Assignment 6 Design Document

This assignment seems to be the most involved and detailed assignment that I will be doing, and thus this design document will have a lot of information to cover. I will going over the different ADT and methods that I will be programming and will give my thought process on how I am going to go about implementing them.

Nodes

Nodes are defined by their left child, right child, symbol they represent, and their frequency. There are several methods that a node needs to implement.

```
node_create(symbol, frequency):
    n = dynamically_allocated_space_of_node()
    n.symbol = symbol
    n.frequency = frequency
    n.right = no_child
    n.left = no_child
    return n

node_delete(Node n):
    free_dynamically_allocated_memory(n)
    set_pointer_to_null(n)
    return

node_join(Node left, Node right):
    n = node_create('$', left.frequency+right.frequency)
    n.left = left
    n.right = right
    return

node_print(Node n):
    print_to_stderr(n.symbol)
    if(!is_empty(n.left))
    node_print(n.left)
    if(!is_empty(n.right))
    node_print(n.right)
    return
```

Priority Queues

I have already written code for a regular queue for a prior assignment, and thus I will be borrowing from that code with regards to certain trivial methods. After thinking about how I would like to implement this ADT, I have decided that I will be my priority queue will be made up of 2 arrays, one of the Nodes

themselves, and one of the 'indices' of each node, independent of the ordering of the nodes within the node array. This will mimic an insertion sort whenever a Node is enqueued or dequeued. I will be performing the insertion algorithm on the array of indices rather than the nodes. Finally, I think that I will have a stack as part of the implementation to keep track of the available indices in the node array. Whenever I dequeue a node, I will push the index of where I removed it to the stack to indicate that that index is free. When I enqueue a node, I will pop an index from the stack and store the node there.

```
pq_create(capacity):
    pq = dynamically_allocated_pq(pq_size)
    pq.node_capacity = capacity
    pq.node_array = dynamically_allocated_array(capacity, pq_size)
    pq.index_array = dynamically_allocated_array(capacity, int_size)
    pq.index_stack = dynamically_allocated_stack(capacity, stack_size)
    for(i in range(capacity):
        pq.index_stack.push(i)
        pq.index_array[i] = -1
        pq.node_array[i] = null_node
    return pq

pq_delete(q):
    free_memory(q.node_array)
    free_memory(q.index_array)
    free_memory(q.index_stack)
    free_memory(q)
    q = null_pointer
    return

pq_empty(q):
    return q.index_stack.size == capacity // All nodes are available

pq_full(q):
    return q.index_stack.size == 0 // There are no available nodes

pq_size(q):
    return capacity - q.index_stack.size // Total space - free space = space used

enqueue(q, Node n):
    if(queue_is_full):
        return False
    index = q.index_stack_pop()
    q.node_array[index] = n
    q.index_array.insert(index)
    return True
```

```
dequeue(q, Node n):
    if(queue_is_empty):
        return False
    index = q.index_array.remove(0)
    q.index_stack.push(index)
    n = q.node_array[index]
    return True

pq_print(q):
    for(i in range(pq_size(q)):
        index = q.index_array[i]
        print(q.node_array[index])
    return
```

Codes

From what I see from this, this ADT will be very similar to the BitVector ADT that we created for the previous assignment. This makes my job much easier as I will only need to make slight modifications to my code.

```
code_init():
    Code c = new Code()
    c.top = 0;
    for(bit b in c.bits):
        b = 0
    return c;

code_size(c):
    return c.top

code_empty(c):
    return code_size(c) == 0

code_full(c):
    return c.top == MAX_CODE_SIZE

code_push_bit(c, bit):
    if(code_full(c)):
        return False
    c.bits[top  / 8].bitArray[top % 8] = c
    top++
    return True

code_pop_bit(c, bit):
    if(code_empty(c)):
```

```
        return False
    bit = c.bits[top / 8].bitArray[top % 8]
    top--
    return True

code_print(c):
    for(i in range(c.top)):
        print(c.bits[i / 8].bitArray[i % 8])
    return
```

I/O

This section of psuedocode should be fairly straightforward as it is just simply reading and writing to a file descriptor.

```
read_bytes(infile, buffer, nbytes):
    int i = 0
    for(i in range(nbytes)):
        char c
        if(read(infile, c, 1) == 0)
            break
        buffer[i] = c
    return i

write_bytes(outfile, buffer, nbytes):
    int i = 0
    for(i in range(nbytes)):
        if(write(outfile, buffer[i], 1) == 0)
            break
        buffer[i] = c
    return i

read_bit(infile, bit):
    static buffer
    static index = -1
    index++
    if(index == 0):
        if(read_bytes(infile, buffer, BLOCK) == 0):
            return False
    bit = buffer.bitAt(index)
    index = (index + 1) % BLOCK
    return True

buffer
index = 0
```

```
write_code(outfile, Code c):
    while(!c.isEmpty()):
        buffer.set_bit(index, c.pop())
        index++
        if(buffer.size == BLOCK):
            write_bytes(outfile, buffer, BLOCK)
            buffer.reset
            index = 0
    return

flush_codes(outfile):
    if(index == 0):
        return
    for(i in range(index + 1, BLOCK)):
        buffer.set_bit(i, 0)
    write_bytes(outfile, buffer, BLOCK)
    buffer.reset
    index = 0
    return
```

Stacks

I will be using the same code that I have used for previous assignments for stack, with the only changes being that the data being held will be Nodes rather than uint32_ts. This will make almost no difference, save for the stack destructor function. This will require me to not only free the stack, but also to free each node within the stack. This is a trivial addition to the stack implementation that I have already written.

A Huffman Coding Module

This is the section that I am the shakiest about, and thus my design ideas are subject to large amounts of change

```
build_tree( histogram):
    PriorityQueue pq
    for(symbol s in histogram):
        node n = new node
        n.symbol = s
        n.frequency = s.frequency
        pq.enqueue(n)
    while(pq.size > 1):
        node left = pq.dequeue
        node right = pq.dequeue
        node new = node_join(left,right)
```

```
        pq.enqueue(new)
    return pq.dequeue()

build_codes(root, table):
    Code c = new Code
    counter = 0
    if(root.isLeaf()):
        table[root.symbol] = c.copy
        return
    if(root.hasLeft()):
        c.push(0)
        build_codes(root.left,table)
        c.pop()
    if(root.hasRight()):
        c.push(1)
        build_codes(root.right,table)
        c.pop()
    return

rebuild_tree(nbytes, tree_dump[]):
    stack s
    for(char c in tree_dump):
        if(c == 'L'):
            c = nextChar()
            s.push(new Node(c))
        if(c == 'I'):
            right= s.pop
            left= s.pop
            Node new = node_join(left,right)
            s.push(new)
    return s.pop

delete_tree(Node root):
    if(root.hasLeft()):
        delete_tree(root.left)
    if(root.hasRight()):
        delete_tree(root.right)
    if(root.isLeaf()):
        delete_node(root)
        root = null
    return
```

Encoder and Decoder

I will be generally be following the instructions on how to create the encoded and decoded files from the assignment PDF.

```
encode():
    parse_command_line()
    histogram = construct_histogram(infile)
    histogram[0]++
    histogram[255]++
    Node n = build_tree(histogram)
    Code table[]
    build_codes(n, table)
    construct_header()
    write_header()
    for(node in post_order_traversal(n)):
        if(node.leaf):
            write('L')
            write(node.symbol)
        else:
            write('I')
   for(symbol s in infile):
       write(table[s])
   flush_codes()
   close_files
   return

decode():
    parse_command_line()
    if(infile.magic_number != MAGIC):
        print_error_message
        return
    outfile.permissions = infile.permissions
    dump = read_dumped_tree()
    Node n = rebuild_tree(infile.tree_size, dump)
    node new = n
    for(bit b in rest of file):
        if(b == 0):
            new = new.left
            if(new.is_leaf):
                outfile.write(new.symbol)
                new = n
        if(b == 1):
            new = new.right
            if(new.is_leaf):
                outfile.write(new.symbol)
```

```
            new = n
close_files
return
```