Hersh Rudrawal
hrudrawa@ucsc.edu
5/12/2021

CSE13s Spring 2021
Assignment 6-Huffman Coding

DESCRIPTION

In this lab we will be creating a program that takes a file and compresses it using Huffman method. And also a Huffamn decoder to reconstruct the original file.

TOP-LEVEL DESIGN

Node

This is the implementation of the node. We will use nodes to create trees. Each node in the will represent one one of the possible ascii characters.

//given on assignment pdf
Typedef struct Node Node
Struct Node
    Node *left   //left child
    Node *right   //right child
    Uint8 symbol
    Uint64 frequency

*node_create(symbol,frequency)
    Use malloc to allocate memory for the Node
    If malloc failed, return a null pointer
    Set symbol and frequency to the appropriate parameters
    Set left and right child Nodes to null
    Return a pointer to the Node

node_delete(Node)
    Use free to free the Node
    Set the Node to a null pointer

*node_join(Node l, Node r)
    Create a new Node with symbol $, and set the frequency to the sum of the l and r frequencies
    Set the left child to l
    Set the right child to r
    Return pointer to the new Node

node_print(Node)
    Print parent node symbol and frequency

If the left and right child nodes are not null
Print their symbol and frequency

Priority Queue
This will be our implementation of the priority queue, it will take in nodes. It will order nodes from lowest frequency to highest(head to tail). It will be used to construct the Huffman tree.

```
//base off queue from assign 3
// and inspired by Eugene's section
Struct PriorityQueue
    Uint32 head    //position of head
    Uint32 tail    //position of tail
    Uint32 size    //size of queue
    Uint32 capacity
    Node **nodes //array of pointer to nodes
```

*pq_create(capacity)
Use malloc to allocate space for pq
Set head, tail, size to 0
Set capacity to capacity
Use calloc to create **nodes- parameters(capacity,sizeof(Node))
If *nodes could not be created, free the pq and return null pointer
Else return pointer to pq

**pq_delete(PriorityQueue)
Free nodes
Free PQ //Priority Queue
Set PQ null pointer

pq_empty(PQ)
Return true if the size of PQ is 0, else return false

pq_full(PQ)
Return true if the size of PQ equals capacity, else false

pq_size(PQ)
Return the size of PQ

When we enqueue in a priority queue, the nodes should be ordered by their frequency, with the highest being at the tail, and the lowest at the head.

enqueue(PQ,Node n)
- Return false if PQ is full
- If PQ is empty
  - Add the Node to *nodes at position tail
- Else
  - Create a loop to iterate the nodes from tail to head- let “i” be the value of the loop
    - If Node n’s frequency is greater than the node in PQ at the position i-1 or if i equals head
      - add n to position i and break
    - If not, copy the node at position i-1 to position i and continue looping
- Increment size and tail
- Return true

dequeue(PQ,Node x)
- Return false if PQ is empty
- Set x to the Node in *nodes at position head
- Decrement PQ size
- Set head to- (head+1)%capacity //ensure the head wraps around
- Return true

pq_print(PQ)
- Iterate through PQ and print nodes

Defines
//provided on assignment pdf
BLOCK = 4096
ALPHABET = 265
MAGIC = 0xDEADBEEF
MAX_CODE_SIZE = ALPHABET / 8
MAX_TREE_SIZE = 3 * ALPHABET - 1

Codes
This will be our implementation of codes. It will be used to turn symbols in a Huffman tree into codes

Typedef struct Code
	Uint32 top
	Uint8 bits[MAX_CODE_SIZE]

code_init()
	Set top to 0
	Set all entries in bits to 0

code_size(Code)
	Return top

code_empty(Code)
	Return true if top equals 0, else false

code_full(Code)
	Return true if top equals Alphabet else false

code_push_bit(Code, bit)
	If Code is full return false
	Find the where in bits to insert the new bit
	Bits index = top/8
	Bit position= top%8

	If bit is 1, set the appropriate bit in bits
	If bit is 0, clear the appropriate bit in bits
	Increment top by 1
	Return true

code_pop_bit(Code, bit)

	Return false if Code is empty

Decrement top by 1
Get the top bit in bits
Set bit to that value
Return true

code_print(Code)
Use for loop to print bits

I/O
This program will be used by our encoder and decoder to read and write to files

//inspired from eugene's section
This function will use the read() function to read bytes off a file, however, if read() did not get the number of bytes required we will loop it until we get the desired number of bits, or read the end of the file. We will return the number of bits we read at the end.

read_bytes(infile, *buf, nbytes)
Create int bytes to store the number of bytes we read
Create while loop (bytes<nbytes)
Call read() with parameters (infile, buf, nbytes-bytes)
If an error occurs or we read no more byte, break out of the loop
Else add the number of bytes read to variable bytes
Add the number of bytes to the extern var bytes_read
Return bytes

//similar to read bytes
write_bytes(outfile, *buf, nbytes)
Create int bytes to store the number of bytes we wrote
Create while loop (bytes<nbytes)
Call write() with parameters (outfile, buf, nbytes-bytes)
If an error occurs or we can not write more byte, break from loop
Else add the number of bytes written to variable bytes
Add the number of bytes to the extern var bytes_written
Return bytes

```
read_bit(infile,*bit)
        Create a static buffer of size BLOCK
        Create static index, set to 0 // number of bts read
        If the buffer is empty, index=0, fill it by calling read_bytes

        Find the position of the top bit
        buffer index = index/8
        buffer bit position= index%8
        Create an int x and set it to 1
        Conduct a left shift on x by the value bit position
        Conduct a AND operation on x with buffer[buffer index]
        Shift x back and set bit to x
        Increment index
        Return true

//writes codes to a file
Create a static BUFFER of size BLOCK
Create static INDEX index of bit need to read
write_code(outfile, Code)
        Create for loop to Iterate the code, i < code size
                Check if the buffer is full
                        Write the buffer to the outfile
                        Reset index to 0
                Create an int x and set it to 1
                Conduct a left shift on x by the value of the index
                Create for loop to iterate Code
                        Get and store bit at position i
                        If the bit is a 1
                                Set the bit at index i
                        Else
                                Clear the bit at index i
                        Increment index by 1

//write any remaining codes
flush_codes(outfile)
        Check if INDEX equal  0 do nothing
        else Iterate from the INDEX to the end of the BUFFER
                Set all bits to 0

        Write the buffer to the outfile, bytes= (index-1)/8+1
        Set INDEX to 0
```

Stacks
Will be used by the decoder to reconstruct a Huffman tree

//based off stack from assignment 3/4
Struct Stack
    Uint32 top //top of stack
    Uint32 capacity //capacity of stack
    Nodes **items //stack will contain pointers to nodes

Stack stack_create(capacity)
    Create stack using malloc
    Set top to 0;
    Set capacity to the new capacity
    Use calloc to allocate memory for items
    If we could not allocate memory, free the stack and return a null pointer
    Return pointer to Stack

Stack stack_delete(Stack)
    Free *items
    Free the Stack
    Set pointer to NULL

Stack_empty(Stack)
    Return true if var top in the Stack is 0

stack_full(Stack)
    Return true if var top in the Stack is equal to the capacity

stack_size(Stack)
    Return var top in stack

Stack push(Stack, Node){
    If the Stack Is full, return false
    Add the Node to *items at next free position(top)
    Increment var top in the Stack
    Return true

stack_size(Stack){
    Return var t in Stack

stack_pop(Stack, Node){
    If stack is empty return false

```
        Set Node to the item at position top in *items
        Decrement top by 1
        Return true

stack _print(stack){
        Create for loop with i= top,i>=0{
        Print out the item in the stack positioned at i
```

Huffman Coding Module
This program will will contain functions to build a tree, build codes, rebuild trees and delta trees

```
//This function read the histogram array, and createes nodes out of all non zero symbols. These
//nodes then enter a priority queue. Then we start joining nodes together to create a tree
*build_tree(Array[ALPHABET])
        Create a priority queue- “PQ”
        Use for loop to iterate histogram/Array
                Create a node for all non zero positions
                Add node to PQ

        //Then we deque 2 nodes, join them together and add them back into the queue until 1
        //node remains
        While PQ has 2 or more nodes
                Deque 2 nodes
                Join them together with node_join
                Add the new node to PQ
        Pop the root Node from PQ, delete PQ and return the root

//This function will traverse the tree and construct codes for each symbol. When it traverses the
//left side of the tree, a 0 gets added to the code. When it traverses the right side, a 1 gets
//added to the code. When the function reaches a leaf, it adds the code to the code array- stores
//codes for all symbols
build_code(Node,Code table[ALPHABET])
        Create static Code c using code_init() //code for symbol
        Check if Node is not null
                Check if Node is leaf //left if has no children
                        If node is leaf, add c to the table- table[symbol]

                //traverse the left side first, marked with bit 0
                Else call build_code on left Node and push bit 0 to c
                Pop top bit from c

                //remove most recent bit and now traverse right side- marked with bit 1
                Call build code on right Node add push bit 1 to c
                Pop top bit from c
```

```
//rebuilds a huffman tree from reading the tree_dump array. This function uses stacks to store
//nodes and join them together to for the tree
*rebuild_tree(nbytes, tree_dump[nbytes])
        Create new stack
        Create for loop to iterate tree_dump, i<nbytes
                If element i in tree dump is an “L”,
                        get the next element and create a node out of element
                        Add node to stack
                        Increment i
                If element is “I”
                        Pop from the stack the stack to get right child
                        Pop from stack to get left child
                        Join left and right child nodes and push parent node to stack
        Once done looping, pop the last element from stack and return it

//delete the nodes of the tree using post order traversal
delete_tree(Node)
        Check if Node is not empty
                Check if node does not have children
                        If so delete the node
                Else call delete tree on left node
                Call delete tree on right node


Header
//will contain information about the original uncompressed file
//provided on lab document
Typedef struct Header
        Uint32 magic
        Uint16 permissions //input file permissions
        Uint16 tree_size // emitted tree size in bytes
        Uint64 file_size
```

Encoder

This program will read a file and find the Huffman encoding to compress the files. This program takes the following options: -h help, -v (print out statistics- compressed/uncompressed file size and space saved) -i infile(path of the file containing data), and -o outfile(file to write the compression into).

To get the options we can use get opt and a switch statement

```
main(argc,argv)
    Create and set FILE infile to stdin
    Create and set FILE putfile to stdout
    Create and set int opt to 0 //stores getopt
    While get opt does not return -1
        Create switch opt
        Case -h
            Print help message
        Case -v
            Set bool stats to true
        Case -i
            Check and set infile
        Case -o
            Check and set outfile
        Default
            Error message
```

Next, we need to read the input file 1 byte at a time and create a histogram. we can create an array of 256 items, each index representing an ASCII value, and increment an index when we read the appropriate character. Then we put the values in the array into a priority queue

```
    Create uint64 array of size ALPHABET called - “histogram”
    Increment element 0 and 255 by one
    Create a temp file
    Loop through the infile one byte at a time
        Get the byte and store it in variable “x”
        Increment the array at position x
        Write the infile to the temp file

    Call build_tree() function and store new Node in “root”
    Create code table out of root
    Create a Header called header
    Set magic to macro MAGIC
    Use fstat on infile to set header permissions
    Use fchmod to set outfile permissions
```

Set header's tree_size to - 3*unique symbols -1
Use fstat to set header file_size
Write the header to the outfile using write bytes

Perform post-order traversal on root and write the tree dump to outfile
Create a function for post-order traversal below

Read all symbols from the temp file, and write to the outfile the appropriate code with write_code().
Flush any remaining codes
Print stats if needed
Close file and delete tree and temp file

```
//tree dump for tree
post_order(Node,outfile)
    Check if node is not empty
        Check if node is leaf
            Write "L" to outfile //write L to indicate leaf
            Write symbol of node to outfile
        Else call post_order of left child node
        Call post_order of right child node
        Write "I" to outfile //write I to indicate interior node
```

Decoder
This program will read a compressed file and decode it. This program takes the following options: -h help, -v (print out statistics- compressed/decompressed file size and space saved) -i infile(path of the file containing data), and -o outfile(file to write the compression into).
To get the options we can use get opt and a switch statement
//inspired by eugene's section
main(argc,argv)

Create and set FILE infile to stdin
Create and set FILE putfile to stdout
Create and set int opt to 0 //stores getopt
While get opt does not return -1
Create switch opt
Case -h
Print help message
Case -v
Set bool stats to true
Case -i
Check and set infile
Case -o
Check and set outfile
Default
Error message

Create a temp file
Read the infile and write it out to the temp file
Read the header from the temp, ensure the magic number is correct else return error
set the permissions of the outfile, from the header
Read and store the tree dump into an array
Call reconstruct_tree on tree dump and store new Node in “root”

Create a new node curr, and set equal to root
While the number of bytes written is less than file_size
Check if curr is not null
If curr is a leaf write its symbol to the outfile and set curr to root again
else
Use read_bit() to read from the temp file
If bit = 0, set curr to its left node
If bit = 1, set curr to its right node

Delete tree and temp file
Close infile and outfile

CHANGES

- After Eugene's section I changed how my decoder writes to the outifle. Before, I used a recursive function.
- Also changed how i use the infile, instead of reading from it directly, i copy it to a temp file and read from that.