Assignment 6
Huffman Coding

DESIGN DOCUMENT

Zack Traczyk
CSE13S - Spring 2021

Due: May 23th at 11:59 pm

1 Objective

Implement static lossless file compression using Huffman encoding.

2 Given

- Header files for definitions, header, node, priority queue, code, io, stack, Huffman
- C file for entropy

3 Data Structures

This assignment required a couple implementing three different datatypes to work: nodes, a stack, and a priority queue. The implementation for these abstract datatypes will be described in detail below.

3.1 Nodes

Nodes are implemented in `node.c` and follow the assignment document specifications. As given in the document, a node is a structure that holds pointers to its left and right child, an 8-bit unsigned integer for its symbol, and a 64-bit unsigned integer for its frequency. Nodes are used to construct a Huffman tree which is required for encoding.

Nodes are used to create a tree and as such require a couple of functions to interface with them:

- `node_create`: creates a node given a symbol and frequecy. Sets the left and right children to `NULL`.

- `node_delete`: deletes a node by freeing allocated memory and setting its pointer to `NULL`.
- `node_join`: joins to nodes given their pointers under a parent node with a symbol of '$' and a frequency equal to the childrens the sum of the left and right frequencies.
- `node_print`: this function is not utilized during encoding or decoding but is helpful for debugging

This functions are all only a few lines, and therefore are trivial to implement given the explanations above.

3.2 Stack

Stacks are implemented in `stack.c`. A stack is a structure that contains an array of pointers to Nodes (the elements the stack holds), an unsigned 32-bit integer to the next empty value in the stack, and an unsigned 32-bit integer that stores the total number of nodes that can be stored.

A stack can be popped from and push to, and its size can be tested. Therefore the following functions are implemented for a stack:

- `stack_create`: creates a stack given its capacity (how many elements it can hold).
- `stack_delete`: deletes a stack by freeing allocated memory and setting its pointer to `NULL`.
- `stack_empty`: since a stack keeps track of the top element to know where to push/pop a value, a stack is therefore empty when the `top` is 0
- `stack_full`: a stack is then full when the `top` is equal to the capacity
- `stack_size`: since new Nodes are inserted from `top`, the stack size is equal to `top`.
- `stack_push`: to push a new element, there must first be room. This is checked by calling `stack_full` and returning false if `stack_full` returns true. If there is room in the stack, the item is added at index `top` and `top` is then incremented.
- `stack_pop`: when elements are popped the stack checks first if there is anything to pop using `stack_empty`. If there are elements, `top` is decremented and the pointer to a Node at the now decremented `top` is passed to a Node pointer given as a parameter.
- `stack_print`: this function is not utilized during encoding or decoding but is helpful for debugging

3.3 Priority Queue

A priority queues is implemented in `pq.c` using a min heap. A priority queue is a structure that contains an array of pointers to Nodes (this is where elements that are enqueued and dequeued are stored), an unsigned 32-bit integer to the next empty value in the min heap, and an unsigned 32-bit integer that stores the total capacity of the queue.

- `pq_create`: creates a priority queue given its capacity (how many elements it can hold).
- `pq_delete`: deletes a pq by freeing allocated memory and setting its pointer to `NULL`.
- `pq_node_frequency`: this is a helper function to help with enqueuing and dequeuing, it is not publicly accessible. All this function does is given a pointer to a priority queue and an index, it calls `node_frequency` on that Node and returns the value.
- `pq_empty`: a priority queue is similar to the `Stack` datatype and is therefore empty when the `top` is 0
- `pq_full`: a priority queue is then full when the `top` is equal to the capacity
- `pq_size`: the priority queue size is equal to `top`.
- `enqueue`
- `dequeue`
- `pq_print`: this function is not utilized during encoding or decoding but is helpful for debugging

3.4 Codes

Huffman compression works by assigning codes to symbols where the most used symbols have the shortest codes. These codes are implemented using a structure containing an unsigned 32 bit integer to track the top of the code, and an array of unisgned 8 bit integers to store the bits in the code like a stack. Since elements in a code are bits stored inside array elements, `top` tracks the top bit not the top index. Codes are constructed by traversing the tree left = 0 and right = 1 until a leaf is found at which the order of lefts and rights, 1s and 0s is the code. This is explained further in subsection 4.2 on the following page. However, this functionality dictates that a code needs to be able to pop and push bits as well as check the size.

The following functions are implemented in `code.c`:

- `code_create`: creates a code and initializes it to 0. The code structure is defined globally so its members can be accessed. The size of a code is constant so there is no need to allocate space at runtime
- `code_delete`: deletes a code by freeing allocated memory and setting its pointer to `NULL`.

- `code_empty`: since a code keeps track of the top element to know where to push/pop a value, a stack is therefore empty when the `top` is 0
- `code_full`: a code is then full when the `top`$/8$ is equal to the capacity (`MAX_CODE_SIZE`)
- `code_size`: the code size is equal to `top`.
- `code_push`: to push a new element, there must first be room. This is checked by calling `code_full` and returning false if `code_full` returns true. If there is room in the stack, the new bit is inserted in the array at the `top`$/8$ element at the `top`$\%8$bit.
- `code_pop`: when elements are popped the stack checks first if there is anything to pop using `code_empty`. If there are elements, `top` is decremented and the pointer to a bit at the now decremented `top` is passed to a Node pointer given as a parameter.
- `code_print`: this function is not utilized during encoding or decoding but is helpful for debugging

4 Programs

Before creating the two required programs, encode and decode, some more helpful functions needed to be implemented to help with io functionality and to assist with the encoding and decoding.

4.1 IO

Two global variables are defined in `io.c`: `bytes_read` and `bytes_written`. These variables track how many times bytes have been read and written to/ from a file. Additionally, io needs to be handled for six main functions: reading bytes, writing bytes, reading the header, writing the header, reading a bit, and writing codes.

Reading and writing bytes and the header are almost identical. In *alg.* (1) on the next page read functionality is shown but write functionality is identical and header reading and writing only requires the size of the read/write call to be equal to the size of the `Header` structure.

4.2 Huffman Trees

Huffman trees utilize a few different functions when encoding or decoding. These functions mainly deal with constructing, deleteing, reconstructing, and traversing Huffman trees.

4.3 Encode

- -h : Command line options
- -i `infile` : The file containing bytes to be encoded (default is `stdin`)

```
bytes read = 0;
last bytes read = 0;
while (bytes read += read(from infile to buffer for nbytes) is less than bytes to read
 do
    if last read is equal to bytes read then
        Break;
    end
    last bytes read = bytes read;
end
global bytes read += bytes read;
return bytes read
```

Algorithm 1: Basic Reading

- -o `outfile` : The output file to store encoded bytes (default is `stdout`)
- -v : Prints compression statists of the encoding process to `stderr`

4.4 Decode

- -h : Command line options
- -i `infile` : The file containing bytes to be decoded (default is `stdin`)
- -o `outfile` : The output file to store decoded bytes (default is `stdout`)
- -v : Prints statists of the decoding process to `stderr`

4.5 Entropy

The source code for this program (`entropy.c`) is provided in the class resources repository. Entropy calculates the entropy, or variation of contents, for a file.

5 Parse

Program arguments are parsed and stored using flags. A set is not used since the order of the inputs is irrelevant and the parsed data will ultimately be stored in variables anyway.

After program arguments are parsed the input file is parsed with encode or decode. Execution happens with every grab of a byte to the buffer. Refer to execute for how the algorithm works.

6 Execute

6.1 Encode

First, the infile is checked to see if it is seekable. Since Huffman encoding requires two passes, the file must be read twice. This is impossible for piped input, therefore a tempfile is made whenever the input is unseekable. Compressing a file requires a few steps. First, a histogram is constructed of the most used characters. If a tempfile has been initalized, then the parsed characters and then written to the tempfile to be read on the second pass. Once each character has a frequency, the histogram elements are converted to nodes and enqueued into a priority queue. Then, two nodes are popped off at at time and joined together and queued back into the priority queue until only one node remains. This last node is the root to the Huffman tree. Finally the Huffman coding happens as the tree is traversed and codes are assigned to all the characters. This pseudocode for this procedure can be seen in *alg.* (2) on the following page.

6.2 Decode

Decode is (unsurprisingly) the opposite of encode. Codes are read from the file and decoded.

```
if not seekable then
    create tempfile;
end
call fstat on infile and assign to statbuf;
if filein is not stdin and fileout is not stdout then
    modify permissions of outfile to match infile;
end
initalize a histogram of length ALPHABET;
histogram at 0 is 1;
histogram at 255 is 1;
initalize read buffer;
while length from read_bytes(read from filein to buffer with size of BLOCK) is greater
 than 0 do
    if tempfile exists then
        write_bytes(wrte to filout from buffer for BLOCK bytes);
    end
    for every read byte in buffer do
        increment corresponding histogram array element to buffer charater;
    end
end
root of Huffman tree = build_tree(from histogram);
create a header and write to fileout;
initalize a table with empty codes;
build_codes(root, from table);
write the tree to outfile;
if tempfile used then
    set filein to tempfile;
end
while length from read_bytes(read from filein to buffer with size of BLOCK) is greater
 than 0 do
    for every read byte in buffer do
        write_code(fileout, from code table of the current character);
    end
end
print statistics if verbose;
close files and free allocated space;
```

Algorithm 2: Encode

Algorithm 3: Decode