Ruchit Patel

CSE13S, Spring 2021

Prof. Darrell Long

9 May 2021

Assignment 6: Huffman Codes

Brief Description

- In this assignment, a compression scheme--Huffman compression--is implemented.
- The lab has two main components: encoding and decoding.
- Arguments:
    - Encoder and Decoder:

        -h (prints the help message),

        -i (specifies the input file),

        -o (specifies output file).

        -v (prints compression or decompression statistics to stderr)

Descriptions/Implementation

Argument Parsing

- Arguments are parsed using getopt GNU utility.
- If -i argument is encountered, the file name pointed by optarg is opened. Error handling is also done there (see Error Handling for more details). The handling is similar for -o flag.
- If -v is encountered, inside the case for -v, an integer value is set (no Set since only one argument is set for later).

Pseudocode:

```
define flags
while (getopt returns valid)
        switch to the argument case if valid
                function according to the argument
```

if invalid print usage message and return

Node

- The Node abstract data structure (ADT) is used to represent each leaf of the Huffman Tree.

- Structure/Members of a Node
    a. Pointer to left node : Points to the left child.
    b. Pointer to right node : Points to the right child.
    c. Integer Symbol : Symbol of the node.
    d. Integer Frequency : Frequency of the symbol.

- Methods associated with a Node (Pseudocode)
    a. Node pointer node_create(integer symbol, integer frequency)

    allocate memory for Node structure

    set the left and right node pointer to null

    set symbol to *symbol*, frequency to *frequency*

    b. node_delete (pointer to Node pointer)

    if Node is not empty/null

        free the memory for Node structure

        set Node pointer to null

    c. Node pointer node_join (left node pointer, right node pointer)

    create a new node with *symbol* = '$', *frequency* = sum of left and right node frequencies

    set the left and right pointers members to to *left* and *right* node pointers respectively

    return the pointer to the newly created node

    d. node_print (Node pointer)

    print all the members of Node structure

Priority Queues (PQs)

- Priority Queues are used to actually construct the huffman tree consisting of nodes.
- Nodes are stored in an array (treated like a queue) and sorted with insertion sort.

- Structure/Members of a Priority Queue
    a. Integer head : Index to the head of the PQ.
    b. Integer tail : Index to the tail of the PQ.
    c. Integer Capacity : Total number of rows for the matrix.
    d. Integer Size : Total number of cols for the matrix.
    e. An Array : An array/queue of Node pointers.

- Methods associated with a Priority Queue (Pseudocode)
    a. PQ pointer pq_create (integer capacity)

    assign memory for PQ structure

    set head, tail, and size to zero. set capacity to *capacity*

    allocate memory (*capacity* * size of the Node structure) for the array of size *capacity*

    return pointer to newly created PQ

    b. pq_delete (pointer to PQ pointer)

    if PQ is not empty/null

        for each member in the PQ

            free the node using node_delete method

        free the PQ structure

        set PQ pointer to null

    c. boolean pq_empty (PQ pointer)

    return whether the *size* member of the PQ is equal to zero

    d. boolean pq_full (PQ pointer)

    return whether the *size* member of the PQ is equal to the *capacity* member of the PQ

e. integer pq_size (PQ pointer)

return the *size* member of the PQ

f. boolean enqueue (PQ pointer, Node pointer)

if PQ is not already full (otherwise return false)

add the Node at the tail

increment the *size* of the PQ

increment the *tail* index of the PQ (wrap around if possible)

run insertion sort (based on lab3 shell sort on the PQ array to sort in in ascending order)

[note: element here while comparing refers to the Node frequency and while swapping refers to the actual node]

for every element starting from tail till *size* elements (wrap around):

compare element with the previous element (can wrap around)

if the current element is less than the previous element then

move the element one over (wrap around if necessary)

keep comparing until a smaller element is found or array index = 0

swap the current element with the resulting index

return true

g. boolean dequeue (PQ pointer, pointer to Node pointer)

if the queue is not empty (otherwise return false)

copy the Node pointed by PQ's *head* member to the Node pointed by *Node pointer*

free the memory using node_delete

increment the head pointer (wrap around if possible)

decrement the size of the PQ

return true

h. pq_print (PQ pointer)

beginning at the head index of PQ, loop until tail (wrap around if req'd)

print each node using node_print function

Code

- The Code ADT is used to store the actual "codes" related to each symbol in the Huffman tree.

- Structure/Members of a Code
    a. Integer top : Index of the top/next bit position.
    b. An Array : An array of integers (8) to store bits of Code.

- Methods associated with a Code (Pseudocode)
    a. Code code_create ()

    create a Code structure

    initialize its *top* to zero

    initialize all values to zero

    return the newly created Code struct

    b. Integer code_size (Code pointer)

    return the *top* value of the Code structure

    c. Boolean code_empty (Code pointer)

    return whether the *top* value of the Code structure is zero

    d. Boolean code_full (Code pointer)

    return whether the *top* value of the Code structure is equal to max number of bits possible for the *bit* array of Code (namely 256)

    e. Boolean code_push_bit(Code pointer, Integer Bit)

    if the Code array is not full already (return false otherwise)

        locate the index of byte where top^{th} bit resides (top / 8)

        locate the bit inside that byte (top % 8)

        if the bit argument is one

set the bit using bitwise operation (byte ORed with 1 left shifted by bit location)

else

clear the bit using bitwise operation (byte ANDed with NOT of 1 left shifted by bit location)

increment *top* value of Code

return true

f. Boolean code_pop_bit(Code pointer, Integer pointer)

if the Code array is not empty already (return false otherwise)

locate the index of byte where *top*th bit resides (top / 8)

locate the bit inside that byte (top % 8)

if the bit argument is one

set the integer pointed by *Integer pointer* to one

else

set the integer pointed by *Integer pointer* to zero

return true

g. code_print(Code pointer)

loop from *top* index and each time

decrement loop counter

get the bit from Code's array at loop counter's position

print one if the bit is one, else zero

I/O

Functions:

a. Integer read_bytes(input file descriptor, buffer buf, number of bytes)

variable to count total number of bytes read so far, index into buffer

while bytes read so far are less than *number of bytes* <u>*or*</u> not end-of-file

call read syscall to read a block of data

increment number of bytes read so far with the return value

store the bytes in buffer at position *index into buffer*

increment the index into buffer variable

return total number of bytes read

b. Integer write_bytes(input file descriptor, buffer buf, number of bytes)

variable to count total number of bytes written so far, index into buffer

while bytes written so far are less than *number of bytes* *or* not error due to write

call write syscall to write a block of data from buffer at position *index into buffer*

increment number of bytes read so far with the return value

increment the index into buffer variable

return total number of bytes written

c. Boolean read_bit(input file descriptor, integer pointer bit)

if the current buffer is empty

read in a *BLOCK* of data into *buf*

pass the bit at buffer index through *integer pointer bit*

increment buffer index

if all the bytes have been read, reset the buffer index (buffer empty)

return true if buffer index != 0 and end-of-file has not been reached (false otherwise)

d. write_code(outfile file descriptor, Code pointer)

reset the buffer index to zero (just for safety)

starting from the *top* of the code to zero, read each bit of the code

write the bit to buffer (w/ the same previous bitwise operation)

increment the buffer index

keep doing the above steps until buffer is full (index == *BLOCK* * 8)

once the block is full write the block to the file and reset the buffer index to zero

if there are still bits in the buffer (index != zero), flush the buffer to the file using flush_codes

e. flush_codes(outfile file descriptor)

if there are still bits in the buffer (index != zero)

write, after, zeroing out, *n* bytes from buffer to file where *n* = n/8 + 1 (ceil)

Stack

- A Stack ADT is used by the decoder to store Nodes and reconstruct the Huffman tree.

- Structure/Members of a Stack
    a. Integer top : Index of the top/next stack element.
    b. Integer capacity : An array of integers (8) to store bits.
    c. Node array pointer : Points to an array of Node pointers.

- Methods associated with a Code (Pseudocode)
    a. Stack pointer stack_create(integer capacity)

        allocate memory for stack structure

        initialize top element to 0, capacity to *capacity*

        allocate memory for *items* array of size of the Node pointer times capacity

        return pointer to newly created stack

    b. stack_delete(pointer to stack pointer)

        free each node in the node array using node_delete function

        free the memory for stack structure

        set the stack pointer to null

    c. boolean stack_empty(stack pointer)

        return whether the *top* element of the stack structure has value zero

    d. boolean stack_full(stack pointer)

        return whether the *top* element has the same value as the *capacity* element

    e. integer stack_size(stack pointer)

        return the *top* element of the stack structure

    f. boolean stack_push(Stack pointer, Node pointer)

        if the stack is not full already (return false otherwise)

            store the *node pointer* in the *Node* array at the *top* index

            increment the *top* value

return true

g. boolean stack_pop(Stack pointer, Pointer to Node pointer)

decrement the *top* value

store the node pointer value in *Node* array at *top* index in *pointer to Node pointer*

return true

h. stack_print(stack pointer)

loop starting at the *top* index

decrement the loop counter

print node in the array at the index loop counter

Huffman Module

Methods: [based on lab doc description]

a. Node pointer build_tree(integer histogram array)

create a priority queue

for each element in the array (> 0)

create a node and insert it into the PQ

while there are two or more elements in the PQ

dequeue two elements and join them (first dequeued element is the left one)

enqueue the joined node

return the remaining node in the PQ (the root node)

b. build_codes(root Node, Code table)

if current node is the leaf node

push the code to the code table

recurse starting at the root node

make a new code

push bit 0 to the code (to indicate left path)

recurse from the left child

pop the bit from the code

push bit 1 to the code (to indicate right path)

recurse from the right child

pop the bit from the code

c. Node pointer rebuild_tree(integer nbytes, tree_dump array)

create a stack of size *nbytes*

while the stack size is > 1

if the symbol is *L* then create a node out of following symbol

push the created node onto the stack

if the symbol is *I* then pop two nodes from the stack (first one is the right child)

join the nodes and push them onto stack

return the remaining node (the root node)

d. delete_tree(pointer to root Node pointer)

recurse starting at the root node

recurse from the left child

delete the left child using node_delete (or current depending on context)

recurse from the right child

delete the right child using node_delete (or current depending on context)

delete the root node and set it to *NULL*

delete the left child (or current depending on context)

Error Handling

An error is returned if: (done throughout the program)

- Input or Output file cannot be opened.
- Permissions to file could not be set.
- Memory cannot be allocated.

High-level Encoder Program Flow (error checking is done throughout the program)

[Based on the lab doc description]

Set default files to stdin and stdout

Parse arguments and open files

Change output file permissions

Create a histogram of a file

Construct a priority queue

Construct a code table

Construct the header structure

Write the post-order traversal of Huffman tree

Write the code for each input file character to output file (lookup in code table)

Free memory and close files

High-level Decoder Program Flow (error checking is done throughout the program)

[Based on the lab doc description]

Set default files to stdin and stdout

Parse arguments and open files

Read in the header structure and check the *MAGIC* number

Change output file permissions

Create a tree_dump array and read the tree dump in it

Rebuild the huffman tree

Walk through the Huffman tree depending on the input file bits and write the symbol to output file

Free memory and close files