much like the encode file, this file also contains a help message that will display the usage of this part of the program to the user. It is also capable of receiving both an infile and outfile for the decompression process. If no files are given the user will give the file through stdin and receive the results through stdout. There is also the option to receive the bytes used in the decompression process. The process will start by reading the Huffman tree that has been dumped by the infile and reconstructing it. A stack of nodes is necessary for this process to be complete. Once rebuilt, the tree will be traversed and the original file will be printed to the desired outfile, bit by bit.

  Define help
  Print help statement

  Define main
  Variable for opt
  Variable for infile set to 0
  Variable for outfile set to 1
  Boolean for stats
  Parse through command line
  If h entered
  Print help statement
  End program
  If v entered
  Enable printing stats

If i entered
    If opening the file does not equal null
        Set file to infile
    Else
        Print error and end program
If o entered
    If opening the file does not equal null
        Set file to outfile
    else
    Print error and end program
Build the header and temporarily set values to zero
Create buffer variable
Loop through header of infile and set corresponding values to
The variables
Check if the magic number is correct
Fill the tree by reading bytes from infile
Use the data to rebuild the tree
Decompress the file by reading and writing bits
    Do this with the rebuilt tree
If stats desired, print out stats
Free memory
Close files
Return 0

Error Handling:
When handling errors for this program, the main errors occurred in regards to user/file input. There had to be checks for valid files being used to actually be compressed. Along with that there had to be permission given via the magic number to actually decompress the encoded file. Some other errors that had to be handled involved how memory was being allocated. The program has to appropriately allocate memory for certain variables and at the end of running has to free up the space it used for those variables.

Credit:
When creating this code I largely used asgn5.pdf as my main source of reference for creating this program.