Assignment 6

Huffman Coding
DESIGN Document

Description of the program

In computer sciencee, a Huffman code is a particular type of optimal prefix code that is commonly used for lossless data compression. The following implementation will be used as means to compress a file:

Step 1: Input file
Step 2: Read input file
Step 3: Find the Huffman encoding of the contents of the file
Step 4: Use the encoding to compress the file.

Files to be included

Source and header files:

1. **encode.c** contains the implementation of the Huffman encoder
2. **decode.c** contains the implementation the Huffman decoder
3. **defines.h***(Provided)***:** contains the implementation of the macro definitions used throughout the assignment
4. **header.h***(Provided)*: specifies the implementation for number theory functions
5. **node.h***(Provided)*: interface for note ADT
6. **node.c:** implementation of note ADT
7. **pq.h***(Provided)***:** contains the priority queue ADT interface
8. **pq.c:** Contains the implementation of the priority queue ADT interface
9. **code.h***(Provided)***:** Specifies the interface of the code ADT
10. **code.c**: Specifies the implementation of the code ADT
11. **io.h***(Provided)***:** Specifies I/O module interface
12. **io.c**: Specifies I/O module implementation

13. **stack.h**_(Provided)_**:** Specifies stack ADT interface
14. **stack.c**: Implements stack ADT library
15. **huffman.h**_(Provided)_: Specifies Huffman coding module interface
16. **huffman.c**: Implementation of Huffman coding module

Additional Files:

1. **Makefile:** formats all source code, including the header files.
2. **README.md:** Description of how to use my program and Makefile. It also includes any command-line option that my program accepts. Any false positives reported by scan-build should be documented and explained here as well. Note down any known bugs or errors in this file as well for the graders.
3. **DESIGN.pdf (_This file_):** The design document describes the preliminary design and design process for my program with sufficient detail for potential replication
4. **WRITEUP.pdf:** Analysis and description of the produced program, as well as graphs generated and/or information, gathered from outputs

Pseudocode/Structure:

Encode.c - command-line options:

- h: prints out a message describing the purpose of the program
- i infile: Specifies the input file
- o outfile: Specifies the output file
- v: prints compressions statistics

Decode.c - command-line options:

- h: prints out a message describing the purpose of the program
- i infile: Specifies the input file
- o outfile: Specifies the output file
- v: prints decompressions statistics

node.c:

node_create(symbol, frequency):

Node constructor

node_delete(node):

Node destructor

node_join(Node_left, Node_right):

Joining right and left nodes

node_print(node):

Debug function to verify that notes are created and joined correctly,

<u>pq.c:</u>

pq_create(capacity):

constructor for priority queue.

pq_create(queue):

destructor for priority queue

pq_create(queue):

Returns true if queue is empty false otherwise.

pq_size:(queue):

Returns number of items in queue

enqueue(queue, node):

Enqueues a node into the priority queue

dequeue(queue, node):

dequeues a node from the priority queue

pq_print(queue):

Debug functions, that prints the information regarding the priority queue

<u>code.c:</u>

Code_init:

Create a new “Code” on the stack

Code_size(c):

Returns the size of the “Code”

Code_empty(c):

Returns true if “Code” is empty false otherwise.

Code_full(c):

Returns true if the “Code” is full

Code_set_bit(c, i):

Set the bit index i in the code, setting it to 1.

Code_clr_bit(c, i):

Clears the bit at index i int the Code, clearing it to 0

code_get_bit(c, i):

Gets the bit at index i in the “Code”

code_push_bit(c, bit):

Pushes a bit onto the “Code”

code_pop_bit(c, bit):

Pops a bit off the “Code”

code_print(c):

Debug functions to help verify if the code is working or not.

<u>io.c:</u>

read_bytes(infile, *buf, nbyts):

Wrapper functions that performs reads

write_bytes(outfile, *buf, nbytes):

Wrapper function that performs writes

Read_bit(infile, bit):

Read a block of bits

write _code(outfile, c):

Each bit in the code c will be buffered into the buffer.

flush_codes(outfile):

Flushes code from the buffer

<u>Stacks.c:</u>

stack_create(capacity):

Stack constructor

Stack_delete(stack):

Stack destructor function.

Stack_full(stack):

Return true is stack is full, fsale otherwise

stack_empty(stack):

Returns true if stack is empty false otherwise.

stack_size(stack):

Returns the number of nodes in the stack

stack_push(stack, n):

Pushes a node onto the stack

stack_pop(stack, n):

Pops a node off the stack

stack_print(c):

Debug functions to help verify if the stack is working or not.

Huffman.c:

build_tree([alphabet]):

Constructs a Huffman tree given a computed histogram.

build_codes(root, [alphabet]):

Populates a code table, building the code for each symbols in the Huffman tree.

dump_tree(outfile, root):

Conducts a post-order traversal of the Huffman tree rooted at root, writing it to outfile

rebuild_tree(nbytes, tree_dump[static nbytes]):

Reconstructs a Huffman tree

delete_tree(root):

Destructor for a huffman tree

Credit

- I took the pseudocode from the assignment 6 paper written by prof. Darrel Long