Natalie Valett, nvalett
CSE13S
Prof. Darrell Long
April 19, 2021

Asgn6 Design: Huffman Coding

Summary:

A huffman code is a data-compression algorithm designed to store the same information using less bits than would otherwise be needed. It utilizes a Huffman tree to map each character present in the input file to new character codes. The result is that the most commonly used characters will have shorter character codes due to their closer proximity to the tree's root. With this being true, the amount of bits needed to represent characters is inversely proportional to their frequency within the text, so the most common characters can be represented using significantly less space.

Encoder:

The encoder does this by reading in a file and making a histogram which tallies the frequency of each character present in the file. We'll do this with an array whose indices correlate to ascii values 0-255, and whose values stored at each index indicates the number of appearances of that character in the input file. Then we will create a priority queue which will store all the characters in the histogram whose frequency is non-zero. The priority queue should store the character in ascending order in relation to frequency, so the least common characters should come first, and the most common should be last. that is meant to represent a Huffman Tree of the characters.

Decoder:

The decoder takes in the input and read in the header information, confirming the magic number, and setting properties/permissions. The decoder then has the task of reading the dumped tree from the infile and reconstructing it using rebuild_tree(). Once the tree is reconstructed, the decoder is then able to read in the encrypted data and decrypt it by traversing the tree, following the path in the reconstructed tree given by each bit. The end result should be the original output.

Data -> Histogram -> Priority Queue -> Huffman Tree -> Code Table

Histogram:

An array of 256 uint64_t's
[(frequency of a), (frequency of b), (frequency of c), … , (frequency of ascii 256)]

Priority Queue:

An array of nodes representing each item in the histogram whose frequency > 0

Min-priority queue: where lowest priority items are first in the array -> ascending order

- Lowest priority items get stored first
- Using insertion sort when pushing things onto queue in order of priority

When dequeuing, shift all elements in array left

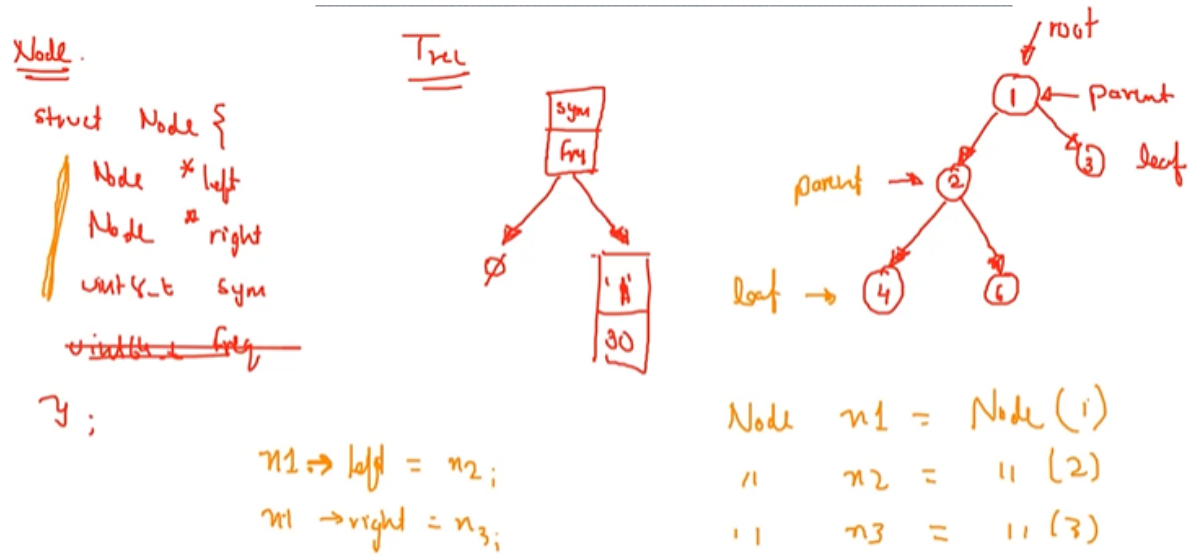
Nodes:

Print in post-order format

Parent nodes always have symbol $ and frequency of sum of its children's frequencies

Huffman Tree:

Trees are made out of a series of connected nodes



^ Credit to Harsh_Section_[05-10-21], ~22:00

Build Tree (Encode):

For length of the priority queue, we make a new node, and set its .left and .right properties to the 2 elements at the top of the priority queue (so the lowest priority elements), and set the original node's frequency to the sum of left.frequency and right.frequency. After constructing this new node, we then enqueue it into the priority queue.

Psuedocode:

```
build_tree(hist[]):
        Pq = pq_create(histogram)
        For i in histogram:
                If histogram[i] != 0:
                        // create node, passing char value of ascii and frequency
                        n = node_create(char(i), histogram[i])
        While (queue_size(pq) >= 2):
```

```
            Left = dequeue(pq)
            Right = dequeue(pq)
            Parent = node_join(left, right)
            enqueue(pq, parent)
    Return dequeue(pq) // returns root node
```

Rebuild Tree (Decode):

Reconstructs tree from the treedump array. It will iterate through tree_dump and construct a Huffman tree of nodes from the data provided.

Psuedocode:

```
rebuild_tree(nbytes, tree_dump[]):
    tree_stack
    For i in range(nbytes):
        If i == 'L':
            Leaf = node_create(next byte)
        If i == 'I':
            Right = stack_pop()
            Left = stack_pop()
            Parent = node_join(left, right)
            stack_push(parent)
    Return stack_pop()
```

Codes:

Codes are a new ADT that replicate a stack of bits. We'll use codes to keep track of our path while traversing the Huffman tree, giving us the ability to save each character code to the code table once we reach a leaf. This ADT is similar to a bitvector and a stack.

Psuedocode:

```
code_init():
    Code c
    C.top = 0
    C.bits = {0}
    Return c
Code_size(c):
    Return c.top
code_empty(c):
    Return c.top == 0
code_full(c):
    Return c->top == ALPHABET // 256 = max
code_push_bit(c, bit):
    If (code_full(c)):
```

```
            Return false
        Else if (bit == 0):
            c.bits[c.top / 8] &= ~(0x1 << (c.top % 8)); // && og byte with 11..0..11
            C.top++ //iterate c.top
        Else:
            c.bits[c.top / 8] |= (0x1 << (c.top % 8));
            C.top++ //iterate c.top
        Return true
code_pop_bit(c, bit):
        If code_empty(c):
            Return false;
        Else:
            C.top-- // decrement top
            // shift og vector right by offset, then & with 0x1 to get single bit at top
            Bit = c.bits[c.top / 8] >> (c.top % 8) & 01
            Return true
code_print(c):
        For i in range c->top:
            print(c.bits[i])
```

Code Table:

Constructing the code table is the final step in actually converting the original file into a Huffman code. The code table is another array whose indices correlate to ascii values 0-255, but this time the values at each index will represent the huffman code for that character. We get this code for each character by searching the Huffman tree for the character, and each branch we go down we will push either a 0 (if we go left) or a 1 (if we go right) onto a stack. If we reach a node with no children, we backtrack by popping from the pathfinding stack, and returning to the parent element in the tree. If the desired character is found, whatever binary code we have on the stack is the code which we'll store in that index in the code table.

Psuedocode:

```
build_codes(Node root, Code table[]):
        Code = code_init()
        // post-order traversal from root
        post_order_traversal(root, code, code_table);

post_order_traverse(Node root, code, code_table):
```

```
If (root->left):
        code_push_bit(code, 0) // add 0 to code when stepping left
        post_order_traverse(root->left)
        stack_pop_bit(code) // pop after recursing down left branch
If (root->right):
        code_push_bit(code, 1) // add 1 to code when stepping right
        post_order_traversal(root->right)
        stack_pop(stack) // pop when done recursing right branch
If (root->right == NULL && root->left == NULL):
        code_table[root->symbol] = code
```

Header:

Once we construct the code table, we must construct a header to pass all necessary information from input file to the encoded output file. This will include the magic number given as a macro, permissions, tree_size, and file size. (NOTE: see page 11 of doc for instructions on how to retrieve this data.)

Writing code to outfile:

- Write constructed header to outfile
- Print post-order traversal of tree to outfile, using L followed by byte of symbol for leaves, and I for interior nodes.
- Read through infile and for each character read, write corresponding character code to outfile using write_code()
- Flush codes
- Close files

I/O:

Read_bytes() should read bytes from buffer until nbytes is reached or there's no more bytes to read in buffer

write_bytes() should write bytes to buffer until nbytes is reached or there's no more bytes to read in buffer

read_bit() should read a byte and distill out a single bit from it and pass it to the bit pointer

write_code() should write bits from code to buffer one at a time, then outfit buffer contents to outfile when buffer is full (when BLOCK bytes are occupied)

flush_codes() should erase whatever extraneous bits end up in buffer when the encrypted data doesn't fill exactly one BLOCK of space.

Huffman Coding Module:

Psuedocode:

```
// returns root node
Node build_tree(histogram):
    // q = priority queue storing all character nodes representing their symbol and frequency
    N = queue_size(q)
    For i in range n - 1:
        x = create_node()
        Left = dequeue(q)
        Right = dequeue(q)
        //code_join()
        X.left = left
        X.right = right
        X.frequency = left.frequency + right.frequency
        enqueue(q, x) // push the new joined node onto the priority queue
    return x
```