Liam Murray

Professor Miller

CSE13s

November 25, 2022

Assignment 7: A Huffman Tree Grows in Santa Cruz

General Idea:

The general idea of the assignment is to create a compression and decompression algorithm, to compress data and save space. The implementation of this algorithm will use Huffman coding, in which a histogram of the frequency of each type of character in the file will be generated, a Huffman binary tree will be generated based on that data, and a binary code is assigned to each character based on the frequency with which they occur. That data will then be sent to the receiver which will then decode the data using the same tree information as the encoder.

The encoder and decoder portions of the Huffman tree will use a priority queue, along with the node data type. These two ADTs will be primarily used to implement the sorted tree, which allows for the codes to be constructed. The code constructor will use a stack to track the path through the tree, which directly translates into a binary code. The stack data type will also be present in the decoder, as a stack is used to reconstruct the Huffman Tree.

Pseudocode:

Node: The node is used to store data points in a way that is accessible. It’s implementation is very versatile, allowing it to be used in the priority queue as well as in the stack and tree implementations.

Implementation:

Structure definition, taken from the assignment document.

```
    Node:
        Node * left //pointer to left child
        Node * right //pointer to right child
        Uint8_t  symbol //Nodes symbol
        Uint64_t frequency //how often symbol appears
//node creator
node_create(symbol, frequency)
    Node pointer nd = allocate(size of (Node));
    nd->symbol = symbol
    nd-> frequency = frequency
    Return nd
//node delete
node_delete(**nd)
    free(*nd)
    Nd = NULL
//node join: joins two nodes and generates a parent node
Node * node_join(*left, *right);
    Sum = left->frequency + right-> frequency
    Node* parent = node_create($, sum)
    parent ->left = left
    parent->right = right
    Return parent
```

```
//node print
node_print(Node *nd)
    print(Node symbol, node frequency)
    print(Node left child)
    print(Node right child)
```

Priority Queue. This implementation of a priority queue will mimic a linked list, as the nodes with a lower frequency can be added to the front of the linked list easily. For this reason, much of the code seen here is taken from my assignment 6 code. The code includes auxiliary functions used to make coding the linked list much simpler.

```
    Pq structure definition:
    Struct PriorityQueue
        Int capacity = 0;
        int elements = 0;
        Node head;
        Node tail;
    pq_create(capacity)
        PriorityQueue * pq = allocate(sizeof(PriorityQueue))
        pq->capacity = capacity
        Head = node_create(NULL)
        Tail = node_create(NULL)
        Return pq
    pq_delete(**pq)
         Start = *pq->head
        for(i = *pq->elements, while i != 0, i - 1 each pass)
            Next = start->right
            node_delete(start)
            Start = next
        *list = NULL
    Return
//pq empty. Returns true if pq is empty, false if otherwise
pq_empty(pq)
    If pq->elements == 0
        Return true
    Else
        Return false
//pq full. Returns true if pq is full, false if otherwise
pq_full(pq)
    If pq->elements == pq->capacity
        Return true
    Else
        Return false
//pq size. Returns pq size
pq_size(pq)
    Return pq->elements
//enqueue. Puts a node in the queue, and return true if successful
pq_enqueue(pq, Node)
    If pq_full == true
        Return false
    Start = pq->head
    for(i = *pq->elements, while i != 0, i - 1 each pass)
            Next = start->right
            If ((start->frequency)  <  (node->frequency)  <  (next->frequency))
                start->right = node
                next->left = node
                node->left = start
                node->right = next
                Elements += 1
                Return true
            Start = next
//dequeue. Removes an element from the list and processes it
pq_dequeue(pq, **Node)
    If pq_empty(pq) == true
        Return false
    *Node = head->right
    head->right = head->right->right //make the head point to the node after the removed node
    head->right->right->left = head //make the node after the removed node point back to the head
    Elements -= 1
    Return true
//pq print. Prints the priority queue
pq_print(pq)
    Start = *pq->head
    for(i = *pq->elements, while i != 0, i - 1 each pass)
        Next = start->right
        node_print(start)
        Start = next
    Return
```