Liam Murray

Professor Miller

CSE13s

November 25, 2022

Assignment 7: A Huffman Tree Grows in Santa Cruz

General Idea:

The general idea of the assignment is to create a compression and decompression algorithm, to compress data and save space. The implementation of this algorithm will use Huffman coding, in which a histogram of the frequency of each type of character in the file will be generated, a Huffman binary tree will be generated based on that data, and a binary code is assigned to each character based on the frequency with which they occur. That data will then be sent to the receiver which will then decode the data using the same tree information as the encoder.

The encoder and decoder portions of the Huffman tree will use a priority queue, along with the node data type. These two ADTs will be primarily used to implement the sorted tree, which allows for the codes to be constructed. The code constructor will use a stack to track the path through the tree, which directly translates into a binary code. The stack data type will also be present in the decoder, as a stack is used to reconstruct the Huffman Tree.

Pseudocode:

Node: The node is used to store data points in a way that is accessible. It’s implementation is very versatile, allowing it to be used in the priority queue as well as in the stack and tree implementations.

Implementation:

Structure definition, taken from the assignment document.

Node:

Node * left //pointer to left child

Node * right //pointer to right child

Uint8_t symbol //Nodes symbol

Uint64_t frequency //how often symbol appears

//node creator

node_create(symbol, frequency)

Node pointer nd = allocate(size of (Node));

nd->symbol = symbol

nd-> frequency = frequency

Return nd

//node delete

node_delete(**nd)

free(*nd)

Nd = NULL

//node join: joins two nodes and generates a parent node

Node * node_join(*left, *right);

Sum = left->frequency + right-> frequency

Node* parent = node_create($, sum)

parent ->left = left

parent->right = right

Return parent

//node print

node_print(Node *nd)

print(Node symbol, node frequency)

print(Node left child)

print(Node right child)

Priority Queue. This implementation of a priority queue will mimic a linked list, as the nodes with a lower frequency can be added to the front of the linked list easily. For this reason, much of the code seen here is taken from my assignment 6 code. The code includes auxiliary functions used to make coding the linked list much simpler.

Pq structure definition:

Struct PriorityQueue

Int capacity = 0;

int elements = 0;

Node head;

Node tail;

pq_create(capacity)

PriorityQueue * pq = allocate(sizeof(PriorityQueue))

pq->capacity = capacity

Head = node_create(NULL)

Tail = node_create(NULL)

Return pq

pq_delete(**pq)

Start = *pq->head

```
        for(i = *pq->elements, while i != 0, i - 1 each pass)
            Next = start->right
            node_delete(start)
            Start = next
        *list = NULL
    Return
//pq empty. Returns true if pq is empty, false if otherwise
pq_empty(pq)
    If pq->elements == 0
        Return true
    Else
        Return false
//pq full. Returns true if pq is full, false if otherwise
pq_full(pq)
    If pq->elements == pq->capacity
        Return true
    Else
        Return false
//pq size. Returns pq size
pq_size(pq)
    Return pq->elements
//enqueue. Puts a node in the queue, and return true if successful
pq_enqueue(pq, Node)
    If pq_full == true
        Return false
    Start = pq->head
    for(i = *pq->elements, while i != 0, i - 1 each pass)
            Next = start->right
            If ((start->frequency) < (node->frequency) < (next->frequency))
                start->right = node
                next->left = node
```

```
                    node->left = start
                    node->right = next
                    Elements += 1
                    Return true
                Start = next
//dequeue. Removes an element from the list and processes it
pq_dequeue(pq, **Node)
    If pq_empty(pq) == true
        Return false
    *Node = head->right
    head->right = head->right->right //make the head point to the node after the removed
node
    head->right->right->left = head //make the node after the removed node point back to the
head
    Elements -= 1
    Return true
//pq print. Prints the priority queue
pq_print(pq)
    Start = *pq->head
    for(i = *pq->elements, while i != 0, i - 1 each pass)
        Next = start->right
        node_print(start)
        Start = next
    Return
```

Stack. This implementation of a stack will mimic a linked list, as the pushed nodes can be easily added to the front of the linked list. For this reason, much of the code seen here is taken from my assignment 6 code.

stack structure definition taken from the assignment document:

```
	struct Stack

		uint32_t top;

		uint32_t capacity;

		Node **items;

	stack_create(capacity)

		Stack * stack = allocate(sizeof(Stack))

		stack->capacity = capacity

		stack->*nodes = allocate(capacity, size of (node))

		Return stack

	stack_delete(**stack)

		for(i = stack->top, while i > 0, i - 1 each pass)
			node_delete(stack->nodes[i])
		free(stack->nodes)
		free(*stack)
	Return
//stack empty. Returns true if stack is empty, false if otherwise
stack_empty(pq)
	If stack->top == 0
		Return true
	Else
		Return false
//stack full. Returns true if stack is full, false if otherwise
stack_full(stack)
	If stack->top +1 == stack->capacity
		Return true
	Else
		Return false
//stack size. Returns stack size
```

```
stack_size(stack)
    Return stack->top
//push. Puts a node on the stack, and return true if successful
stack_push(stack, Node)
    If stack_full == true
        Return false
    stack->*nodes[stack->top +1] = node
    Return true
//pop. Removes an element from the stack and processes it
stack_pop(stack, **Node)
    If stack_empty(pq) == true
        Return false
    **Node = stack->nodes[top]
    stack->top -= 1
    Return true
//stack print. Prints the stack
stack_print(stack)
    for(i = stack->top, while i !=  0; i - 1 each pass)

        node_print(stack->node[i])
    Return
```

The code module: The code module is used to store the binary codes for each character as one traverses through the huffman coding tree. It uses a stack implementation to track the steps taken through the tree, and generate a binary code representation of each character.

Code implementation, as per the assignment document:

```
    typedef struct {
        uint32_t top;
        uint8_t bits[MAX_CODE_SIZE ];
     } Code;
//initialize the code, and zero out the bits
Code Code_init(void)
    Code * code = code //I do not yet understand how this constructor is meant to work
```

```
        code->top = 0;
        For(i =0, while i < MAX_CODE_SIZE, i + 1 each pass)
                code->bits[i] = 0;
        Return code
//code size; returns bits pushed onto code
uint32_t code_size(Code *c)
        Return code->top

//Returns true if the Code is empty and false otherwise.
bool code_empty(Code *c)
        If code->top == 0;
                Return true
        Return false

//Returns true if the Code is full and false otherwise.
bool code_full(Code *c)
        If code->top == MAX_CODE_LENGTH
                Return true
        Return false

//set bit in the code
bool code_set_bit(Code *c, uint32_t i)
        If (i > ALPHABET) or( i < 1)
                Return false
        Byte_address = i / 8
        Bit_address = i % 8
        Bit_num = pow(2, (bit_address - 1))
        code->bits[byte_address] = code->bits[byte_address] OR bit_num
        Return true
//clear bit at index i
bool code_clr_bit(Code *c, uint32_t i)
```

```
        If (i > ALPHABET) or( i < 1)
                Return false
        Byte_address = i / 8
        Bit_address = i % 8
        Bit_num = pow(2, (bit_address - 1))
        Bit_num = NOT bit_num
        code->bits[byte_address] = code->bits[byte_address] AND bit_num
        Return true

//returns bit as index i
bool code_get_bit(Code *c, uint32_t i)
        If (i > ALPHABET) or( i < 1)
                Return false
        Byte_address = i / 8
        Bit_address = i % 8
        Bit_num = pow(2, (bit_address - 1))
        if (code->bits[byte_address] AND bit_num) > 0
                Return True
        Else
                Return False

//pushes a bit on the code
bool code_push_bit(Code *c, uint8_t bit)
        If (code_full(c) == true)
                Return false
        c->top += 1
        If bit == 1
                code_set_bit(c, (c->top))
        If bit == 0
                code_clr_bit(c, (c->top))
        Return true
```

```
//pops a bit off the code
bool code_pop_bit(Code *c, uint8_t *bit)
        If (code_empty(c) == true)
                Return false
        Bit = code_get_bit(c, c->top)
        code_clr_bit(c, (c->top))
        c->top -= 1
        Return true

//prints the code
void code_print(Code *c)
        for(i = 0, while i <= c->top, i +1 each pass)
                if(code_get_bit(c, i) == true)
                        Print "1"
                Else
                        Print "0"
        Print newline
        Return
```

IO Module: The purpose of the I/O module is to recreate common stdio.h functions for file management to practice and understand how common functions work.

```
//reading bytes from a file
Int read_bytes(infile, *buffer, n_bytes)
        Counter  = 0;
        while(read(infile, *buffer, 1) != 0)
                Counter += 1
                If counter = n_bytes
                        break
        Return counter
//writing bytes from a file
Int write_bytes(infile, *buffer, n_bytes)
```

```
    Counter  = 0;
    while(write(infile, *buffer, 1) != 0)
        Counter += 1
        If counter = n_bytes
            break
    Return counter

//reading bits
uInt8_t buffer[BLOCK];
Index = 0
Bool read_bit(infile, *bit)
    If index == (BLOCK * 8)
        if(read_byte(infile, buffer, 1) == 0)
            Return false
    Byte_address = index / 8
    bit _address = index % 8
    Bit_num = pow(2, bit_address - 1)
    if(buffer[byte_address] AND bit_num > 0)
        Bit = 1
    Else
        Bit = 0
    Index += 1
    Return true
//writing bits
uInt8_t buffer[BLOCK];
Declare i;
void write_code(outfile, Code *c)
    for(i = 0; while i <= code_size(c), i + 1)
        Buffer[i] = code_get_bit(c, i)
        If i == BLOCKS *8
            write_bytes(outfile, buffer, BLOCK)
```

```
    return
//flush codes
Void flush_codes
    Byte_address = (i / 8) % BLOCK
    For (count = byte_address + 1, while counter < BLOCK, count += 1)
        Buffer[counter] = 0
    write_bytes(outfile, buffer, byte_address)
    return
```

Huffman Coding Module: For creating and interpreting huffman coding related functions

```
//builds a huffman tree
Node *build_tree(uint64_t hist[static ALPHABET])
    PriorityQueue  pq = pq_create(ALPHABET)
    For (i = 0, while i < ALPHABET, i ++)
        If hist[i] > 0
            Node = node_create(i, hist[i])
        pq_enqueue(pq, node)
    Declare Node* parent
    while(pq_size(pq) > 1)
        Node1 = pq_dequeue(pq)
        Node2 = pq_dequeue(pq)
        Node Parent = node_join(node1, node2)
        pq_enqueue(parent)
    Return parent
//build codes. Populates a code table for each symbol in the tree
//auxiliary function. Post order traversal for the code construction
 //pushcode: code for what to add to the node. -1 is root, 0 is left, 1 is right much code taken from
lecture 17 slides
Void code_build(Code *code, node, push code, Code table[ALPHABET])
    If (pushcode == 1)
        code_push(code, 1)
    Else if (pushcode == 0)
```

```
        code_push(code, 0)
    if(node != NULL)
        code_build(code, node->left, 0, codetable)
        if(Node->left == NULL and node->right == NULL)
            table[node->symbol] = code
            code_pop_bit(code, *NULL)
            return
        code_build(code, node->right, 1, code table)
        if(Node->left == NULL and node->right == NULL)
            table[node->symbol] = code
            code_pop_bit(code, *NULL)
            return
        if(code_empty == false)
            code_pop_bit(code, *NULL)
        return

Void build_codes(Node* root, Code table[ALPHABET])
    Code * code = code_init()
    code_build(code, root, table, -1)
    return
```