Jessie Zhu Tuesday, May 11, 2021

Assignment 6: Huffman Coding

Design Document

For this assignment, we will be implementing a Huffman encoder and decoder to encode and compress a file and decode and decompress a file. Our code should be lossless compression and decompression. Statistics about the compression and decompression will be printed as well.

The purpose of this assignment is learn how to implement a Huffman encoder and decoder. That encompasses using nodes for the Huffman tree and priority queues to link the nodes, both are key aspects of making the encoder and decoder work. We will also be making our own io function for reading and writing files.

Design Process/Questions I need answered:

- The io.c functions were the hardest to implement because it was least familiar
- Logic of how encoding and decoding weren't challenging once the ADTs were all constructed
- Originally my scan-builds were showing up with errors of dereferenced null inside my recursion functions. They were easily fixed with an extra condition for the node pointer when traversing left and right.
- In my pq, I didn't realize that my struct needed **item rather than *items because the items were of node type. This also showed up on my scan-build and was fixed as well after changed the *items to **items.
- The way I originally planned to do pq was to enqueue low (zero index) to high priority (highest index) and then pop from the highest index. That way I would avoid wrapping the queue. But then I realized that this would no longer be a queue since I don't enqueue and dequeue from the different ends. So I changed this.
- Overall, I thought this project wasn't too challenging. There was just a lot of things to make and if something wasn't working, it takes a lot of time to figure what isn't working.

Jessie Zhu   Tuesday, May 11, 2021

Pseudocode:

Encode

In encode, we will be reading the file and keeping track of the occurrences of each symbol. Then a Huffman tree will be created using the number of occurrences and printed in output and a code table will be created where the index of the table represents a symbol of the file. For each symbol in the file, the corresponding code from the table will be outputted into the output file. Statistic about the compression will be printed if v is selected. Because

```
Encoder:
main:
    take program argument with getopt, and set appropriate infile, outfile, v (bool)
    for every character in file:
        compute histogram, count occurrence of the character
    use temp file if input is stdin to hold input
    Create Huffman tree using histogram (will use priority queue)
    Create code table where each index of table represents a symbol.
    Emit encoding of tree to a file, (separate recursion function is probably good)
    Go through symbols of the input again, for each symbol emit code to output
file
    if v selected, print the statistics for file compression
```

Decode

Decode will undo what encode does. Reading the file will first give us the header Huffman tree that will be used to decode out file. Then transversing the tree will give us the decoded symbols for output. Repeat until all symbols of the output have been decoded. Statistics about decompressed file size will be printed if v is selected.

Decoder:

```
main:
        take program argument with getopt, and set appropriate infile, outfile, v (bool)
        read header
        read the dump tree to reconstruct Huffman tree
        read the rest of the file, for decoded item in file:
                use the Huffman tree to decode the symbol, traversing down the tree
                when a lead node is reached, the symbol has been found
                print symbol to output
        if v selected, print stats for decompressed file
```

ADTs

We will be using nodes for the Huffman tree.

Node ADT:

Each node will have left and right node, symbol, frequency

- node_create: allocate space, set symbol, frequency
- node_delete: deletes the left and right node, frees the pointer
- node_join: creates a new parent node with symbol $ with children left and right
- node_print: prints out all nodes

Priority Queue will keep track of the the nodes, dequeue elements of least importance

Priority Queue ADT:

- pq_create: allocate space, set capacity of queue
- pq_delete: deletes the queue, sets pointers to null
- pq_empty: check if queue is empty, return status

- pq_full: check if queue is full, return status
- pq_size: returns number of items in queue
- pq_enqueue: add node to queue in order of priority, shift other items if necessary
- pq_dequeue: removes the item of least priority (one of the ends of the queue depending on how I decide to implement it)
- pq_print: prints the nodes in order of priority (should already be in order so just print queue normally)

Codes will be used to traverse through the tree and create a code for each symbol

Codes ADT:

Has attributes top, and bits array of max_code_size

- code_init: will not have memory allocation (new code is created on the stack, setting top to 0, and zeroing out the bits)
- code_size: returns the number of bits pushed onto the code
- code_empty: returns whether or not code is empty
- code_full: returns whether or not the code is full, since max length of code is 256
- code_push_bit: pushes bit onto the code, return true if successful, false if failed
- code_pop_bit: removes item from code stack, pass through pointer. Returns true if successful, false if not.
- code_print: prints all bits

I/O will be used instead of low level system calls for reading and writing in encode and decode.

I/O:

- read_bytes - read bytes until buffer is full or end of file is reached. Number of bytes read is returned.
- write_bytes - write bytes until end of buffer. Number of bytes written to outfield is returned.
- read_bit - read bytes into buffer and get every single bit individually for the whole file.

- write_code - buffer each bit in code c and when buffer is full, write into outfile
- Flush_code - emptys buffer into outfile

IO functions

```
read_bytes:
    total read = 0, current read = 1
    while number of currently read bytes is not zero and and we have not read enough bytes yet:
        read bytes
        increment total read
    return total read
write_bytes:
    total write = 0, current read = 1
    while number of currently written bytes is not zero and and buffer not emptied
        increment total read
    return total write
read_bit:
    if buffer empty, fill buffer
    get bit at buffer index
    pass back bit
    increment buffer index
    if bit passed back was the last bit +1, then bit counldn't have been read, return false, otherwise return true;
write_code:
    for bit in current code:
        put the bit into the buffer at index array using set or clear bit
        increment buffer index and if buffer is full, write out buffer and reset index
flush_codes:
    if buffer not empty, write out remaining bits of the buffer using write_bytes
```

Stacks:

Identical to stacks used before, except items in stack are nodes.

Huffman Coding Module

- build_tree - takes in histogram (array) and creates a tree.
- build_codes - for each symbol in Huffman tree, the constructed code for the symbol is added into code table, which has one index for each possible symbol
- rebulid_tree - rebuilds a Huffman tree given tree dump. Length of tree dump is given and returns the root node of the rebuilt tree
- delete_tree - post order traversal of tree to free all nodes. Set pointer to null

Huffman:

```
build_tree:
    create pq of ALPHABET size
    for every unique character in histogram:
        create node and enqueue
    while size of pq is less at least 2:
        dequeue 2 nodes: left and right
        join left and right into parent node
        enqueue parent
    dequeue and return root node
build_codes:
    create static code
    if current node is leaf:
        set symbol in table to code
    else:
        push 0 to code, recurse left, pop
        push 1 to code, recurse right, pop
```

Huffman, continued:

```
rebuild_tree:
    create stack for nodes
    for byte in tree dump:
        if byte is L:
            then create node with symbol following L
            push node to stack
        else if byte is I:
            pop 2 nodes: right and left
            join nodes left and right under parent node and push parent to stack
    pop and return root node


delete_tree:
    if current node is leaf:
        delete node
    else:
        recurse delete_tree node left
        recurse delete_tree node right
        delete current node
```