APP NAME: HUFFMAN CODING

Design Specification

SOFIA PETROVA, UC Santa Cruz, CSE 13S – Fall 2021
Oct. 25. 2021

Document Version: 2.0
Application Version: 2.0

Accepted by:

Darrell Long

NAME	Date
TITLE	

NAME	Date
TITLE	

NAME	Date
TITLE	

Table of Contents

1. Amendment History

November 5, 2021- Edits to add pseudocode and modify system architecture description

November 7, 2021- Polishing descriptions and submitting

Date	Doc. Version	Amendment Description
10/25/2021	1.0	Initial draft, Sofia Petrova
11/5/2021	2.0	Final draft, Sofia Petrova

2. References

Information contained in this document was based on the following documents:

Assignment 5

Huffman Coding

Prof. Darrell Long

CSE 13S – Fall 2021

3. Acknowledgments

The information presented in this document was drawn from various discussions with the following individuals:

- Darrell Long

4. Included files

- encode.c- contain implementation of the Huffman encoder.
- encode.h- contains functions to help complete encoding functions
- decode.c- contain implementation of the Huffman decoder.
- defines.h- contains the macro definitions used throughout the assignment.
- header.h- contains the struct definition for a file header
- node.h- contains the node ADT interface
- node.c- contains implementation for the node ADT
- pq.h- contains the priority queue ADT interface
- pq.c- contains the priority queue ADT implementation
- code.h- contains the code ADT interface
- code.c- contains the code ADT implementation
- io.h- contains the I/O module interface
- io.c- contains the I/O module implementation
- stack.c- contains implementation for the stack ADT
- stack.h- contains the stack ADT interface
- huffman.c- contains implementation of Huffman coding module interface
- huffman.h- contains Huffman coding module interface
- README.md contains instructions for compiling and running the code
- DESIGN.pdf contains the design and design process for the program, describing it well enough that the program should be replicable to someone who has not seen the code. This is the current document
- Makefile compiles, formats, and runs the code, as well as cleaning output files

NOTE: Many file descriptions are taken directly from the Assignment 5 document written by Darrel Long for CSE13S Fall as they best describe each file.

5. Design Overview

5.1. Purpose of Document

This program contains a Huffman encoder and decoder. The encoder builds a priority queue full of symbol frequencies and then, out of that, a Huffman tree. It then creates a stack of bits based on the tree. Both the tree and bits are dumped into the output file.

The decoder then rebuilds the Huffman tree by traversing the tree dump and rebuilding the leaves and parents. It will use the bit stack to traverse the rebuilt tree and output a leaf whenever reached.

5.2. System Architecture

5.2.1. Overall Structure

This program contains 3 Abstract Data Types: Codes (which implement a stack of bits), a priority queue; nodes, and a stack. It also contains modules to implement Huffman Tree functions and a module for input/output. A histogram array is created in the beginning of the encoding process. Then, for each symbol in the file, its corresponding frequency is incremented on the histogram. This histogram is then used to create nodes, which are then added to a priority queue. The priority queue is then used to create the Huffman tree, which is written into the output file. A code is also created based off the histogram and outputted in the file.

The test harness takes arguments from the command line while running the executables, and depending on the arguments given by the user, the program will encode or decode the given file to and from the appropriate files.

encode.c test harness arguments:

-h	display program help and usage.
-i infile	Specifies the input file to encode. The default input is stdin.
-o outfile	Specifies the output file write compressed output into. The default output is stdout
-v	prints decompression statistics to stderr, including uncompressed file size, compressed file size, and space saving.

To decode, the tree dump is read from the encoded file and reconstructed into a proper Huffman tree. The codes written underneath the tree dump are then used to traverse the tree and output the read bytes to a new file.

decode.c test harness arguments:

-h	display program help and usage.
-i infile	Specifies the input file to decode. The default input is stdin.
-o outfile	Specifies the output file write decompressed output into. The default output is stdout
-v	prints decompression statistics to stderr, including compressed file size, decompressed file size, and space saving.

5.2.2. ERROR HANDLING

- For inputting an invalid argument (including arguments -i or -o without a specified file)
    - Output the help screen normally outputted by -h
- For inputting an invalid file (failure to open an input or output file)
    - Print error message and exit with error code 1
- For memory allocation failures
    - Print error message and exit with error code 1
- For writing or reading text or files that are too small
    - Print error message and exit with error code 1

6. Graphical Representation of Tree

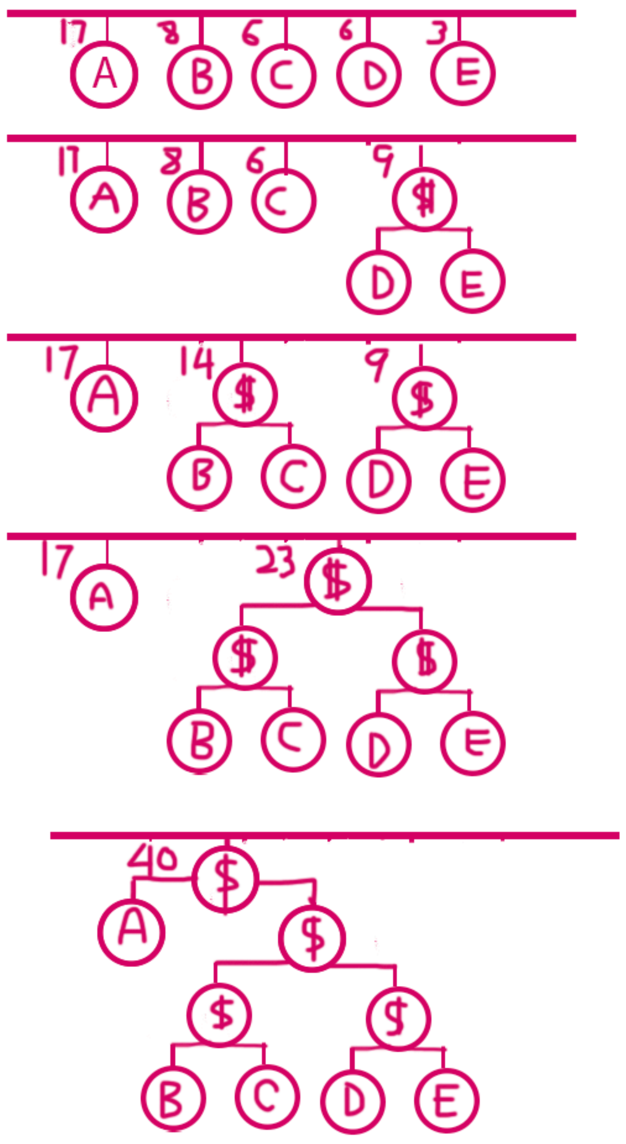

Figure 1: Representation of nodes in priority queue being enqueued onto Huffman tree with a completed tree by the end.

7. Pseudocode Implementation

6.1 Encode function

```
Create Histogram array
Set first and last array elements to be incremented
For each element in the array
     If element isn’t 0
          Create node
          Enqueue node onto priority queue
Create Huffman tree
     While there’s more than one node in the queue
          Pop last two nodes on queue
          Join nodes
          Put joined node and leaves on Huffman tree
          Put joined node on queue
Create code
For each element in the alphabet
     Build code by
          Checking node to see if it has left or right
          Writing 0 if left and 1 if right, then calling recursively
with the left and right nodes
Open outfile
Dump tree into outfile
For each symbol in the input file
     Write code into output file
```

6.2 Decode function

```
Allocate memory for file input names
Read magic number
Get input file permissions and set them to output file
Rebuild tree:
    For every node in the tree dump
        Traverse until hitting a leaf
            Push leaf onto stack
        If hitting an I
            Pop stack twice
            Rejoin nodes popped
            Put back on Huffman tree
            Pushed joined node onto stack
For every character in the emitted binary
    Traverse the Huffman tree
    If reaching a leaf
        Output the symbol to file
Delete tree
Free memory
```

6.3 Node ADT

```
Node Creation
    Allocate memory for node
    Set left of node to null
    Set right of node to null
    Set symbol to given symbol
    Set frequency to given frequency

Node Delete
    Free memory for node
    Set pointer to null
```

```
Node Join
    Allocate memory for parent node
    Set left node of parent node to given left node address
    Set right node of parent node to given right node address
```

6.4 Code ADT- Functions very similarly to stack ADT with extra set bit functions

```
Initialize Code
    Create new code
    Set top to 0
    For all bits in the code
        Zero out the code
    Return the new code

Code Empty
    If code is empty
        Return true
    Else return false

Code Full
    If code is full
        Return true
    Else return false

Set Bit
    If code is full return false
    Bitmask to i-th bit of a byte, set to 1

Clear Bit
    If code is full return false
    Bitmask to i-th bit of a byte, set to 0
```

```
Get bit
    Bitmask to i-th bit of a byte and return true if it’s there

Push bit
    If code full
        Return false
    Else
        Push 0 or 1 onto stack depending on what’s specified
        Increment top of stack

Pop bit
    If code full
        Return false
    Else
        Decrement top of stack
        Pop 0 or 1 onto stack depending on what’s specified
```

6.5 Priority Queue ADT

```
Priority Queue struct
    int capacity
    Node array address the side of the max tree size
    Count/number in queue

Create Priority Queue
    Allocate memory
    Set count to 0
    Set capacity to given capacity

Enqueue
    Add given node onto the last index of the priority queue
```

```
     Fix heap
     Increment count

Dequeue
     Store first element of the queue
     Set first element to last element
     Decrement count
     Fix heap

Delete Queue
     Free memory
     Set pointer to null
```

6.6 Input/Output Module

```
Read bytes
     Initialize characters read
     While the number of characters being read is less than bytes to
read
          Read from infile

Flush codes
     Write extra bytes to output file

Dump tree
     If root node has a left node
          Dump tree to outfile with left node as argument
     If root node has a right node
          Dump tree to outfile with right node as argument
     If neither
          Write L for leaf
     Else write I for interior
```

```
Write code
     For code size
          Write code bytes to output file
Write bytes
     Write given bytes to outfile using Linux write()
```