DESIGN

Trie ADT

```
def trie_node_create(input code):
  TrieNode tn = allocate memory
   For child in tn.children:
     set child to NULL
   return tn

def trie_node_delete(input TrieNode n):
  free(n)
  Set n to NULL

def trie_create():
  TrieNode *root = trie_node_create(empty code)
  return root

def trie_delete(input TrieNode ptr n):
  For child in n.children:
    if(child isn't NULL):
      trie_delete(child)
    trie_node_delete(child)
   trie_node_delete(n)
   Set n to NULL

def trie_reset(input root node):
  for child in root.children:
    If child isn't NULL:
      trie_delete(child)

def trie_step(input node, symbol_integer):
      return node.children[symbol_integer]
```

Word ADT

```
def word_create(input syms array, input len):
  Word *w = allocate memory
  w.len = input len
  If w.len > 0:
    W.syms = allocate memory size of len * byte
    For byte in w.syms:
      Byte = syms[byte]
  return w

def word_append_sym(input Word, symbol integer):
  If word is empty:
    return word_create(symbol integer, 1)
  If word isn’t empty;
    word.len ++
    Word.symsp[len] = symbol integer
  Return new word

def word_delete(input Word):
  If word isn’t NULL:
    If word.syms isn’t NULL
      free(word.syms)
      Set to word.syms to NULL

def wt_create():
  WordTable wt = allocate memory the size of MAX_CODE(256) number of Words
  wt[EMPTY_CODE] = word_create(0,0)
  return wt

def wt_reset(input Wordtable ptr wt):
  For all words ‘x’ except the empty string:
    word_delete(x)
     free(x)
     Set x to NULL

def wt_delete(input Wordtable ptr wt):
  for all words ‘x’ in Wordtable:
  If x isn’t NULL:
    word_delete(x)
    Set x to NULL
free(wt)
Set wt to NULL
```

Note:BitVector ADT used from assignment 4

IO Functions

Global vars: Bitvector ptr buffer(for writing pairs), wbuff(buffer for writing words), bpi(bit index for buffer pair bits), bwi(byte index for writing word symbs to buffer)

```
def read_sym(input infile, sym ptr):
  static curr_index = 0
  static end_file_index = 0
  static read_size = 0
  if buffer doesn’t exist OR curr_index = 4096 OR curr_index == eof
    If buffer exists:
      Free(buffer)
      Set to NULL
   Create buffer and allocate memory the size of 4KB
   Curr_index = 0
   R_size = read(infile, buffer, BLOCK(4KB)
if(0 bytes weren’t read):
  If 4KB read:
    Sym = buffer[curr_index]
    curr_index++
  Else if less than 4kb read:
    Sym = buffer[curr_index]
    Curr_index++
Else:
  If buffer exists:
    free(buffer)
    Set buffer to NULL
  return false

Return true
```

```
def buffer_pair(input outfile, code, sym, bitlen):

  If buffer is full or uninitialized:

    If buffer is full:
       free(buffer)
        Symbol_index = 0
    Buffer = bv_create(8*(BLOCK-1))

    for i in range(bitlen):
      masked = code ANDed with 1 << i
      bit = masked >> i
      If bit == 1:
         bv_set_bit(buffer,i)
      Else if:
         bv_clr_bit(buffer,i)
      If buffer is full:
         write(outfile,buffer->vector,BLOCK);
         Flush buffer
   Index = final iteration of i

   for i in range(8):
     masked = code ANDed with 1 << i
     bit = masked >> i
     if bit == 1:
       bv_set_bit(buffer, i)
     Else if:
       bv_clr_bit(buffer, i)
     If buffer is full:
        write(outfile,buffer->vector,BLOCK);
        Flush buffer


def read_pair(input infile, code ptr, sym ptr, bitlen):
  read_index = 0
  R_size = 0
  if read_index == R_size*8 or buffer == NULL:
    if(buffer)
      delete(buffer)
      Set buffer to NULL
      buffer = bv_create(8*(BLOCK - 1));
      R_size = read(infile, buffer->vector, BLOCK)
   read_index = 0
   Temp_code = 0
   Temp_sym = 0

    if(r_size != 0):
       if(r_size ==);
         For i in range(bitlen):
            Bit = bv_get_bit(buffer,read_index)
            If bit == 1:
              Temp_code = Temp_code OR (1 left shift by i)
         If read_index ==BLOCK**:
           read_index = 0
           R_size = read(infile, buffer->vector, BLOCK)
      Code = &Temp_code

         For i in range(8)
            Bit = bv_get_bit(buffer,read_index)
            If bit == 1:
              Temp_sym = sym_code OR (1 left shift by i)
           If read_index == R_size:
             read_index = 0
             R_size = read(infile, buffer->vector, BLOCK)

        Sym = &Temp_sym



def flush_pairs(input outfile):
  If buffer != NULL:
    write(outfile, buffer-vector, byte(bpi))

def buffer_word(input outfile, Word *w):
  for x in range(w->len):
    bwi++
    if(bwi < BLOCK):
      wbuff[bwi] = w->syms[x]
  if(bwi == BLOCK):
   write(outfile,wbuff,BLOCK)

def flush_word(input outfile):
  write(outfile,wbuff,bwi)


def read_header(input infile, input FileHeader ptr h):
  read(infile,h, size of FileHeader)
  If not little endian:
    h.magic = swap32(h.magic)
    h.protection = swap16(h->protection)


def write_header(input infile, input FileHeader ptr h):
  If not little endian:
    h.magic = swap32(h.magic)
    h.protection = swap16(h->protection)
  write(infile,h, size of FileHeader)
```

Pseudocode for bit length:

```
blength(input code):
  If code != 0:
    Return log(code) + 1
  Else:
    Return 1
```

Code for encode.c

Note:Code for LZ-78 algorithm was based on pseudocode provided in asgn7.pdf document.

```
 while(getopt != -1)
    if choice == -v:
      display_stat = true
    If choice == -i
      Input_file_descriptor = open(optarg, readonly)
    If choice == -o:
      Output_file_descriptor = open(optarg, writeonly | create_if_not_existing)

  Input_filebuff = fstat()
  Output_filebuff = stat()
  FileHeader fh = allocate memory
  fh->magic  = magic_number
  fh->protection_bits = Input_filebuff.protection_bits

  write_header(Output_file_descriptor,fh)
Run LZ-78 encode Algorithm on input file(original file) and write encodings to output file(compressed file)

if(display_stat):
  print compressed file size
  print decompressed file size
  Compression_ratio = 100 * (1 - compressed file size/decompressed file size)

trie_delete(root)
free(fileheader fh)
close(input file)
close(output file)
```

Code for decode.c

```
while(getopt != -1)
    if choice == -v:
      Display_stat = true
    If choice == -i
      Input_file_descriptor = open(optarg, readonly)
    If choice == -o:
      Output_file_descriptor = open(optarg, writeonly | create_if_not_existing)

  Input_filebuff = fstat()
  Output_filebuff = stat()
  FileHeader fh = allocate memory
_read_header(Input_file_descriptor,fh)
_if (fh.magic = magic number):
    run LZ-78 decode Algorithm on input file(compressed file) and write decompression to output file(decompressed file)


if(display_stat):
  print compressed file size
  print decompressed file size
  Compression_ratio = 100 * (1 - compressed file size/decompressed file size)

free(fh)
wt_delete(wordtable ptr)
close(output file)
close(input file)
```