Assignment 5 - Huffman Coding

Teresa Joseph

CSE 13S - Professor Long

Fall 2021 - RD October 28/ FD November 7

Purpose

Assignment 5 is the implementation of Huffman encoders and decoders. Encoders can read a file and compress it while decoders can return a file to its original (decompressed) form. Nodes, priority queues, and stacks are implemented to create these encoders and decoders. An I/O interface is implemented as well to read and write files. The two test harnesses encode.c and decode.c can take on command line arguments h, i, o, and v for printing a help message, reading an input file, writing an output file, and displaying compressing statistics respectively. Both should run together to see how the file size changes.

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
- if n is not null, set n's left and right children to none
- return n

node_delete

- free pointer n
- set n to null

node_join

- call node_create to make parent node
- parent_node's symbol = $
- parent_node's frequency = left's frequency + right's frequency
- return parent_node

node_print

print node symbol and frequency

if children exist, print their symbols and frequencies too

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 number of entries = 0

allocate memory of size Node for all nodes in the queue

return q

pq_delete

free pointer q

free all nodes in queue

set q to null

pq_empty

if q's number of entries is 0, then return true

else, return false

pq_full

if q's number of entries is = to q's capacity, return true

else, return false

pq_size

return q's number of entries

enqueue

if pq_full is true, return false

else, set pointer n to q's node array at index (current number of entries)

increment the number of entries by 1

call build_heap with nodes and number of entries

return true

dequeue

if pq_empty is true, return false

else, set pointer n to q's first node (root)

decrement the head by 1

set the first entry to the nodes's array at index (current number of entries)

call min_heapify with nodes, starting node, and number of entries

return true

pq_print

for each entry, print symbol and frequency

min_heapify

let left index be the starting node*2

let the right index be the left index+1

let the min index be the starting node

if right index is less than the size and if its previous index's frequency is less than the min index's previous frequency, set the min index to the right index

if left index is less than the size and if its previous index's frequency is less than the min index's previous frequency, set the min index to the left index

if the min index is not the starting node:

swap the min index's previous value & the starting node's previous value

call min_heapify again recursively with the min index

build_heap

set the parent's index to half of the total size

for each value leading up to parent index, call min_heapify on it

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 (ALPHABET), return true

else, return false

code_set_bit

if index i is out of range, return false

set byte index to be i/8

set bit index to be i%8

create a mask with 1 logical shift left of the bit index

OR the byte index in the bits array with the mask

return true

code_clr_bit

if index i is out of range, return false

set byte index to be i/8

set bit index to be i%8

create a mask with 1 logical shift left of the bit index

AND the byte index in the bits array with the mask

return true

code_get_bit

if index i is out of range, return false

set byte index to be i/8

set bit index to be i%8

create a mask with 1 logical shift left of the bit index

if AND of byte index in the bits array with the mask is true, return true

else, return false

code_push_bit

if code_full is true, return false

set byte index to be top/8

set bit index to be top%8

create a mask with 1 logical shift left of the bit index

if the bit is 0, AND byte index in the bits array with the mask

if the bit is 1, OR byte index in the bits array with the mask

increment the top by 1

return true

code_pop_bit

if code_empty is true, return false

decrement the top by 1

set byte index to be top/8

set bit index to be top%8

create a mask with 1 logical shift left of the bit index

if & byte index in the bits array with the mask isn't 0, set bit to 1

else, set bit to 0

return true

code_print

print each bit for a given code_size of c

io.c

read_bytes

let current bytes read be 0

let bytes read be 0

while bytes read is less than the given value:

call read() on current bytes read

when no bytes remain, break from loop

increment the bytes read by the current bytes read

return bytes read from infile

write_bytes

let current bytes written be 0

let bytes written be 0

while bytes written is less than the given value:

call write() on current bytes written

when no bytes remain, break from loop

increment the bytes written by the current bytes written

return bytes written from infile

read_bit

create static variable buffer, store BLOCK number of bytes and set all to 0

set index count to 0

set the end to -1

at the start, when index = 0:

set bits read to read_bytes()

if bits read < BLOCK, set end to bits_read*8 + 1 (to get next one)

if bits red < or = to 0, return false

set byte index to be top/8

set bit index to be top%8

set temp to buffer array of byte index

create a mask with 1 logical shift left of the bit index

if temp AND mask is not 0, set the bit to 1 and else, set the bit to 0

increment the index

if index is total number of bits, set index to 0

if index is at the end, return false

return true

write_code

create static variable buffer (create outside function)

create static index at 0 (outside function)

for i representing each bit in c until BLOCK bytes filled with bits:

call code_get_bit()

set byte index to be top/8

set bit index to be top%8

create a mask with 1 logical shift left of the bit index

AND the buffer array at index byte index with the inverse of the mask

at bit = 1, OR buffer array at index byte index with mask

increment the index

at index = total number of bits, call write_bytes and set index = 0 to restart

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 Node 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

for every i until the top, print the value at items[i]

huffman.c

build_tree

create priority queue

for every item in histogram, create node with item and histogram[i] and enqueue

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

if the size of the queue is 1, queue the remaining node (the root)

delete the queue to free memory

return root

building_codes (for build_code)

if the root is null, return

if both of the children are null:

set the table array at the root's symbol's index to pointer c

increase the count of the recursion and decrease the count of the depth

return

call code_push_bit and push 0 to c

call recursively with root's left

call code_pop_bit to remove seen bit

call code_push_bit and push 1 to c

call recursively with root's right

call code_pop_bit to remove seen bit

decrement depth by 1

build_codes

initialize Code c

call building_codes above (for recursive call)

dump_tree

if root is null, return

recursively call outfile with root's left

recursively call outfile with root's right

if not left and not right:

write 'L' with write_bytes

write the symbol of the corresponding node with write_bytes

else, write 'I' with write_bytes

rebuild_tree

create stack

for i from 0 to nbytes:

if 'L' encountered, create node and push it onto stack

else, if 'I' encountered:

create left and right nodes

pop right node, then pop left node

create parent node with node_join of left and right

push parent onto stack

create root node

pop root from stack and delete the stack

return root node

delete_tree

if root pointer is null, return

call recursively to delete root's left

call recursively to delete root's right

call node_delete to free root

set root pointer to null

encode.c

main function:

create global histogram of ALPHABET size and global table of ALPHABET size

identify stat for file permission (calling it fileStat)

create header variable of Header type

```
    parse command line arguments with getopt
        if h, print help message
        if v, set v_flag to true
        if i, open with optarg and readonly
        if o, open with optarg and readonly + create
            here, also set file permissions for both infile and outfile
    create histogram with create_hist
    set Node root to build_tree of histogram
    call build_codes with histogram and code table
    open infile to read again with lseek
    create buffer and allocate BLOCK enough memory
    set header data
        header.magic is MAGIC
        head.permissions = fileStat.st_mode
        header.tree_size = 3 * call to get_unique_count() - 1
        header.file_size = fileStat.st_size
    write bytes to outfile with header and called to write_bytes()
    dump tree with dump_tree()
    until the end of infile:
        set return length to read()
        if length is less than 1, break
        else, loop from 0 to length and call write_code with code table
    flush remaining codes with flush_codes()
    set outfile with fstat() as described in assignment document
    store the size of the compressed file with fileStat.st_size
    here, check the v_flag:
        calculate the statistic and print data of sizes
    free all memory
    close both files
create_hist
    allocate enough memory of BLOCK to a buffer
```

until size of ALPHABET, set all histogram values to 0

set the first and last indices of the histogram to 1

while true:

set return length to read()

if length is less than 1, break

else, loop from 0 to length and call write_code with code table

free the buffer

get_unique_count (of histogram)

set count to 0

for the length of the histogram:

if the value at the current index is 0, continue

else, increment the count

return the count

decode.c

identify stat for file permission (calling it fileStat)

create header variable of Header type

parse command line arguments with getopt

if h, print help message

if v, set v_flag to true

if i, open with optarg and readonly

if o, open with optarg and readonly + create

here, also set file permissions for both infile and outfile

call read_bytes with infile and header

allocate enough memory to buffer of tree_size

call read_bytes again but with buffer instead

create Node root of tree

set root to rebuild_tree of header.tree_size

free the buffer

for i from header.file_size:

if decode of the root is false, break

set outfile info with fileStat.st_size

check the v_flag:

calculate the statistic and print data of sizes

free memory by deleting root and tree

decode()

if root is null, return false

if root's left and root's right is null, write root symbol to outfile and return true

if read_bit of infile and bit is false, also return false

if bit is 0, recursively call with root's left

if bit is 1, recursively call with root's right

return false at the end

Notes

- symbol and frequency will be specified by n->symbol and n->frequency
- number of entries and capacity will be specified by pq->num_entries 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

Overall Description

The files for node, pq, code, io, stack, and huffman have multiple functions implemented for their respective purposes. All of them have create(), the ones that have memory allocation have delete(), all of them have print() for debugging purposes, etc.

node.c uses malloc to allocate memory for the Node pointer n, deletes it to free memory, and adds the frequency of its children and uses the symbol $ to make a parent node. It is quite straightforward and easy to follow along. Most of it resembles the overall set up of assignment four files.

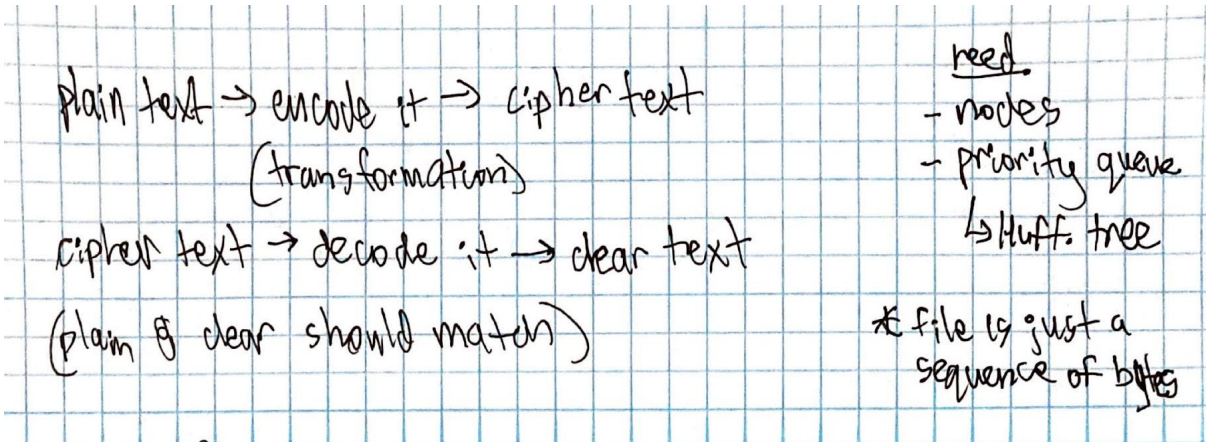
pq and stack are alike in the sense that they both enqueue/push and dequeue/pop and behave similarly in general. pq.c specifically creates a queue with malloc, deletes it to free memory, checks if it is empty or full, and returns its size. stack.c does all of this as well. In pq, enqueue and dequeue make use of a minimum heap implementation to sort its nodes by priority. This ensures that the nodes with the highest priority appear first in the queue. This is important for implementation in the encoder. Stack makes use of push and pop, and is very similar to the stack implementation in assignment four, except for the fact that it uses nodes instead of items.

code.c is used for manipulating codes. It can initialize codes, return the side of a code, check if it is empty or full, set bits, clear bits, get bit values, and push or pop bits. Some of these functions resembled those of pq and stack. In addition, this file makes use of bit vector ideas. Logical AND, OR, and left shifts were used heavily here to get bytes and bits and consequently alter the values of the array.

io.o exists to read/write bytes/bits. More specifically, it reads and writes bytes with calls to read() and write(), keeping track of the number of bytes read or written from the infile or outfile respectively. It can also read bits, write codes, and flush codes. Reading bits uses logical ideas from code.c to access bits. Writing and flushing codes go together, as they ensure all bits are written to the outfile, including any remaining ones that write_code might not have gotten.

huffman.c contains substantial functions, namely for building the tree, building codes, dumping the tree, rebuilding the tree, and deleting the tree. These are used in both encode.c and decode.c heavily. Building the tree requires plenty of dequeuing of children and enqueuing of parents into a priority queue. It returns the root to be used in other functions. Building code relies on recursive calls relying heavily on code_push_bit and code_pop_bit to assemble codes. Dumping the tree involves writing L to nodes without children and I to interior nodes (with children) to be used when decoding. Finally, deleting the tree involves freeing the memory of all of the nodes of the tree.

Other (notes and process)



process for encoder

1) histogram → 256 indices array, start at 0 for all
   (0 to 2^8-1 bits ⇒ 256 values)
   look at symbol frequency, increase by iterating over input
2) make Huffman tree w/ priority queue (must fill it)
   ↳ in array, if non-zero entry, make min heap operation (parent ≤ child)

```
while pq->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
   ↳ 0 to go from left node to right node, top to bottom
4) dump tree: postorder(n)

```
if node is not NULL:
    postorder(n->left)
    postorder(n->right)
    // do something w/ n
```

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

process for decoder

1) Reconstruct Huffman tree, (iterate over tree dump)
   ↳ while stack > 1

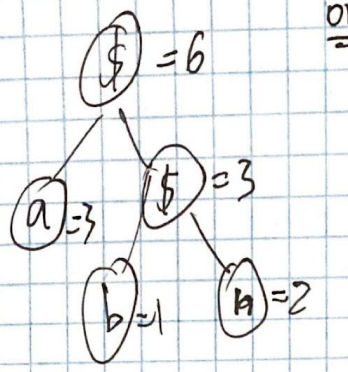
```
left = pop()
right = pop()
parent = join(left, right)
push(parent)
```

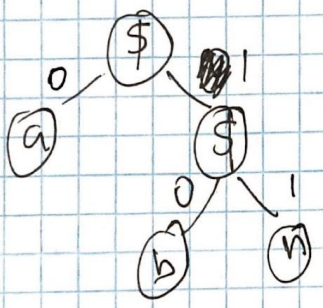
2) walk bits & traverse tree

banana

b, a, n

1, 3, 2





a = 0

b = 10

n = 11

6 + 1 carrier term = 7

↳ uncompressed (encode) & decompressed (decode)

compressed = 7 + size of header?

(35 in repo executable)

for v:

decode 100(1 − comp/decomp)

encode 100(1 − comp/uncomp)