Guillermo Alfaro
CSE13S Winter 2022

Huffman Coding

Description:

Reads file by the byte creating shorthand bit codes for each byte based on how frequently they show up. Then reconstructs file using shorthand version saving space.

Files

1. Encode.c + encode.h

   Controller of file compressor, creates histogram and encoded file.

2. Decode.c + decode.h

   Controller of file decompressor, rebuilds encoded file.

3. Node.c + node.h

   Contains all functions to interact with Nodes.

4. Pq.c + pq.h

   Contains all functions to interact with PQs.

5. Code.c + code.h

   Contains all functions to interact with Codes.

6. Io.c + io.h

   Contains all functions that act as better fopen() and fcloses().

7. Stack.c + stack.h

    Contains all functions to interact with Stacks.

8. Huffman.c + huffman.h

    Contains functions that perform huffman encoding and decoding.

9. Makefile

    Blueprint on how to assemble files.

10. README.md

    Manual on how to use exacutables.

11. Design.pdf

    This pdf.

Pseudocode:

```
Encode.c
    Main
        Arguments variables
        Get opt switch
            Cases for each argument
        While loop
            Reads input file into buffer
            Creates histogram using for loop
            Break if read size of input file
        Build_tree
        Build_codes
        Reset infile to top
        While loop
            Re-reads input file
            For every character that shows up write its code
            counterpart to outfile
            Break if read size of input file
        Flush_codes
        Close files

Decode.c
    Main
        Initialze variables
        Get opt switch
            Cases
        Checks if magic number is present
        Reads in header info
```

Reads amount of bytes from header

…

…

Node.c

Node *node_create(symbol, frequency)

Assigned symbols to node->symbol

Assigned frequency to node->frequencey

Returns pointer to struct Node.

Void node_delete(Node **n)

Frees Node

Set memory to NULL after

Node *node_join(*left, *right)

Creates new node with *left as its node->left and *right as its node->right node->symbold = $

Pq.c

Insertion

PriorityQueue *pq_create(capacity)

Malloc sizeof(uint32_t) capacity amount of times

Capcity is maximum so we dont use calloc

Void pq_delete(*q)

Frees the PQ

Set pointer to NULL after

Bool pq_empty(*q)

Checks top to see if something is present

Returns true if *q is empty false otherwise

Bool pq_Full(*q)

If top is at capacity

Returns true is PQ is full

```
Uint32_t pq_size(*1)
    Returns number of items in PQ(top)

Bool enqueue(*q, node *n)
    If pq_full(q) = true
        Return false
    Else enqueue node
        Assign node to top space
        Increments top
        Return true

Bool dequeue(*q, node **n)
    If pq_empty(*q) = true
        Return false
    Else dequeue
        Pass the highest priority node to the **n
        Decrememnts top
        Return true
    Void pq_print(*q)
        Prints q

Code.c
    Code code_init(void)
        Initialize code
        code->top = 0
        While (1)
            Iterate through bits assigning each to 0 until end of bits
            then breaks
    Uint32_t code_size(*c)
        Returns number of bits in code->bits
        Is exact number of bits pushed in

    Bool code_empty(*c)
        Returns true if empty
        Else returns false
```

```
Cool code_full(*c)

        Returns true if full

        Else returns false


Bool code_set_bit(*c, uint32_t i)

        c->bits[i] = 1

                If error then return false

        Else return true


Bool code_clr_bit(*c, uint32_t i)

        c->bits[i] = 0

                If error then return false

        Else return true


Bool code_get_bit(*c, uint32_t i)

        If (c->bits[i])

                Return true

        Else return false

        (fool proof ? ^)
```

```
Bool code_push_bit(*c, uint8_t bit)
    If code_full(c) is true
        Return false
    Else add bit to top of stack ??
        (not sure how to do this uet)
        Return true


Bool code_pop_bit(*c, uint8_t *bit)
    If code_empty(c_) is true
        Return false
    Else remove top of stack bit and assign it to *bit
        Return true
Void code_print(*c)
    Debugger
    For loop (i)
        print(c->bits[i])
```

Io.c

```
Int read_bytes(int infile, uint8_t *buf, int nbytes)
    Read infile with buf as buffer
```

Set read() to add into a variable that once qualds nbytes breaks

loop

Return this variable

Int write_bytes(int outfile, uint8_t *buf, int nbytes)

Does exact same as read_bytes except with writing

Bool read_bit(int infile, uint8_t *bit)

Performs a read() on infile into buffer var

Buffer is size of global variable BLOCK

“Dole out” bits till buffer is empty,

Use index that tells to return bit through *bit

Return false if no more bits to be read

True if there are still bits to be read

Void write_code(int outfile, *c)

Same process as read_bit

Once buffer of size BLOCK is full write buffer to outfile

Repeat

```
Void flush_codes(int outfile)

    Zero out buffer


Stack.c

Stack *stack_create(uint32_t capacity)

    Stack var malloc sizeof(uint32_t) capacity amount of times

    This creates a maximum amount

    var->capacity = capacity


Void stack_delete(stack **s)

    Frees the stack

    Sets points to NULL after


Bool stack_empty(*s)

    If s->items has something in it return false

    Else return true


Bool stack_full(*s)

    If s->items is equal to s->capacity return true
    Else return false

Uint32_t stack_size(*s)

    Loop items and iterate amount of nodes in stack

    Return this number

Bool stack_push(*s, node *n)

    Assignms s->node[s->top + 1] = n

        Return true if successful

    Return false if s->top + 1 = capacity

Bool stack_pop(*s, **n)

    If stack_empty is true return false

    Else s->nodes[s->top] = *n

    Decrements top

Void stack_print(*s)

    Iterate s->top amount of times printing each s->node[]
```

Huffman.c

```
Node *build_tree(uint63_t hist[static ALPHABET])
    Creates Huffman tree with an alphabet number of indices
    that are actually present (so no wasted allocation)
    Creates PQ and placeholder nodes
    Uses huffman tree array to create nodes for each present ASCII
    Dequeus two at a time combins then requeues till 1 left
    Returns root node of this tree



Void build_codes(*root, code table[static ALPHABET])
    If root is not null
        If no more left or rights
            Copy depth code into ASCII position in table
        Push 0 into ASCII table to indicate left traverse
        Recurse with node.left
        Pop bit


        Push 1 into ASCII table to indicate right traverse
        Recurse with node.right

        Pop bit


Void dump_tree(outfile, *root)

    If root

        Recurse with root.left

        Recurse with root.right



        If there not more child nodes

            Write that you found leaf in outfile

        Else

            Write that that there is more to recurse


Node *rebuild_tree(nbytes, tree[nbytes])

    Initialize Node and Stack of nodes

    While nbytes - 1

        If leaf

            Write node after into stack

        If I

            Pop most recent nodes and combine for parent
```

```
Void delete_tree
    Free all pointers
    Set pointers to null
```

Citations:

Citation 1:

I used this source to be able to get an exact bit. It uses bit shift operators to achieve this feat. I claim no credit of these singular lines of code I partially used.

https://stackoverflow.com/questions/47981/how-do-you-set-clear-and-toggle-a-single-bit