CSE13s Winter 2022
Assignment 6: Huffman Coding
Theodore Ikehara, tikehara@ucsc.edu
Due: March 2th at 11:59 pm
Design Doc (Draft)

1. Introduction:
   I. About:
   Compression of data is used every day in this world of computers. Everytime you watch a video online or download packages from the internet. All of this data is compressed when being sent to your computer to reduce the amount of data that the network needs to transfer to your computer. Compression also has changed the way data transfer works and the efficiency and speed of data transfer. This has allowed for more time efficiency of people downloading materials or transferring their data over the internet. Compression is a huge part of this all, as it has changed the fundamental approach to this issue of data transfer. The compression of data has been a hard problem, and remained an unsolved problem for many years, until David Huffman, a graduate student at MIT, came up with a new method in 1951. This method allowed for lossless data compression and changed the world of compression forever. This compression is what we will be attempting in this assignment.
   II. Lossless compression:
   In this assignment we will be attempting the Huffman's algorithm in lossless data compression. Lossless data compression is the way in which we will compress the data that leads to no data loss. How this way is achieved is through the creation of an internal shorthand that allows for redundant data to be lost. This will then create a set of the data and store the redundant data under one data point. And thus the data will end up taking less space as redundant data will no longer exist in the compressed file.
   III. Encoder:
   The encoder is one of the main parts of this assignment that we will have to implement. The encoder will read in an input and find the

Huffman encoding of the data and perform the compression using this Huffman encoding. The encoder will also take in a combination of command-line options. The encoder will count the number of occurrences of redundant data or unique symbols and construct a Huffman tree using the computed histogram and priority queue. Then we will traverse a Huffman tree performing a tree dump. Then we compute through each symbol and print its code to the output file.

IV. Decoder:

The second part is to reverse the actions completed in the encoder part of this assignment. We need to first read in a compressed file and reconstruct the Huffman tree. Then bit by bit reconstruct the data with nodes and linking the branches. We should then be able to read the original file from this.

2. The Implementation: This is the description of the ADT implementations

I. Nodes:

This is an important ADT that we will be using to construct the Huffman trees. Each node ADT will contain a pointer to its left child and a pointer to its right child. The nodes will contain all the data as raw bytes so we can find the redundant data. The nodes will also contain the frequency this is a key component to Huffman compression.

II. Priority Queues:

This is a major component for encoding portion of this assignment as in the encoder we will be creating a priority queue of nodes. A priority queue is just a normal queue with priorities set, and the queues with the highest priorities will be popped first. Here we may want to refer to the heapsort implementation. This may be helpful.

III. Codes:

This will be the portion of the assignment where we implement how the symbols of the original file will be stored and coded to reduce the amount of redundancy on the files that we are trying to compress. Here we will also make definitions of the codes that we will be setting the symbols to.

IV. I/O:
Here we will deal with all the reading and outputting files. We will read and write bytes using this ADT and will use the code ADT to write and read the codes from the files. This will be the main ADT that we will be using to pull the other ADTs together to write a final file and read initial files.

V. Stacks:
We will be implementing a stack of nodes for the decoder to reconstruct the Huffman tree. This is very similar in nature to the priority queue we used for the encoder portion of the assignment.

3. Eugene's section notes:
Notes for Eugene's Lab section 2/18/22: This is the main section used to complete this assignment.

4. Pseudo Code top level:
   I. Encoder: This is the implementation of the encoder, this portion performs the encoding compression process
      1. Take in command line options and
      2. Compute histogram
      3. Construct huffman tree with histogram
      4. Construct code tables
      5. Emit an encoding of the Huffman tree to a file
      6. Loop through all the symbols until file is done reading
   II. Decoder: This is the implementation of of the decoder, this portion performs the decoding decompression process
      1. Take in the command line options
      2. Read the emitted code from an infile and insert to stack of nodes
      3. Then reconstruct the huffman tree bit by bit
      4. Output the end file (this should reverse what we did in encoder)
   III. Nodes: This is the ADT for nodes implementation needed for priority queue
      1. Node create
         a. This is the constructor for the node ADT

    b. We will set the values to the ADT wide struct
    c. We will allocate space to the node
2. Node delete
    a. This will clear the dynamically allocated node
3. Node join
    a. This will join the node the nodes and the parent nodes symbol will be '$'

IV. Priority Queues: This is the implementation of priority queue of nodes this is a queue that sorts
1. Pq create
    a. Creates the priority queue
    b. Dynamically allocates the size of the queue
    c. Sets all the vars of the ADT
2. Pq delete
    a. This deletes the priority queue
3. Pq empty
    a. Checks if the head is at 0
4. Pq full
    a. Checks to see if the head is at max capacity
5. Pq size
    a. Returns the location of head
6. Pq enqueue
    a. Puts a node in position 0
    b. Increments the head
    c. Sorts the queued nodes
    d. Return true if everything was success
7. Pq dequeue
    a. Loads the value of the node at head
    b. Decrement head

V. Codes: This is the implementation of of the codes that encode and decode will make use of when "emitting the codes"
1. Code init
    a. Initialization of the codes
    b. Set all the codes to 0 first to initialize
2. Code size

    a. Return the top
3. Code empty
    a. Check if top is 0
4. Code full
    a. Check if top is max size
5. Code set bit
    a. Check if bit location exists
    b. If so set to 1
6. Code clr bit
    a. Checks if location exists
    b. If so set to 0
7. Code get bit
    a. If location is 1 return true
    b. Else return false
8. Code push bit
    a. Check if codes is full
    b. If not set top to input
    c. Increment top
9. Code pop bit
    a. Check if codes is empty
    b. If not take the top bit and output
    c. Decrement top

VI. I/O: This is the library that we will be using to read and write to files with bits

1. Read bytes
    a. Keeps track of the bytes that are being read
    b. Loop through infile until all the bytes are read
    c. Sum to the bytes read
2. Write bytes
    a. Keeps track of the bytes that are being written
    b. Loop through entire file until no more bytes to write
    c. Sum to the bytes written
3. Read bit
    a. Read in a block of bytes into a buffer
    b. Find the size for the bit to read

        c. This will keep looping until no more bits to read
    4. Write code
        a. Loops through the codes and writes to outfile
        b. Iterates until code is empty
    5. Flush code
        a. Sets everything back to 0
        b. If there are left over bytes delete them

VII. Stacks: This is the implementation of the stack ADT
    1. Stack create
        a. This is the constructor for the stack ADT
        b. Dynamically allocates memory to stack
        c. Sets the capacity of the stack
    2. Stack delete
        a. Deletes the stack
    3. Stack empty
        a. Checks if the head is at 0
    4. Stack full
        a. Checks if the head is at the capacity
    5. Stack size
        a. Checks the location of head
    6. Stack push
        a. Adds item to the stack
        b. Increments head
    7. Stack pop
        a. Takes out item from the stack
        b. Decrements head

VIII. Huffman Coding: This is the implementation of the Huffman coding module this is where the use of all the other ADTs will occur
    1. Build tree
        a. This constructs tree using pq
        b. Loop through the alphabet constant check using the histogram if that is part of the code
        c. If so enqueue it
        d. Then loop through the pq
        e. Dequeue the node and join return the root node

2. Build code
    a. Set code to 0
    b. If the node is a leaf node
    c. Then put the node symbol into the code
    d. If not then keep building codes until it is leaf
3. Dump tree
    a. If it is a leaf node assign ‘L’
    b. If not leaf node assign ‘I’
    c. And write these to the output file
4. Rebuild tree
    a. Reconstructs that the huffman tree
    b. Looks up the codes in the tree
    c. Assigns back the code
5. Delete tree
    a. Deletes the tree