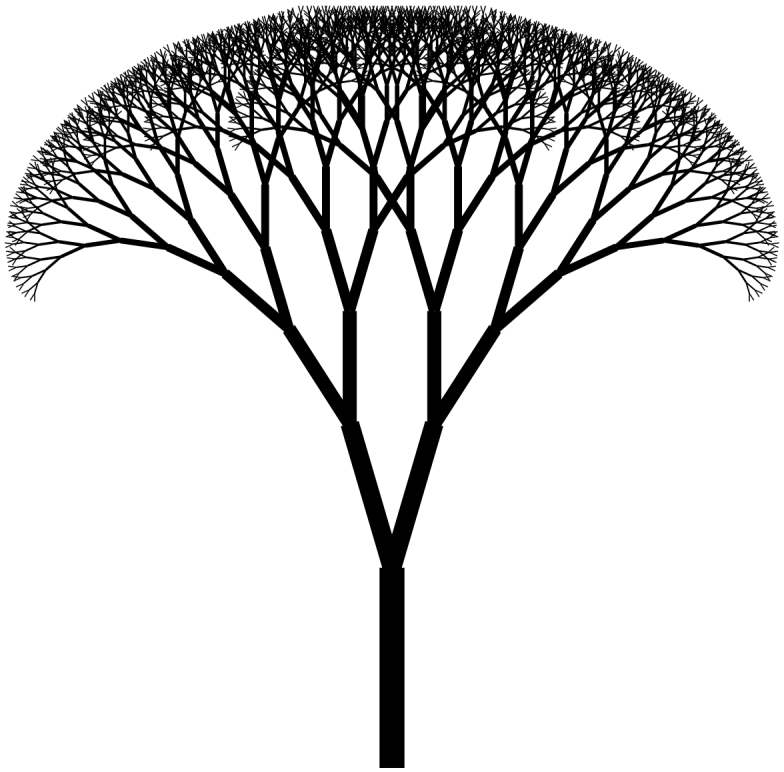
Huffman Coding Design

Ralph Miller

November 7th 2021



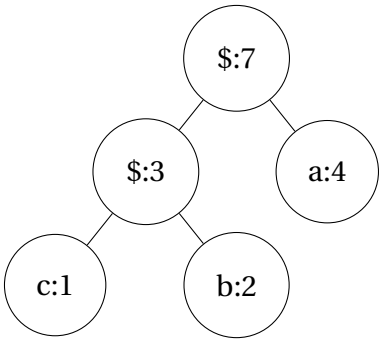
"By the fruits which it bears is the tree known."

Jan Hus

Contents

1 Objective

In this lab assignment we will be implementing a compression algorithm known as Huffman encoding and decoding. This uses a Huffman tree structure to accomplish this. This program will encode an input file and compress it to an output file which can be then decoded.



A Huffman Tree

1.1 Deliverables

```
decode.c : Contains Huffman encoder
encode.c : Contains Huffman decoder
huffman.c: Implements huffman coding module interface
node.c   : Contains Node ADT
pq.c     : Contains Priority Queue ADT
code.c   : Contains Code ADT
io.c     : Implemnts I/O, reads in 4KiB blocks
stack.c  : Contains Stack ATD
Makefile : Makes all binaries
README   : Readme file in markdown
```

2 Encoding

The encoding program will read an infile, or default to stdin, calling IO to read in 4KiB blocks. It will then construct a histogram array that is 256 in size. One element for each ASCII symbol. It will then read the infile byte by byte and add 1 to the element of the histogram for each character read. Then it will use the Priority Queue to construct a Huffman tree, giving each node a parent node which contains the sum of each child node. Then the Huffman tree is traversed and the code to traverse to a symbol is saved into a code table for each element of the histogram. Then a header is made for the compressed file. Finally the header, dumped tree, and codes to decompress the infile are written to the outfile.

2.1 Command-line Options

```
-h: prints out a help message
-v : Print compression statistics to stderr.
-i <input> : Specify input file to compress (stdin by default)
-o <output> : Specify output file to write the compressed input to(stdout by default)
```

3 Decoding

The decoding program will read an infile or default to stdin, calling IO to read in 4KiB blocks. It will read and verify the header. If the headers magic number is not correct then the program will exit. If the header is correct the tree dump is then read from the infile and a huffman tree is rebuild using the dumped tree. Once we have the tree we can read the codes to traverse the tree until a symbol is found. When a symbol is found it is written to the outfile and we traverse the tree from the beginning. This goes on until all of the symbols are written and the encoded file is decoded. The result is the original file written to outfile.

3.1 Command-line Options

```
-h: prints out a help message
-v : Print compression statistics to stderr.
-i <input> : Specify input file to compress (stdin by default)
-o <output> : Specify output file to write the compressed input to(stdout by default)
```

4 Min Heap Algorithm

Lets cover the algorithm behind Min Heap as we will need to use it in our Priority Queue functions. To implement our Min Heap algorithm we will need the functions below.

```
void swap(uint32_t *x, uint32_t *y);
uint32_t min_child(uint32_t *A, uint32_t first, uint32_t last);
void fix_heap(uint32_t *A, uint32_t first, uint32_t last);
void build_min_heap(uint32_t *A, uint32_t first, uint32_t last);
```

4.1 Pseudocode

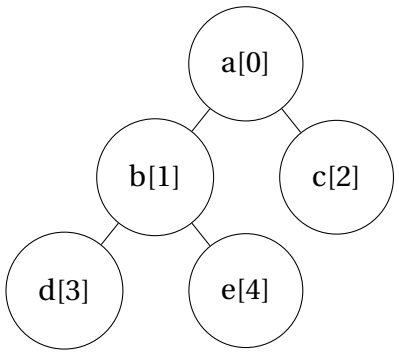
```
// function to swap two indices in an array
void swap(uint32_t *x, uint32_t *y) {
    temp equals *x
    *x equals *y
    *y equals temp
}
// gets the smallest child of parent
uint32_t min_child(uint32_t *A, uint32_t first, uint32_t last) {
    left equals 2 * first
    right equals left + 1

    if right is <= last AND A[right - 1] is < A[left - 1] {
        return right
    }
    return left
}
// recursive function that fixes the heap until Min Heap is reached
void fix_heap(uint32_t *A, uint32_t first, uint32_t last) {
    found = false
    mother = father
    great = min_child(A, mother, last)

    while mother is <= last / 2 AND found is false {
        if A[mother - 1] > A[great - 1] {
            swap A[mother - 1] and A[great - 1]
            mother = great
            great = min_child(A, mother, last)
        }
        else {
            found = true
        }
    }
}
// function to construct min heap
void build_min_heap(uint32_t *A, uint32_t capacity) {
    first = 1
    last = capacity
    for father = last / 2, as long as father is > father - 1, father -- {
        fix_heap(A, father, last)
    }
}
```

5 Priority Queue

The encoder makes use of a priority queue of nodes. Functioning much like a regular queue, but assigns each of its elements a priority. So that elements with a high priority are dequeued first, and those with a low priority are dequeued last. We will be using the Min Heap data structure for our priority queue, as nodes with the lowest frequency will rise to the top, and can be dequeued first. An example of a min heap used for our priority queue is shown below.



ex) Min Heap

5.1 Pseudocode

To implement our priority queue, we will need the functions below:

```
typedef struct PriorityQueue PriorityQueue;
PriorityQueue *pq_create(uint32_t capacity);
void pq_delete(PriorityQueue **q);
bool pq_empty(PriorityQueue *q);
bool pq_full(PriorityQueue *q);
uint32_t pq_size(PriorityQueue *q);
bool enqueue(PriorityQueue *q, Node *n);
bool dequeue(PriorityQueue *q, Node *n);
void pq_print(PriorityQueue *q);
```

```
// struct definition
struct PriorityQueue {
    uint32_t front  // index of front
    uint32_t rear   // index of rear
    uint32_t capacity // capacity of our queue
    uint32_t *items // array pointer
};

// Constructor function
// sets front and rear to 0
// sets capacity to passed in capacity
// allocates memory for the queue using the size of the struct
// allocates memory for items using capacity
// returns a pointer to the queue
PriorityQueue *pq_create(uint32_t capacity) {
    set PriorityQueue *q to malloc(sizeof(PriorityQueue))
    if q exists {
        set q.front to 0
        set q.rear to 0
        set q.capacity to capacity
```

```
        set q.items to calloc(capacity, sizeof(Nodes))
        if q.items no longer exists {
            free q
            set q to NULL
        }
    }
    return q
}

// Destructor function
// frees memory and sets pointer to NULL
void pq_delete(PriorityQueue **q) {
    if *q AND *q.items exist {
        free *q.items
        free *q
        set *q to null
    }
    return
}

// returns true if queue is empty
// returns false otherwise
bool pq_empty(PriorityQueue *q) {
    if front AND rear equal 0 {
        return true
    }
    return false
}

// returns true if queue is full
// returns false otherwise
bool pq_full(PriorityQueue *q) {
    if rear equals q.capacity {
        return true
    }
    return false
}

// returns the number of items currently in the queue
uint32_t pq_size(PriorityQueue *q) {
    return q.rear
}

// enqueues a node into the queue
// return false if the queue is full prior to enqueuing
// return true if successful
bool enqueue(PriorityQueue *q, Node *n) {
    if pq_full is true {
        return false
    }
    q.items[rear] equals n
    increment rear by 1
    return true
}

// dequeues a node into the queue passing it to double pointer n
```

```
// return false if the queue is full prior to dequeuing
// before we dequeue we need to fix the heap making sure it is a min heap
// return true if successful
bool dequeue(PriorityQueue *q, Node **n) {
    if pq_empty is true {
        return false
    }
    call fix_heap() on the queue
    *n equals q.items[front]
    q.items[front] equals 0
    decrement rear by 1
    return true
}

// debug function
// prints out the queue, front and rear index
void pq_print(PriorityQueue *q) {
    for each item in q.items {
        print q.items[item]
    }
    print front and rear
}
```

---

6 Abstract Data Types

6.1 Nodes

This ADT covers the functions of a node in a Huffman tree. Each node contains a pointer to its left and right child. A symbol, and the frequency of that symbol. Below are the functions required for our ADT.

```
typedef struct Node Node;
Node *node_create(uint8_t symbol, uint64_t frequency);
void node_delete(Node **n);
Node *node_join(Node *left, Node *right);
void node_print(Node *n);
```

6.1.1 Pseudocode

```
// Node struct included in Asgn doc

struct Node {
    Node *left;         // Pointer to left child
    Node *right;        // Pointer to right child
    uint8_t symbol;     // Node's symbol
    uint64_t frequency; // Frequency of symbol
};

// Constructor function for a node.
// Sets the nodes symbol and frequency.

Node *node_create(uint8_t symbol, uint64_t frequency) {
    pointer n = Allocated memory using malloc to the size of struct
    point n to symbol
    point n to frequency
    return n
}

// Destructor function for a node.
// frees memory of a node and sets pointer to NULL

void node_delete(Node **n) {
    if (*n AND *n.symbol AND *n.frequency exsists) {
        free *n.symbol
        free *n.frequency
        free *n
        set *n to NULL
    }
    return
}

// joins left and right child node
// returns a pointer to the created parent node
// sets parent nodes children to left and right
// sets parent nodes symbol to '$'
// sets parent nodes frequency to the sum of left and right child frequency.

Node *node_join(Node *left, Node *right) {
```

```
    lr_sum = sum of left and right child frequencies
    symbol = $
    parent pointer = call nodecreate() passing in lr_sum and symbol
    return parent pointer
}

// debugging function
// prints out all nodes with their symbols and frequencies

void node_print(Node *n) {
    for all nodes in the sizeof(Nodes) {
        print *n.symbol
        print *n.frequency
        if (node has children) {
            print children nodes symbol and frequency
        }
    }
}
```

6.2 Stack

This ADT is designed to create a stack for our decoder to reconstruct a Huffman tree. The struct defines variables; top and capacity to traverse the stack, and stores Node items in the stack array. Below are the functions required for our ADT.

```
typedef struct Stack Stack;
Stack *stack_create(uint32_t capacity);
void stack_delete(Stack **s);
bool stack_empty(Stack *s);
bool stack_full(Stack *s);
uint32_t stack_size(Stack *s);
bool stack_push(Stack *s, Node *n);
bool stack_pop(Stack *s, Node **n);
void stack_print(Stack *s);
```

6.2.1 Pseudocode

```
// Stack struct included in Asgn doc

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

// Constructor function for Stack
// sets top to zero
// sets capacity to the passed in capacity
// the maximum number of notes the stack holds is specified by capacity
// allocates memory for the stack using the size of the struct
// allocates memory for items using the capacity
// returns a pointer to the stack

Stack *stack_create(uint32_t capacity) {
    set Stack *s to malloc(sizeof(Stack))
    if s exists {
        set s.top to 0
        set s.capacity to capacity
        set s.items to calloc(capacity, sizeof(Nodes))
        if s.items no longer exists {
            free s
            set s to NULL
        }
    }
    return s
}

// Destructor function for Stack
// frees memory and sets pointer to NULL

void stack_delete(Stack **s) {
    if *s AND *s.items exist {
        free *s.items
```

```
        free *s
        set *s to null
    }
    return
}

// returns the num of items of the stack
// the number of items in the stack is the value of top

uint32_t stack_size(Stack *s) {
    return s.top
}

// returns true if stack is empty
// returns false otherwise

bool stack_empty(Stack *s) {
    if stack_size is 0 { // stack is empty!
        return true
    }
    return false
}

// returns true if stack is full
// returns false otherwise

bool stack_full(Stack *s) {
    if stack_size equals capacity {
        return true
    }
    return false
}

// pushes a node onto the stack
// returns false if stack is full prior to pushing
// returns true if successful

bool stack_push(Stack *s, Node *n) {
    if stack_full is true {
        return false
    }
    put n in top index of items array
    increment top
    return true
}

// pops a node off of the stack
// returns false if stack is empty prior to popping
// return true if successful

bool stack_pop(Stack *s, Node **n) {
    if stack_empty is true {
        return false
    }
    decrement top to previous position
    set *n.items[top]
```

```
    set items[top] to 0
    return true
}

// debug function
// prints top and Stack items

void stack_print(Stack *s) {
    for each item in Stack items {
        print top
        for each i in items[] {
            print items[i]
        }
        print newline
    }
}
```

6.3 Codes

This ADT represents a stack of bits that is maintained while traversing the Huffman tree in order to create a code for each symbol. The struct definition is made transparent so we are able to pass the struct by value. Below are the functions required for our ADT.

```
typedef struct Code;
Code code_init(void);
uint32_t code_size(Code *c);
bool code_empty(Code *c);
bool code_full(Code *c);
bool code_set_bit(Code *c, uint32_t i);
bool code_clr_bit(Code *c, uint32_t i);
bool code_get_bit(Code *c, uint32_t i);
bool code_push_bit(Code *c, uint8_t bit);
bool code_pop_bit(Code *c, uint8_t *bit);
void code_print(Code *c);
```

6.3.1 Pseudocode

```
// Code struct included in Asgn doc

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

// Constructor function for Code
// creates a new code on the stack
// sets top to 0
// zeros out the array of bits
// returns initialized Code

Code code_init(void) {
    set pointer c to Code struct
    set *c.top to 0
    for i = 0 to sizeof(bits[]) {
        set bits[i] to 0
    }
    return pointer c
}

// returns the size of the Code
// ie the number of bits pushed onto the code

uint32_t code_size(Code *c) {
    set counter to 0
    for i = 0, if i < *c.sizeof(bits[]), i++ {
        if bits[i] is NOT 0 {
            counter ++
        }
    }
    return counter
}
```

```
// returns true if Code is empty
// returns false otherwise

bool code_empty(Code *c) {
    for i = 0, if i < *c.sizeof(bits[]), i++ {
        if (bits[i] is NOT 0) {
            return false
        }
    }
    return true
}

// returns true if Code is full
// returns false otherwise
// may need to revisit this ************************

bool code_full(Code *c) {
    for i = 0, if i < *c.sizeof(bits[]), i++ {
        if (bits[i] is 0) {
            return false
        }
    }
    return true
}

// sets the bit at index i in Code to 1
// returns false if i is out of range
// returns true if successful

bool code_set_bit(Code *c, uint32_t i) {
    if i is > MAX_CODE_SIZE {
        return false
    }
    set *c.bits[i] to 1
    return true
}

// sets the bit at index i in Code to 0
// returns false if i is out of range
// returns true if successful

bool code_clr_bit(Code *c, uint32_t i) {
    if i is > MAX_CODE_SIZE {
        return false
    }
    set *c.bits[i] to 0
    return true
}

// gets the bit at index i in Code
// returns false if i is out of range or equals 0
// returns true if bit[i] is equal to 1

bool code_get_bit(Code *c, uint32_t i) {
    if i is > MAX_CODE_SIZE or bit[i] equals 0 {
```

```
        return false
    }
    if bit[i] is equal to 1 {
        return true
    }
    // if none of the conditions are met return false, as there is an error
    return false
}

// pushes a bit onto Code, given by the value of bit passed in
// returns false if the Code is full before pushing
// returns true if successful

bool code_push_bit(Code *c, uint8_t bit) {
    if code_full is true {
        return false
    }
    set *c.bits[top] to bit
    increment top
    return true
}

// pops a bit off of Code
// passes value of popped bit to pointer bit
// returns false if Code is empty before popping
// returns true if successful

bool code_pop_bit(Code *c, uint8_t *bit) {
    if code_empty is true {
        return false
    }
    set *bit to bits[top]
    decrement top
    return true
}

// debug function
// prints the Code stack
// ie value of top and the array of bits

void code_print(Code *c) {
    print top
    for i = 0, i < sizeof(bits[]), i++ {
        print bits[i]
    }
}
```

7 File IO

Since we cant read bits at a time from the storage of the divice, we want to read and write in 4KiB blocks when reading the infile or writing the outfile. We will also use bit vectors to manipulate a buffer so that we are able to read and write the bits from it. To do this we will make use of these functions below.

```
int read_bytes(int infile, uint8_t *buf, int nbytes);
int write_bytes(int outfile, uint8_t *buf, int nbytes);
bool read_bit(int infile, uint8_t *bit);
void write_code(int outfile, Code *c);
void flush_codes(int outfile);
```

7.1 Pseudocode

```
//loops calls to read() into buffer
//returns the amount of bytes read
int read_bytes(int infile, uint8_t *buf, int nbytes) {
    while bytesread < BLOCK {
        bytesread = read infile
    }
    return bytesread
}
//loops calls to write() out of buffer
//returns the amount of bytes written
int write_bytes(int outfile, uint8_t *buf, int nbytes) {
    while bytes_written < BLOCK {
        bytes_written = write outfile
    }
}
//reads individual bits out of the buffer
//returns true if bit is a 1, false otherwise
bool read_bit(int infile, uint8_t *bit) {
    call read_bytes()
    for all bits in buffer {
        if bit is 1 return true
        else return false
    }
}
//writes each bit of the code into the buffer
void write_code(int outfile, Code *c) {
    for all bits in c {
        buffer[bit_index] = bit
        if bit_index == BLOCK {
            flush_codes
        }
    }
    flush_codes
}
//calls write_byes to write code into outfile
void flush_codes(int outfile) {
    call write_bytes for all codes in buffer
}
```

8 Huffman Coding Module

The Huffman Coding Module is contained in huffman.c The functions below are used by the encoder and decoder to create and destroy Huffman trees.

```
Node *build_tree(uint64_t hist[static ALPHABET]);
void build_codes(Node *root, Code table[static ALPHABET]);
void dump_tree(int outfile, Node *root);
Node *rebuild_tree(uint16_t nbytes, uint8_t tree_dump[static nbytes]);
void delete_tree(Node **root);
```

8.1 Pseudocode

```
//builds a huffman tree using the histogram and priority queue
/returns root of huffman tree
Node *build_tree(uint64_t hist[static ALPHABET]) {
    create priority queue
    for each symbol in histogram {
        create a node
        set symbol
        set frequency
        enqueue node
    }
    create priority queue
    while pq_size > 1 {
       dequeue a left node
       dequeue a right node
       parent = node_join(left, right)
       enqueue parent
    }
    dequeue root
    return root
}
//populates code table for each symbol in the tree
void build_codes(Node *root, Code table[static ALPHABET]) {
    traverse tree
    if the root is not NULL {
        go left
        add 0 to the code
        if root.left is not a leaf {
            call build_codes(root.left)
        }
        go right
        add 1 to the code
        if root.right is not a leaf {
            call build_codes(root.right)
        }
        //we have our code so now add it to the table
        Code table[symbol] = code
    }
}
//writes to outfile the tree string
//ex LaLbLcII
//a L indicates a Leaf an I indicates an iterior node or parent
```

```
void dump_tree(int outfile, Node *root) {
    traverse tree
    if the root is not NULL {
        go left
        if root.left is not a leaf {
            write I
            call dump_tree(root.left)
        }
        go right
        if root.right is not a leaf {
            write I
            call build_codes(root.right)
        }

    //it is a leaf
    write L
    write root.symbol
    }
}
//rebuilds a tree given its post-order tree dump stored in tree_dump
Node *rebuild_tree(uint16_t nbytes, uint8_t tree_dump[static nbytes]) {
    for all bytes in dump_trees {
        if tree_dump[i] = L {
            Node n = tree_dump[i + 1]
            enqueue n
            increment i
        }
        if tree_dump[i] = L {
           dequeue left
           dequeue right
           Node root = $
           root.left = left
           root.right = right
           enqueue root
        }
    }
    dequeue root
    return root
}
//does a post order traversal to free all memory of nodes
void delete_tree(Node **root) {
    traverse tree
        if the root is not NULL {
        go left
        if root.left is not a leaf {
            free(root)
            call delete_tree(root.left)
        }
        go right
        if root.right is not a leaf {
            free(root)
            call delete_tree(root.right)
        }
}
```