

Assignment 5 - Huffman Coding

Teresa Joseph

CSE 13S13S - Professor Long

Fall 2021 - RD October 28/ FD November 7

Purpose

Assignment 5 is the implementation of Huffman encoders and decoders. Encoders will be able to read a file and transform it while decoders can return a file to its original form. Nodes, priority queues, and stacks will be implemented to create these encoders and decoders. An I/O interface will be implemented as well to read and write files. The main test harness will be able to take on command line arguments h, i, o, and v for printing a help message, reading an input file, writing an output file, and compressing statistics respectively.

Pseudocode

node.c

create structure Node (*left, *right, symbol, frequency)

node_create

 allocate memory of size of Node to pointer n

 if n is not null, create n's symbol = symbol

 if n is not null, create n's frequency = frequency

 return n

node_delete

 free pointer n

 set n to null

node_join

 parent_node's symbol = \$

 parent_node's frequency = left's frequency + right's frequency

 return parent_node

node_print

 check that nodes exist from node_create

 check that node_join works

 test functions with an example node

pq.c

create structure PriorityQueue (head, tail, capacity)

pq_create

allocate memory of size of PriorityQueue to pointer q

if q is not null, q's capacity = capacity

q's head and q's tail both = null

return q

pq_delete

free pointer q

set q to null

pq_empty

if q's head is 0, then return true

else, return false

pq_full

if q's head is = to q's capacity, return true

else, return false

pq_size

return q's head

enqueue

if pq_full is true, return false

else, set pointer n to q's head

increment the head by 1

return true

dequeue

if pq_empty is true, return false

else, set pointer n to q's tail

decrement the head by 1

return true

pq_print

check that pq_create creates the queue

check that enqueueing and dequeuing work

test functions with an example queue

code.c

create structure Code (top and bits)

code_init

create new Code c on the stack

set c's top = 0

for all iters in array of bits, set iters = 0

return c

code_size

return c's top (the number of bits pushed onto Code)

code_empty

if c's top is = 0, return true

else, return false

code_full

if code's top is = to code's max length (MAX_CODE_SIZE, return true

else, return false

code_set_bit

if index i is out of range, return false

else, set c's bit at index i = 1

return true

code_clr_bit

if index i is out of range, return false

else, set c's bit at index i = 0

return true

code_get_bit

if index i is out of range or if bit at index i is 0, return false

else, return true

code_push_bit

if code_full is true, return false

set the top to the bit

increment the top by 1

```

        return true
code_pop_bit
    if code_empty is true, return false
    set pointer bit to top-1
    decrement the top by 1
    return true
code_print
    check that pushing and popping work
    check that setting and clearing bits work
    test functions with an example code

```

io.c

```

read_bytes
    while nybte is met (in comparison to buf) or there are no more bytes:
        call read() on bytes
    return number of bytes read from infile
write_bytes
    while nybte is met (in comparison to buf) or there are no more bytes:
        call write() on bytes
    return number of bytes written to outfile
read_bit
    create static variable buffer, store BLOCK number of bytes
    read block of bytes into buffer
    if no more bits can be read, return false
    if there are more bits that can be read, return true
write_code
    create static variable buffer (create outside function)
    for i representing each bit in c until BLOCK bytes filled with bits:
        write buffer contents to the outfile
flush_codes
    for any leftover buffered bits:
        call write() on bits

```

set bits in last byte to 0

stack.c

create structure Stack (top, capacity, and items)

stack_create

allocate memory of size of Stack to pointer s

if s is not null, s' capacity = capacity

if s is not null, s's top = top

if s is not null, s's items = items

return s

stack_delete

free items

free s

set s to null

stack_empty

if top is 0, return true

else, return false

stack_full

if top = capacity, return true

else, return false

stack_size

return top value

stack_push

if stack_full is true, return false

set top of stack to pointer n

increment top by 1

return true

stack_pop

if stack_empty is true, return false

set pointer n to top of stack

decrement top by 1

return true

stack_print

- check that stack_create works

- check that pushing and popping always return true

- test functions with an example stack

huffman.c

build_tree

- create priority queue

- (use min heap process from assignment 3)

- while queue's size is > 1 :

 - dequeue the left of the current node

 - dequeue the right of the current node

 - create parent node by joining the left and right nodes

 - enqueue the parent node

- return root node (the last dequeue)

build_codes

- while traversing the tree:

 - initialize starting code_value

 - if left node, push bit 0 to code_value

 - if right node, push bit 1 to code_value

 - continue until end of branch

 - if end of branch:

 - look for corresponding symbol at node

 - find symbol in code table and add final code_value

dump_tree

- if root:

 - dump left and right

- if not left and not right:

 - write 'L'

 - write the symbol of the corresponding node

- else, write 'I'

rebuild_tree

```

    while not the end of tree_dump:
        traverse tree_dump
        follow corresponding bit using build_tree
    return root node

delete_tree
    while traversing all nodes:
        set current node to 0
    free pointer root
    set pointer root to null

```

encode.c

```

parse command line arguments with getopt
read infile and construct histogram hist (initialized to ALPHABET size)
increment element #0 and element #255
use priority queue and build_tree to make tree
    iterating over hist, if frequency > 0:
        create corresponding and add to queue
        while 2+ nodes in queue, dequeue two nodes (left then right children)
        create parent node from node_join() of left and right
create code table with build_codes() (also initialized to ALPHABET size)
    use code_init() to create Code c
        if current node is a leaf, c represents path to node (save this in code table)
        else, current is an interior node (push 0 to c and recurse)
    pop bit from c and push 1 once returning from left link
write header to outfile
write tree to outfile using dump_tree()
write code of each symbol in infile to outfile with write_code()
flush remaining code with flush_codes()
close both infile and outfile

```

decode.c

```

read infile header
if magic number doesn't match 0xBEEFD00D, print error message and quit

```

```

else, continue
set outfile should be set to permissions using fchmod()
read dumped tree from infile into an array
use rebuild_tree to reconstruct tree
    array with dumped tree has length nbytes
    iterate over this array from 0 to nbytes
    if element is 'L,' then the next element will be a symbol (use it for node_create())
    if element is an 'I,' pop stack to get the right child and pop again for the left child
        join both to get the parent and push this onto the stack
    remaining element = root
read infile using read_bit()
    beginning at the root, if 0 is read, go to the left child and if 1 is read, go to right
    if leaf node encountered, write node symbol to outfile
    reset current node to root of tree and continue
    exit when number of decoded symbols = file_size
close both infile and outfile

```

Notes

- symbol and frequency will be specified by n->symbol and n->frequency
- head, tail, and capacity will be specified by pq->head, pq->tail, and pq->capacity
- top and bits will be specified by c->top and c->bits
- top, capacity, and items will be specified by s->top, s->capacity, and s->items
- Lecture 10 on stacks and queues was incorporated for setting up the priority queue
- node_print, pq_print, code_print, and stack_print are still being developed
- I'm not entirely sure what "flushing the codes" means, so I'll do more research and fix this part of my pseudocode later

Other (notes and process)

plain text \rightarrow encode it \rightarrow cipher text
(transformation)

cipher text \rightarrow decode it \rightarrow clear text

(plain & clear should match)

need

- nodes
- priority queue
 \hookrightarrow Huff. tree

* file is just a sequence of bytes

process for encoder

1) histogram \rightarrow 256 indices array, start at 0 for all
(0 to 2^8-1 bits \Rightarrow 256 values)

look at symbol frequency, increase by iterating over input

2) make Huffman tree w/ priority queue (must fill it)

\hookrightarrow in array, if non-zero entry, make min heap operation
(parent \leq child)

while pq \rightarrow size > 1

left = dequeue()

right = dequeue()

parent = join(left, right)

enqueue(parent)

}

root = dequeue

3) walk Huffman tree to construct corresponding code for each symbol

\hookrightarrow 0 to ∞ from left node to right node, top to bottom

4) dump tree: postorder(n)

if node isn't NULL:

postorder(n \rightarrow left)

postorder(n \rightarrow right)

// do something w/ n

5) walk input, ~~want~~ want to write out each symbol's code based on 3)
process for decoder

1) Reconstruct Huffman tree (iterate over tree chunk)

↳ while stack > 1

 left = pop()

 right = pop()

 parent = join(left, right)

 push(parent)

2) walk bits & traverse tree