DESIGN

Encoder:

Main:

```
 inputfile;
 table hist[255];
 table code[255];

 //setting histogram table
 for(each byte in inputfile)
 {
   hist[byte]++;
 }

 //setting Queue
 PriorityQueue pq;
 for(each symbol in hist > 0 … i<255)
 {
  //symbol(i) is its ASCII char equivalent of I
  //i = 97 = ‘a’
  pq_insert(new node(symbol(i),hist[i]));
 }

 //setting tree
 while((size(pq) > 1) and (pq.top != parent node))
 {
   Node left = dequeue(pq);
   Node right = dequeue(pq);
   Node parent = join(left,right);
   enqueue(parent);
 }
```

```
//continued
   Node *root  = dequeue(pq);

//building code table
   Code *c
   post_order_traverse(Node root, Code c, table code[255] );\

//building header
Header h;
h.magic_number = MAGIC
h.Inputfilesize = infile.size;(in bytes)
h.Treesize = root.size;(in bytes)
h.Permission  = infile.permission;

//encoding message

   //writing header to outfile
   write_bytes(outfile,(in bytes)h);

  //writing codes for each symbol from infile to outfile
   For(each byte in inputfile)
   {
     write_code(outfile,code[symbol(byte)])
   }
   flush_code(outfile);

   //dumping tree into outfile
   dump_tree(outfile,root);

Delete(root)
```

END of MAIN

DESIGN

Encoder:

//Encoder helper functions

```
Post_order_traversal(Node root, Code c, table code[255])
{
    if(root->left != null)
    {
      c.pushbit(0)
      root = root->left;
    }
    if(root->right != null)
    {
      c.pushbit(1);
     root = root->right;
    }
     code[root->symbol] = c;
     c.popbit();
}

Write_bytes(outfile,buffer)
{
    for(each byte in buffer)
   {
     store byte into outfile
    }
}
```

```
buffer[block];
buff_index;
write_code(outfile, code c)
{
  buffer[buff_index] += c;
  buff_index += c;
  if(buffer is full)
  {
   output each byte from buffer in outfile
   }
}

Flush_code(outfile)
{
  output each byte from buffer in outfile
}

Dump_tree(outfile,root)
{
  Post_order_traverse (root);
  if(leaf is reached)
  {
    output ‘L’ and root->symbol into outfile
  }
  if(parent’s left and right is visited)
  {
   output ‘I’ into outfile
   }
}
```

DESIGN

Decoder:

Main:

```
inputfile;
outfile;
Header h;
read_bytes(inputfile,h,sizeof(header));
If(h.magic_number != MAGIC)
{
  exit(EXIT_FAILURE);
}

Dumped_tree[h.treesize];
Read_bytes(inputfile,dumped_treep[],h.treesize);
Node *rebuilded_tree = rebuildtree(dumped_tree);

Node *n;
For(each bit in inputfile)
{
    if(n == parent_node)
    {
      input n->symbol into outfile;
    }
    if(bit == 0)
    {
       n = n->left;
    }
    if(bit == 1)
    {
      n = n->right;
    }
}
```

END OF MAIN

Helper Functions from Decoder:

```
Tree * Rebuildtree(dumped_tree[tree_size])
{
  Stack s;
  for(each char in dumped_tree)
  {
    if(dumped_tree[i] == ‘L’)
    {
      push new_node(symbol(i)) onto stack
    }
    if(dumped_tree[i] == ‘I’)
    {
     Node *right = pop stack s;
     Node *left = pop stack s;
     Node *parent = new node(right,left)//right left as children
    }

    return (pop stack s);

}

Read_byte(inputfile, buffer, n bytes to read)
{
    read n bytes from inputfile and store into buffer;
}
```

DESIGN

Priority Queues:

```
Struct PriorityQueue
{
 curr_size;
 Node head;
 Node tail;
 capacity
}
pq pq_create(capacity)
{
 pq = malloc (capacity * sizeof(PriorityQueue));
 pq_size = 0;
 head = tail = null;
 pq_capacity = capacity;
 return pq;
}

Void pq_delete(PriorityQueue q)
{
 free (q);
}

bool pq_empty(PriorityQueue *q)
{
Return (curr_size == 0);
}
```

```
bool pq_full(PriorityQueue *q)
{
Return (capacity == curr_size);
}//end of pq_full

Bool enqueue(PriorityQueue *q, Node *n)
{
If(curr_size == capacity)
{
Return false;
}

For( each element in q)
{
If (n > q[current_node])
{
Insert n before q[current_node];
}
}//end of for-loop

Return true;
}//end of enqueue
```

```
Bool dequeue(PriorityQueue *q, Node**n)
{
If(curr_size <=0)
{
Return false
}

(n) = &(*q_head);
q_head = q_head_next;

Return true;
}
```

DESIGN

Stack:

```
Struct Stack
{
 top
 capacity
 node_array[]
}
s stack_create(capacity)
{
 s= malloc (capacity * sizeof(Stack));
 s_top = 0;
 s_capacity = capacity;
 node_array[capacity] = malloc (sizeof(node)*capacity);
 return s;
}

Void stack_delete(Stack s)
{
 free (all nodes in node_array[capacity]);
 free (s);
}

bool stack_empty(Stack s)
{
Return (s_top == 0);
}
```

```
bool stack_full(Stack s)
{
Return (s_top == s_capacity);
}//end of pq_full

Int stack_size(Stack s)
{
Return s_top;
}

Bool stack_push(Stack s, Node n)
{
if(stack_full(s))
{
Return false;
}
Node_array[s_top] = n;
s_top++;
Return true;
}
```

```
Bool stack_pop(Stack*s, Node**n)
{
If(s_top <=0)
{
Return false
}

(n) = &(node_array[s_top-1]);
Node_array[s_top-1] = null;
S_top--;

Return true;
}
```

DESIGN

Codes:

```
Struct Code
{
  top;
  uint8_t bit_stack[256/8];
}

Code code_init()
{
 Code c;
 c.top = 0;
 c.bit_stack = {0,0,0…};
 return c;
}

Int code_size(Code c)
{
 return c.top;
}

Bool code_empty(Code c)
{
  return c.top == 0;
}

Bool code_full(Code c)
{
  return c.top == 256;
}
```

```
Bool code_push_bit(Code c, bit)
{
  If(c.top >= 256)
  {
  Return false;
  }
 if (bit == 0)
 {
   c.top++;
   return true;
  }
  else
  {
    int temp= 1
    int group = c.top/8;
    int index = c.top%8;
    temp<<index;
    c.bit_stack[group] | temp;
  }
  c.top++;
 return true;
}
```

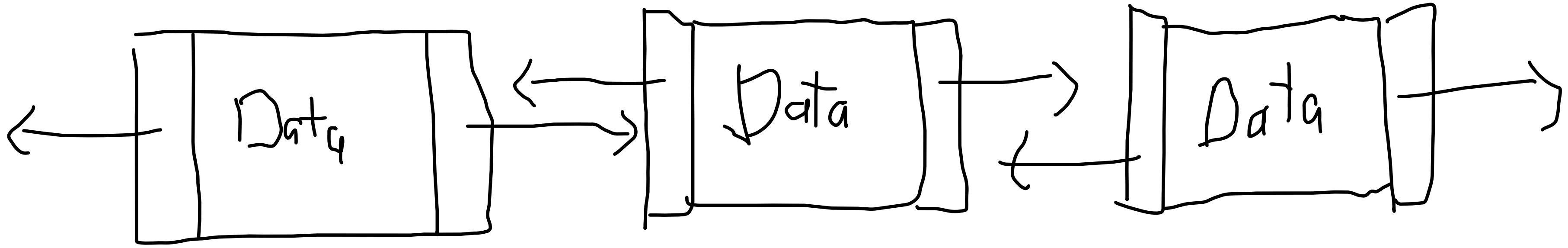
```
Bool code_pop_bit(Code c, bit)
{
  If(c.top >= 256)
  {
  Return false;
  }
 if (bit == 0)
 {
   c.top++;
   return true;
  }
  //getting bit
  int group = c.top/8;
  int index = c.top%8;
  int temp = c.bit_stack[group];
  temp << (7-index);
  temp >> 7;
  bit = temp;

  //removing top bit
  if(bit == 1)
  {
   temp = 1;
   temp << index;
   ~temp;
   c.bit_stack[group] = c.bit_stack[group] &temp;
   }
  c.top -=1;
 return true;
}
```

DESIGN

Node ADT:

-Data fields :

1. Symbol
2. Frequency
3. Node ptr_left
4. Node ptr_right



DESIGN

Nodes:

```
Struct Node
{
  Node left;
  Node right;
  char symbol;
  int frequency;
}

Node node_create(symbol, frequency)
{
 Node n = malloc;
 n.symbol = symbol
 n.frequency = frequency;
 n.left = n;
 n.right = n;
}

node_delete(Node *n)
{
 Free(n)
}

Node * join(Node left, Node right)
{
 Node parent = node_create(‘$’,  (left.frequency + right.frequency) );
 return parent;
}
```