Bill Zhang
jzhan411@ucsc.edu
05/06/2021

CSE13S Spring 2021
Assignment 6: Huffman Coding
Design Document

This program will compress and extract a file using Huffman encoding.

Node adt: Contains pointer to left, pointer to right, symbol and frequency of symbol.

Struct node

Node *left
Node *right
Int symbol
Int frequency

Node *node_create(uint8_t symbol, uint64_t frequency)
The constructor for a node. Sets the node's symbol as symbol and its frequency as frequency

void node_delete(Node **n)
The destructor for a node. Make sure to set the pointer to NULL after freeing the memory for a node.

Node *node_join(Node *left, Node *right)
Joins a left child node and right child node, returning a pointer to a created parent node. The parent node's left child will be left and its right child will be right. The parent node's symbol will be '$' and its frequency the sum of its left child's frequency and its right child's frequency.

Priority Queue

PriorityQueue *pq_create(uint32_t capacity)
The constructor for a priority queue. The priority queue's maximum capacity is specified by capacity.

void pq_delete(PriorityQueue **q)
The destructor for a priority queue. Make sure to set the pointer to NULL after freeing the memory for a priority queue

bool pq_empty(PriorityQueue *q)
Returns true if the priority queue is empty and false otherwise

bool pq_full(PriorityQueue *q)
Returns true if the priority queue is full and false otherwise

bool dequeue(PriorityQueue *q, Node **n)
Dequeues a node from the priority queue, passing it back through the double pointer n. The node dequeued should have the highest priority over all the nodes in the priority queue. Returns false if the
priority queue is empty prior to dequeuing a node and true otherwise to indicate the successful dequeuing of a node.

Code Struct

Struct code
    Int top
    Int bits[MAX_CODE_SIZE] max code size = alphabet 2

Code code_init(void)
    New code on stack
    Top = 0
    Bits = 0

uint32_t code_size(Code *c)
Returns the size of the Code, which is exactly the number of bits pushed onto the Code.

bool code_empty(Code *c)
Returns true if the Code is empty and false otherwise.

bool code_full(Code *c)
Returns true if the Code is full and false otherwise.

bool code_push_bit(Code *c, uint8_t bit)
Pushes a bit onto the Code. The value of the bit to push is given by bit. Returns false if the Code is full
prior to pushing a bit and true otherwise to indicate the successful pushing of a bit.

bool code_pop_bit(Code *c, uint8_t *bit)
Pops a bit off the Code. The value of the popped bit is passed back with the pointer bit. Returns false if
the Code is empty prior to popping a bit and true otherwise to indicate the successful popping of a bit.

IO

int read_bytes(int infile, uint8_t *buf, int nbytes)

int read_bytes(int infile, 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)

Stack Adt
Struct stack
    Int top
    Int capacity
    Node ** items

Stack *stack_create(uint32_t capacity)
The constructor for a stack. The maximum number of nodes the stack can hold is specified by capacity.

void stack_delete(Stack **s)
The destructor for a stack. Remember to set the pointer to NULL after you free the memory allocated by the stack.

void stack_delete(Stack **s)
The destructor for a stack. Remember to set the pointer to NULL after you free the memory allocated by the stack.

void stack_delete(Stack **s)

The destructor for a stack. Remember to set the pointer to NULL after you free the memory allocated by
the stack.

void stack_delete(Stack **s)
The destructor for a stack. Remember to set the pointer to NULL after you free the memory allocated by
the stack.

bool stack_push(Stack *s, Node *n)
Pushes a node onto the stack. Returns false if the stack is full prior to pushing the node and true otherwise to indicate the successful pushing of a node

bool stack_pop(Stack *s, Node **n)
Pops a node off the stack, passing it back through the double pointer n. Returns false if the stack is empty prior to popping a node and true otherwise to indicate the successful popping of a node.

Huffman Encoding Mod

Node *build_tree(uint64_t hist[static ALPHABET])
Constructs a Huffman tree given a computed histogram

void build_codes(Node *root, Code table[static ALPHABET])
Populates a code table, building the code for each symbol in the Huffman tree.

Node *rebuild_tree(uint16_t nbytes, uint8_t tree_dump[static nbytes])
Reconstructs a Huffman tree given its post-order tree dump stored in the array tree_dump. The length
in bytes of tree_dump is given by nbytes. Returns the root node of the reconstructed tree.

void delete_tree(Node **root)
The destructor for a Huffman tree. This will require a post-order traversal of the tree to free all the nodes.
Remember to set the pointer to NULL after you are finished freeing all the allocated memory