ASGN 6 Huffman Coding

Justin Satriano

pq.c

Priority Queue puts items in a queue depending on how frequent that item appears.

```
Struct PriorityQueue
        uint32_t capacity
        uint32_t head
        uint32_t tail
        uint32_t size
        int64_t *items
PriorityQueue *pq_create
        PriorityQueue *q = (PriorityQueue*)
           malloc(sizeof(PriorityQueue))
        q->capacity = capacity
            q->head = 0
            q->tail = 0
            q->size = 0
            q->slot = 0
            q->items = (int64_t*)
              calloc(capacity,
                sizeof(int64_t))
            if (!q->items)
                free(q)
                q = NULL
        return q
void pq_delete
        if (*q)
            free (*q)
            *q = NULL
        return
bool pq_empty
        return q->size == 0
bool pq_full
        return q->size == q->capacity
uint32_t pq_size
        return q->size
bool enqueue
        if (pq_full(q))
            return false
        else if (pq_empty(q))
            q->items[q->tail] = x
            q->size += 1
            q->tail = (q->tail+1)%
                q->capacity
        else q->slot = q->tail
while uint32 n != q->size
          if q->items[(q->slot-1)% capacity] > x
              q->items[q->slot] = q->items[(q->slot+1)% q->capacity]
              q->slot -= 1
          else
              q->items[q->slot] = x
              q->size += 1
              q->tail = (q->tail + 1) % q->capacity
              return true
          n += 1
bool dequeue
        if (pq-empty(q))
            return false
        else
          n = q->items[q->head]
          q->size -= 1
          q->head = (q->head+1)% q->capacity
          return true
void pq_print
        debug func
```

node.c

Nodes are used in the huffman tree and contain a left and right child, a symbol and the frequency of that symbol

```
Struct Node
    Node* left
    Node* right
    uint8_t symbol
    uint64_t frequency
Node *node_create
        Node *n = (Node*) malloc(sizeof(Node))
        n->symbol = symbol
        n->frequency = frequency
          n->left = left
          n->right = right
    return n
void node_delete
      if (*n)
         free(*n)
          *n = NULL
Node *node_join
      Node *parent = node_create($, left->frequency
                                  + right->frequency)
      parent->left = left
      Parent->right = right
      return parent
void node_print
     debug func
```

Code.c

assigns codes to each character/element that appeared in the source file. Represents a stack of bits

```
typedef struct Code
        uint32_t top
        uint8_t bits[MAX_CODE_SIZE]
code code_init
        Code c
        c.top = 0
      return c
uint32_t code_size
     return c->top
bool code_empty
     return c->top == 0
bool code_full
     return c->top == MAX_CODE_SIZE
bool code_push_bit
     if (code_full(c))
         return false
     else
         c->bits[c->top] = bit
         c->top += 1
         return true
bool code_pop-bit
     if (code_empty(c))
         return false
     else
         c->top -= 1
         *bit = c->bits[s->top]
          return true
void code_print
     debug func
```

io.c

make static
int buf_index = 0
uint8_t buf[BLOCK]

used in encoder and decoder. io.c acts as a low-level system call that reads and writes bits from/to files.

```
int read_bytes
     int total_bytes    // keeps track of bytes read so far
     int read_bytes     // keeps track of # of bytes we read in
                              everytime we call read
   While (bytes > 0 && total_bytes != nbytes)
          read_bytes = read(infile, buf, nbytes - total_bytes)
          total_bytes += read_bytes
   return total_bytes
int write_bytes
     int total_bytes //  "   "   "   "   "
     int read_bytes  //  "   "   "   "   "
     while (bytes > 0 && total != nbytes)
          read_bytes = write(outfile, buf, nbytes - total_bytes
          total_bytes += read_bytes
     return total_bytes
void write_code
     for (i = 0; i < code_size(c); i += 1)      * able
         if  get_bit(c, i) == 1                        to write
              set_bit(buf, buf_index)          get_bit, set_bit,
         else                                  clr_bit functions
              clr_bit(buf, buf_index)          in code.c?
         buf_index += 1
         if buf_index == 8 * BLOCK
              write_bytes(outfile, buf, BLOCK)
              buf_index = 0
void flush_codes
     if buf_ind > 0
         convert # of bits left   // look at
         in buffer to # of bytes   asgn5 bitvector
         to write                  create func
void read_bit
     if  buf_ind == 0?
         fill up the buffer
     bit = bit at buf_index
     buf_index += 1
```

Stack.c

```
Struct stack
     uint32_t top
     uint32_t capacity
     Node ** items
Stack stack_create
     Stack *s = (Stack *)malloc(sizeof(Stack))
     s->top = 0
     s->capacity = capacity
     s->items = (int64_t*)calloc(capacity,
                                 sizeof(int64_t))
     if (!s->items)
          free(s)
          s = NULL
void stack_delete
    if (*s && (*s)->items)
          free((*s)->items)
          free(*s)
         *s = NULL
bool stack_empty
     return s->top == 0
bool stack_full
     return s->top == s->capacity
uint32_t stack_size
     return s->top
bool stack_push
     if (stack_full(s))
          return false
     else
          s->items[s->top] = n
          s->top += 1
          return true
bool stack_pop
     if (stack_empty(s))
          return false
     else
          s->top -= 1
          *n = s->items[s->top]
          return true
void stack_print
     debug func
```

encoder

1) create histogram. This is a 256 long array of uint64_ts
2) increment element 0 and 255 by 1
3) create huffman tree using build_tree
   this requires making a priorityqueue, and enqueueing/dequeueing until there is only 1 node left in the queue.
4) create a code table by traversing the huffman tree. Use build_codes
5) Make a header (struct is in header.h)
6) Write header to outfile
7) Post-order traveral of huffman tree to outfile
8) write corresponding code to each symbol in outfile using write_code
9) close files

Decoder

1) Read header from infile. if magic number does not match 0xDEADBEEF, display error message and exit program
2) Set permissions using fchmod
3) Read dumped tree from infile into an array that is tree_size bytes long. Then reconstruct the tree using rebuild_tree
4) read infile 1 bit at a time using read_bit.

huffman.c

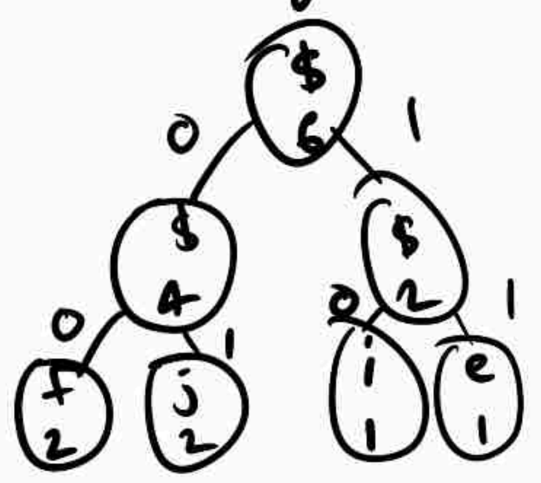
Node *build_tree

after creating a histogram in the encoder, this will construct a huffman tree and return the root node

void build_codes

Populates a code table (the way the histogram was populated?)

f: 00
j: 01
i: 10
e: 11



Node *rebuild_tree

Using tree_dump from decoder, will reconstruct a huffman tree. Returns root node of reconstructed tree

void delete_tree

Destructor for the tree
requires post-order traveral
to free all nodes
set *root = NULL