Alex Clanton

11/5/2021

Prof.Long

CSE13S

Pseudocode/DESIGN.pdf Final

Rundown:

Make a Huffman encoder, Huffman encoding is a form of lossless compression to reduce file size. The code takes a file that has bytes stored and inputs the file to the encoder. The code should compute the occurrences of each symbol in the given file. Construct a Huffman tree using a priority queue, using the tree convert to a coded table that makes the occurrence into a stack of bits(1's 0's). Take the table and make a file that has the encoded data. The code also must be able to decode that file that you just encoded. A Huffman decoder takes a file input, reads the output file, and uses nodes to reconstruct the tree node by node. Re-output the decoded file.

Files:

Encode.c-encoder

Decode.c-decoder

node.c-Node ADT

Pq.c-priority queue ADT

Code.c- code ADT

Io.c- I/O using low level calls

Stack.c-stack ADT

Huffman.c huffman coding module

Outlined structure:

Encode.c:

Take user input for file, using getopt() and getting their inputted file and their desired output file

Use fstat() and fchmod() with a struct stat that's from #include <sys/stat.h>

Save the file size and security to the header

Use read_bytes() go through BLOCK sized data and for each occurrence make a histogram(array)

Construct histogram and give values of each symbol, increment by extra 1

Go through the histogram which is size ALPHABET, and find the number of symbols

Call build_tree() pass it the histogram

Call build_codes() pass it a Code struct that you declared and the Node struct you got from build_tree()

Set the values of file stats, such as MAGIC, permissions,size, and tree size

Call read_bytes() and go through sized BLOCK, and through total read data, use write_code which writes the bits one at a time

Call flush_code() when their extra bits are in a buffer to write it

Check if they wanted compression stats, print out the values from io.c

Close and free

Decoder.c

Read the get opt commands for -h and -i for input and -o for output

Read the header from infile, and make sure the magic number and file information is ok

Use fstat() and read_bytes for the size of the header

If header's magic is the same then were ok

Make a uint8_t tree giving it the size of the tree from the header

Read the data from the dumped tree using read_bytes, and call rebuild_tree() giving it tree size and the array you just made

Go through the read data and if it’s less than the original file size, check if you have read a block

    If you did, then reset the index

If the value read is a 1, assign it to the right Huffman tree

If the value is a 0, assign it to the left Huffman tree

If both are NULL, then increase mem_index and set the symbol $, and swap the root as you reached the end

Check if they wanted compression stats, print out the values from io.c

Close and free

Node.c

node_create(uint8_t symbol, uint64_t)

    Malloc the size of a Node struct

    If it exists then assign a given frequency to the struct, the symbol $, and left and right are 0

    Return a node

node_delete(Node **n)

    If a node pointer exists, then free it and set it to NULL

node_join(Node *left, Node *right)

    Create a node passing it $, and given left and right frequencies

    Return node

Node_print

Print the symbol and left and right values

Priorityque.c

Make a struct of **node struct, and a capacity int and number of items int

PriorityQueue *pq_create(uint32_t capacity)

Call malloc and set the size to a struct of pq

Priority queue gets the capacity, and number of items starts off as 0

Use calloc for the nodes and pass expected capacity

Pq_delete

If a pointer to a priority node exists, then free it and free the priority queue itself

Set the pointers to NULL

Pq_empty

If q->num is 0 return true, else false

Pq_full

If the capacity is equal to the number of items

Pq_size

Return pq num

Enqueue

If the number of items is 0, then set to 0th index to the node

Increment the q num

If the pq is not full

```
Then current number at an array of nodes gets the new node, increase the number by 1
For loop going through the parents which are index/2 -1, call heap_down
Return true
```

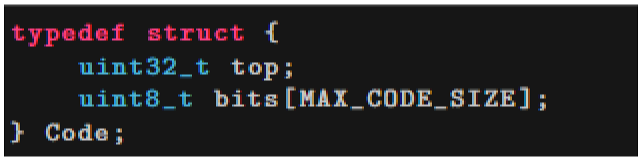
Dequeue

```
If the queue is not empty
Then value at 0th index of the queue is passed back through the node pointer
Set the 0th node to the number-1
Decrement the number by 1
Go through the parents, which are index/2 -1, and call heap_down
```

Heap_down

```
Left and right kids are assigned index/2 +1, and index/2 +2
The smallest value gets the index
If the number of items is 1, we can’t do anything
If the left and right kids are less than the number of items
    And the left and right kids frequency is less than the smallest(index value)
    Then smallest value gets the left and right kid
If the smallest value is not the index
    Then swap the smallest and the index value
```

Code.c



```
typedef struct {
    uint32_t top;
    uint8_t bits[MAX_CODE_SIZE];
} Code;
```

Code code_init(void)

Code c struct gets value of 0, and 0 for the ints

uint32_t code_size(Code *c)

Return top value

bool code_empty(Code *c)

If code_size is 0 return true

bool code_full(Code *c)

If stack is equal to 256

bool code_set_bit(Code *c, uint32_t i)

Check if I is in the range of 0 to 256

c->bits value is byte/8 and |= to mask of 1 with index mod 8

Return true if done otherwise false

bool code_clr_bit(Code *c, uint32_t i)

Check if I is in the range of 0 to 256

c->bits value is byte/8 and &= to mask of 1 with index mod 8

bool code_get_bit(Code *c, uint32_t i)

Check if I is in the range of 0 to 256

c->bits value is byte/8 and >> to mask of 1 with &1

Return true if done

bool code_push_bit(Code *c, uint8_t bit)

If the code is full, return false

```
If the bit is 1, call code_set_bit
If the bit is 0, call code_clr_bit
increment the top
Return true if done
```

bool code_pop_bit(Code *c, uint8_t *bit)

```
If the code is empty
    Return false
Otherwise use code_get_bit and decrement the top
    c->bits value is byte/8 and >> to mask of 1 with &1
```

void code_print(Code *c)

Io.c

read_bytes()

```
    While the total read data is not equal to the number of bytes were supposed to read
        Then call read() and pass the buffer and bytes to read is subtracted from the bytes read
    If read returns a -1 we have an error, exit()
    If read returns a 0 were done at EOF
    Add the bytes read to the global value of bytes_read and read_tot
    Return total read bytes
```

write_bytes()

```
    While the total of bytes written is not equal to the number of bytes
```

```
        Call write() passing the buffer and number of bytes- a total of written bytes
    If write() returns -1, exit()
    If write returns 0 then we are at the EOF
    Add the total bytes written to global values of bytes_written and write_tot
    Return the total of written bytes
```

Flush_codes

```
Find bit index, by doing index/8. Bit_left is 0x1<< index mod 8 subtract 1
    Write to the global write buffer the idx incrementing by 1 & of made mask above
    Write this buffer to the outfp
```

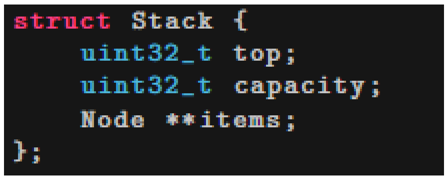
read_bit()

```
    Make a local static array bits and if the global index is equal to 0
    Call read_bytes passing the array and multiple by 8 for the total byte that is read
    Pass the bit back through *bit using get_bit equation
    Increase global index by 1 and mod by 8
    If the index is less than total read_bytes then return true
```

write_code()

```
    Go through the size of code c
        If the get bit equation is a 1 then global buffer gets set_bit equation
        If the get bit equation is a 0 then global buffer gets clr_bit equation
    Increment global index
    If the global index is the size of the block
        Then write the global buffer and reset the index
```

Stack.c



```
struct Stack {
    uint32_t top;
    uint32_t capacity;
    Node **items;
};
```

Stack *stack_create(uint32_t capacity)

Dynamic allocate the static size struct to capacity, set-top to 0, and allocate the nodes

double-pointer to the size of capacity

void stack_delete(Stack **s)

Check if the stack has nodes and exists then free those nodes and the stack itself

bool stack_empty(Stack *s)

If the top of the stack is 0 return true

Otherwise, return false

bool stack_full(Stack *s)

If the top of the stack is equal to capacity return true

Otherwise, return false

uint32_t stack_size(Stack *s)

Return stack capacity

bool stack_push(Stack *s, Node *n)

Check if the stack is full otherwise add the node to an array of nodes and move the top by 1

bool stack_pop(Stack *s, Node **n)

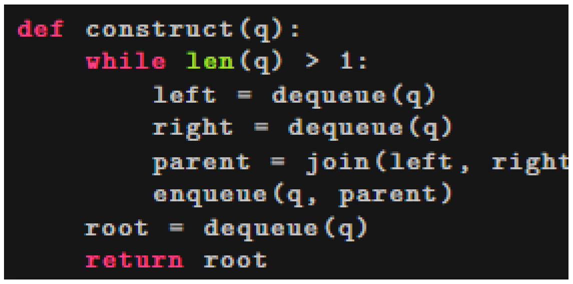
Check if the stack is empty otherwise remove a node from an array of nodes and move the top-down by 1

void stack_print(Stack *s)

Go through the top to bottom of the array of nodes and print their values

Huffman.c

Node *build_tree(uint64_t hist[static ALPHABET])



```
def construct(q):
    while len(q) > 1:
        left = dequeue(q)
        right = dequeue(q)
        parent = join(left, right
        enqueue(q, parent)
    root = dequeue(q)
    return root
```

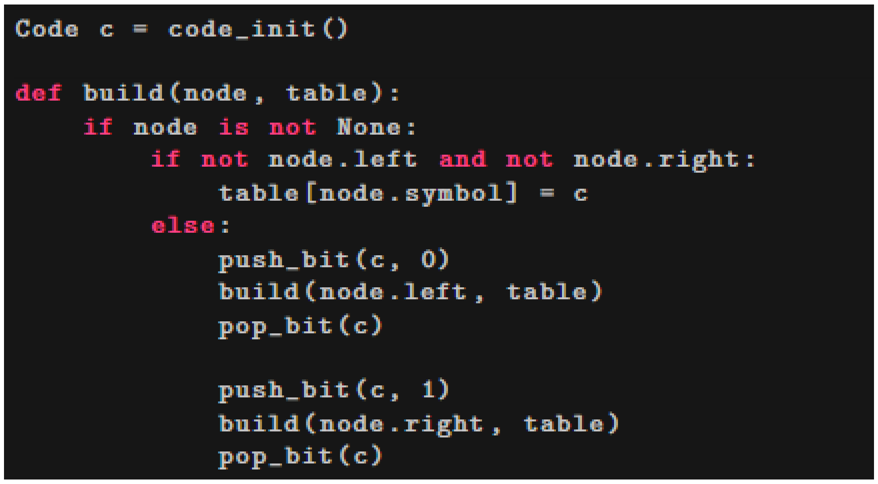
Create a Node node, and 2 children, and a join node with Nodes

Create a priority queue using pq_create()

```
While the track is less than 256
        If the histogram is not 0
        Then call node_create and pass the track and the value at the histogram
        Increment track
While priority queue size is greater than 1
        Call dequeue on kid 1 and kid 2
        Node_join the two kids
        And enqueue the kids
Dequeue the parent node
```

And call pq_delete()

void build_codes(Node *root, Code table[static ALPHABET])



```
Code c = code_init()

def build(node, table):
    if node is not None:
        if not node.left and not node.right:
            table[node.symbol] = c
        else:
            push_bit(c, 0)
            build(node.left, table)
            pop_bit(c)

            push_bit(c, 1)
            build(node.right, table)
            pop_bit(c)
```

If the root exists

If the right and left root are 0

Then c table gets the symbol at the value of the code

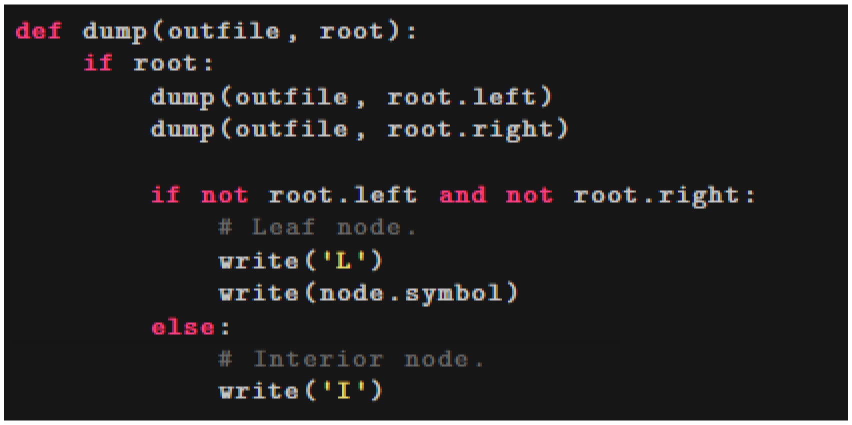
Otherwise push 0, call build_codes recursively passing 0 and root at the left

Call code_pop_bit, pass the code and bits

Call code push_bit pass codes and 1

Call build_codes, pass the root left and code table

Call code_pop_bit and pop code and the bit



```python
def dump(outfile, root):
    if root:
        dump(outfile, root.left)
        dump(outfile, root.right)

        if not root.left and not root.right:
            # Leaf node.
            write('L')
            write(node.symbol)
        else:
            # Interior node.
            write('I')
```

dump_tree()

```
Create variables to hold ‘I’ and ‘L’
If the root exists
    Then recursively call dump_tree passing the outfile and left, and below recursively call with root right
If the roots left and right are 0
    Then call write_bytes() pass 1 to write 1
    Call write_bytes() pass the symbol of the root and 1
Otherwise
    Call write_bytes() and pass the ‘I’ symbol cuz we hit an interior node, passing value of 1
```

Node *rebuild_tree(uint16_t nbytes, uint8_t tree_dump[static nbytes])

```
    Create a stack using stack_create()
    Create a Node node and 2 kid holders
    Go through the total number of bytes
        The array of dump tree at x gets ‘L’ and value 0
        Increment the index
        Create a node using node_create()
        Push the node using stack_push()
    Otherwise
        Use stack_pop() for both kids
        And push the joined nodes of the kids
    Pop the nodes
    Call stack_delete()
    Return node
```

void delete_tree(Node **root)

```
If  the root exists
    Call itself and delete the right and left node pointers
    Then delete the node itself
    Set the pointer to NULL
```