Design Document Assignment 5

Abhinav Prasanna

November 5, 2021

Description

Assignment is to encode and encode ham codes. Easily do this with the encoders and encoders using data structures such as trees, stack, and queues to make it easier to decode and encode.

Node Pseudocode

```
Node node_create(symbol,frequency){
 Node node = (Node) malloc(sizeof(Node));
 if(node!=NULL){
   node.left = NULL
   node.symbol = symbol
   node.right = NULL
   node.frequency = frequency
 }
 return node
}
void node_delete(node){
 if(n!=NULL){
   free(n)
   n = NULL
 }
 Node *node_join(Node *left, Node *right) {
    Node *node = (Node *) malloc(sizeof(Node));
    if (node != NULL) {
        node->frequency = left->frequency;
        node->frequency += right->frequency;
        node->symbol = '$';
        node->right = right;
        node->left = left;
    }
    return node;
}

void node_print(Node *n) {
    if (n != NULL) {
        printf("%d", n->symbol);
        node_print(n->left);
        node_print(n->right);
    }
}
```

PQ Pseudocode

```
Priority Queue will store nodes in the queue. Nodes with the least frequency have the m
struct PriorityQueue {
    uint32_t capacity;
    uint32_t size;
    Node **items;
};

PriorityQueue *pq_create(uint32_t capacity) {
    PriorityQueue *priorityqueue = (PriorityQueue *) malloc(sizeof(PriorityQueue));
    if (priorityqueue != NULL) {
        priorityqueue->capacity = capacity;
        priorityqueue->size = 0;
        priorityqueue->items = malloc(capacity * sizeof(Node));
    }
    return priorityqueue;
}

static inline int parentindex(uint32_t index) {
    int integer = (int) (index - 1) / 2;
    return integer;
}

static inline uint32_t leftindex(uint32_t index) {
    uint32_t integer = 2 * index + 1;
    return integer;
}

static inline uint32_t rightindex(uint32_t index) {
    uint32_t integer = 2 * index + 2;
    return integer;
}

void pq_delete(PriorityQueue **q) {
    if (*q != NULL && (*q)->items != NULL) {
        free((*q)->items);
        (*q)->items = NULL;
        free(*q);
        *q = NULL;
    }
}

void pq_swap(PriorityQueue *q, uint32_t index, uint32_t index2) {
    Node *temp = q->items[index];
    q->items[index] = q->items[index2];
    q->items[index2] = temp;
}

bool pq_empty(PriorityQueue *q) {
    if (q->size == 0) {
        return true;
    }
    return false;
}

bool pq_full(PriorityQueue *q) {
    if (q->size == q->capacity) {
        return true;
```

Code Pseudocode

```
Code code_init(void) {
    Code code = { 0, { 0 } };
    return code;
}

uint32_t code_size(Code *c) {
    return c->top;
}

bool code_empty(Code *c) {
    if (c->top == 0) {
        return true;
    } else {
        return false;
    }
}

bool code_full(Code *c) {
    if (c->top < MAX_CODE_SIZE) {
        return false;
    }
    return true;
}

bool code_set_bit(Code *c, uint32_t i) {
    if (i < MAX_CODE_SIZE) {
        c->bits[i] = 1;
        return true;
    }
    return false;
}

bool code_clr_bit(Code *c, uint32_t i) {
    if (i < MAX_CODE_SIZE) {
        c->bits[i] = 0;
        return true;
    }
    return false;
}

bool code_get_bit(Code *c, uint32_t i) {
    if (i < MAX_CODE_SIZE && c->bits[i] == 1) {
        return true;
    }
    return false;
}

bool code_push_bit(Code *c, uint8_t bit) {
    if (!(c->top < MAX_CODE_SIZE)) {
        return false;
    }
    if (bit) {
        c->bits[(c->top++ / 8)] |= (1 << (c->top++ % 8));
    } else {
        c->bits[(c->top++ / 8)] &= ~(1 << (c->top++ % 8));
    }
    return true;
```

IO Pseudocode

```
We will be using low level read and write system calls. IO module uses looping calls to
int read_bytes(int infile, uint8_t *buf, int nbytes) {
    int total = 0;
    int bytes = -2;
    bool bytecheck = (bytes > 0 || bytes == -2);
    bool totalcheck = (total != nbytes);
    while (bytecheck && totalcheck) {
        bytes = read(infile, buf, nbytes - total);
        total += bytes;
        bytecheck = (bytes > 0 || bytes == -2);
        totalcheck = total != nbytes;
    }
    bytes_read += total;
    return total;
}

int write_bytes(int outfile, uint8_t *buf, int nbytes) {
    int total = 0;
    int bytes = -2;
    bool bytecheck = (bytes > 0 || bytes == -2);
    bool totalcheck = (total != nbytes);
    while (bytecheck && totalcheck) {
        bytes = write(outfile, buf, nbytes - total);
        total += bytes;
        bytecheck = (bytes > 0 || bytes == -2);
        totalcheck = (total != nbytes);
    }
    bytes_written += total;
    return total;
}

bool read_bit(int infile, uint8_t *bit) {
    if (bit_index == 0) {
        end_buffer = read_bytes(infile, buffer, BLOCK) * 8;
    }
    uint32_t bitposition = bit_index % 8;
    uint32_t byteposition = bit_index / 8;
    *bit = ((buffer[byteposition] >> bitposition) & 1);
    bit_index++;
    bit_index %= (BLOCK * 8);
    if (bit_index > end_buffer) {
        return false;
    }
    return true;
}
```



```
void write_code(int outfile, Code *c) {
    uint32_t index = 0;
    uint32_t bitposition = 0;
    uint32_t byteposition = 0;
    uint8_t bit = 0;
    bool bitcheck = false;
    while (index < c->top) {
        bitposition = bit_index % 8;
        byteposition = bit_index / 8;
        bit = ((buffer[byteposition] >> bitposition) & 1);
        bitcheck = (bit == 1);
```

Stack Pseudocode

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

Stack *stack_create(uint32_t capacity) {
    Stack *stack = calloc(1, sizeof(Stack));
    if (stack != NULL) {
        stack->capacity = capacity;
        stack->items = malloc(capacity * sizeof(Node));
        stack->top = 0;
    }
    return stack;
}

void stack_delete(Stack **s) {
    if (*s != NULL && (*s)->items != NULL) {
        free((*s)->items);
        free(*s);
        *s = NULL;
    }
    return;
}

uint32_t stack_size(Stack *s) {
    return s->top;
}

bool stack_empty(Stack *s) {
    bool returnvalue = false;
    if (s->top == 0) {
        returnvalue = true;
    }
    return returnvalue;
}

bool stack_full(Stack *s) {
    bool returnvalue = false;
    if (s->top == s->capacity) {
        returnvalue = true;
    }
    return returnvalue;
}

bool stack_push(Stack *s, Node *n) {
    bool returnvalue = false;
    if (stack_full(s) == false) {
        returnvalue = true;
        s->items[s->top] = n;
        s->top++;
    }
    return returnvalue;
}

bool stack_pop(Stack *s, Node **n) {
    bool returnvalue = false;
```

Huffman Pseudocode

```
Module has functions that should be used for encoding and decoding files.
Node *build_tree(uint64_t hist[static ALPHABET]) {
    Node *l;
    Node *join = NULL;
    Node *node;
    Node *r;
    PriorityQueue *priorityqueue = pq_create(ALPHABET);
    int index = 0;
    while (index < ALPHABET) {
        if (hist[index] != 0) {
            node = node_create(index, hist[index]);
            enqueue(priorityqueue, node);
        }
    }
    while (pq_size(priorityqueue) > 1) {
        dequeue(priorityqueue, &l);
        dequeue(priorityqueue, &r);
        join = node_join(l, r);
        enqueue(priorityqueue, join);
    }
    dequeue(priorityqueue, &join);
    pq_delete(&priorityqueue);
    return join;
}

void build_codes(Node *root, Code table[static ALPHABET]) {
    static Code code = { 0, { 0 } };
    uint8_t bit;

    if (root != NULL) {
        if (root->left == NULL && root->right == NULL) {
            table[root->symbol] = code;
        } else {
            code_push_bit(&code, 0);
            build_codes(root->left, table);
            code_pop_bit(&code, &bit);
            code_push_bit(&code, 1);
            build_codes(root->right, table);
            code_pop_bit(&code, &bit);
        }
    }
}

void dump_tree(int outfile, Node *root) {
    if (root != NULL) {
        dump_tree(outfile, root->left);
        dump_tree(outfile, root->right);
        if (root->left == NULL && root->right == NULL) {
            outfile += 'L';
            outfile += root->symbol;
        } else {
            outfile += 'I';
        }
    }
}

Node *rebuild_tree(uint16_t nbytes, uint8_t tree[static nbytes]) {
```

Encoder Pseudocode

```
void printhistogram(uint64_t *histogram) {
    int index = 0;
    while (index < ALPHABET) {
        if (histogram[index] != 0) {
            printf("character is %c, amount :%" PRIu64 "\n", index, histogram[index]);
        }
        index++;
    }
}
void printhelp(void) {
    printf("SYNPOSIS\n");
    printf("  A Huffman encode\n");
    printf("  Compress a file using the Huffman coding algorithm\n");
    printf("USAGE\n");
    printf("  ./encode [-h] [-v] [-i infile] [-o outfile]\n\n");
    printf("OPTIONS\n");
    printf("  -h            Display program help and usage.\n");
    printf("  -i infile     Input data to compress (default:stdin)\n");
    printf("  -o outfile    Output of compressed data (default:stdin)\n");
    printf("  -v            Print compression statistics.\n");
}
void postorder_traversal(Node *root, uint8_t *array, uint32_t *index) {
    if (root != NULL) {
        postorder_traversal(root->left, array, index);
        postorder_traversal(root->right, array, index);
        bool rootcheck = (root->left == NULL && root->right == NULL);
        if (rootcheck) {
            array[*index] = 'L';
            index++;
            array[*index] = root->symbol;
            index++;
        } else {
            array[*index] = 'I';
            index++;
        }
    }
}
int main(int argc, char **argv) {
    int br;
    int infile = 0;
    int outfile = 1;
    Code table[ALPHABET] = { 0 };
    Node *root;
    int temp = 0;
    Header h;
    int index = 0;
    uint64_t histogram[ALPHABET];
    uint8_t buffer[BLOCK];
    uint16_t unique_symbols = 0;
    bool verbose = false;
    struct stat instatbuf;
    uint32_t dump_index = 0;
    uint8_t dump[MAX_TREE_SIZE];
    while (index < ALPHABET) {
        histogram[index] = 0;
    }
    histogram[0] += 1;
```

Decoder Pseudocode

```
void print_help(void) {
    printf("SYNPOSIS\n");
    printf("    A Huffman encode\n");
    printf("    Decompresses a file using the Huffman coding algorithm.\n");
    printf("USUAGE\n");
    printf("    ./decode [-h] [-v] [-i infile] [-o outfile]\n\n");
    printf("OPTIONS\n");
    printf("  -h               Display program help and usage.\n");
    printf("  -i infile        Input data to compress (default: stdin)\n");
    printf("  -o outfile       Output of compressed data (default: stdout)\n");
    printf("  -v               Print compression statistics/\n");
}

int main(int argc, char **argv) {
    uint64_t byteswritten = 0;
    Node *node;
    Node *root_node;
    uint8_t bit;
    struct stat instatbuf;
    uint8_t buffer[BLOCK];
    uint32_t buf_index = 0;
    Header header;
    int opt = 0;
    int outfile = 1;
    int infile = 0;
    bool verbose = false;
    while ((opt = getopt(argc, argv, OPTIONS)) != -1) {
        switch (opt) {
        case 'h': print_help(); return -1;
        case 'i':
            infile = open(optarg, O_RDONLY);
            if (infile == -1) {
                printf("Error opening file\n");
                return -1;
            }
            break;
        case 'o':
            outfile = open(optarg, O_WRONLY | O_CREAT | O_TRUNC);
            if (outfile == -1) {
                printf("Error opening file\n");
                return -1;
            }
            break;
        case 'v': verbose = true; break;
        default: print_help(); return -1;
        }
    }
    read_bytes(infile, (uint8_t *) &header, sizeof(Header));
    if (header.magic != MAGIC) {
        fprintf(stderr, "Error: unable to read header. \n");
        return -1;
    }
    fstat(infile, &instatbuf);
    fchmod(outfile, header.permissions);
    uint8_t dump[header.tree_size];
    read_bytes(infile, dump, header.tree_size);
    root_node = rebuild_tree(header.tree_size, dump);
```