DESIGN.pdf
Assignment 7
Despina Patronas

Trie implementation

using code.h defined macros (ALPHABET = 256) (STOP_CODE = 0) (EMPTY_CODE = 1) (START_CODE = 2)

<u>Define struct Trienode</u>

Trie struct:

Trienode children[ALPHABET] *//each trie node is initialized with 256 children aka: alphabet*

Code *// the data held within the trienode*

End Trie struct

<u>Define creating the trie:</u>

Creating a trie starts with allocating a trie node initially.
Start the trie node with a root which is a trie node created with EMPTY_CODE index

TrieNode trie_create (void)

Return trie_node_create(EMPTY_CODE)

End trie_create

<u>Define creating trie_node function</u>

The creation of the trie_node itself allocated memory for the trienode and set space to NULL
There is no allocation of the children, however children will each be set to NULL
set the code field as the index
*Setting the indexed child to NULL to keep the memory clean

TrieNode trie_node(index)

Allocate data for the newnode

Iterate from I: [0, ALPHABET]

newnode ->children[i] = NULL
Increment i

end iteration

newnode ->code = index

return newnode

end trie_node

Define Trie_step function

Trie node step will querie a node in the tree for the symbol.
If the symbol is found return the ptr to the trinode child which the symbol was found at.
If the symbol is not found, return null.
If the node has no children return null
*always check if parameter ptr exists first

*TrieNode trie_step (*node, sym)*

If (node->children[sym] == NULL)

Return NULL

Return node->children[sym]

End TrieNode trie_step

Define Free(ptr) Function

_______Function to free ptr
Also set mem to NULL

Define trie node destructor function

_ ______delete the node that was allocated and set the memory to null
_______using the Free() function

Trie_Node_delete (*node)

Free(node)

End Trie_node_delete

<u>Define trie tree destructor function</u>

_______Trie delete will clear all the children from the bottom up.
This is done recursively by calling trie_delete on the children of the node (while there is children call)
And setting all of the children null
*Setting the indexed child to NULL to keep the memory clean
_______*always check if parameter ptr exists first

*Trie_delete (*node)*

Iterate from i: [0, ALPHABET]

Trie_delete(node->children[i])

Node->children[i] = NULL

Increment i

End iteration

Trie_node_delete (node)

End trie_delete

<u>Define trie reset function</u>

_______Trie reset will retain the root while clearing the children from the bottom up
This is done by using the recursive trie_tree delete function on the children of the root
*Setting the indexed child to NULL to keep the memory clean
_______*always check if parameter ptr exists first

*Trie_reset(*root)*

Iterate from I: [0, ALPHABET]

Trie_delete (root->children[i])

Root->children[i] = NULL

Increment i

End iteration

End trie_reset

End Trie implementation

Start Word Implementation

using code.h defined macros (ALPHABET = 256) (STOP_CODE = 0) (EMPTY_CODE = 1) (START_CODE = 2)

<u>Define struct Word</u>

Word contains two fields

** syms* *//holds the symbols for the word (as a byte array)*

length *//holds the length of the words aka num of syms*

End struct word

<u>Define word_creation</u>

_______allocate space for the word and the symbol array
set the fields of the new word
word passed in = symbols which will be set individually into the symbol array
_______*always check if parameter ptr exists first

*word_create (*syms, length)*

Allocate memory for new_word

Allocate memory for new_word syms

Set new_word length and iteratively set the new_word syms at each index: [0,len)

Return new_word

End word_create

<u>Define Word_append_sym function</u>

_______make a new word
if word length is 0, then create a new word with just the symbol and length 1
otherwise word length is old word length +1 and syms + appended symbol
_______*always check if parameter ptr exists first

*Word *Word_append_sym (*old_word, * sym)*

Allocate memory for new_word

Set new_word->len = old_word->len + 1

Allocate memory for new_word syms

Iteratively set the new_word->syms using old_word->syms at each index: [0,oldword->len)

Add the final symbol at the end of new_word (the one passed into fnx)

Return the new_word

End word_append_sym

Define Word_delete function

- ______free the word passed into the function including its fields.

______Set the memory to NULL

*Void word_delete (*word)*

Free the word->syms

Set the word->syms to NULL

Free (word)

End word_delete

Define Wt_create function

______word table is an array of words

to create the word table allocate memory for it using MAX_CODE

______initialize wt index with a word to start. This word has a length of 0 and NULL symbols to start

*WordTable *Wt_create (void)*

Allocate wt

Set the wt[0] = word_create (NULL, 0)

Return wt

End wt_create

Define wt_reset

______reset the word table by setting all the indexes to NULL

______iterate through index: [START_CODE, MAX_CODE) incrementing index by 1

Define wt_delete

call wt_reset on wt to delete

free(wt)

End Word Implementation

Shout out to Oly and Eugene for helping understand the read_bytes, file header and flow of the read_pair / buffer_pair. Thank you TAs for your time and dedication to us throughout the quarter.

IO Implementation

Define BLOCK = 4096 //4KB

Extern variables to keep track of stats use static to keep the:
uncompressed bits[block]
compressed bits[block]

Static buffer arrays to store symbols and bits (one for encode and one for decode)

<u>Define read_bytes</u>

______read bytes will read from infile. Within the loop, keep track of how much bytes is specified to read and how much total bytes has been read. Keep reading bytes and incrementing total as long as total doesn't surpass the "to read" limit. Make sure read bytes is a positive number. Return the number of ______bytes read. Keeps looping until all "reading" bytes are read

*Read_bytes(infile, *buffer, reading)*

Read_bytes = 0

Total_read = 0

Do:

Read_bytes= read(infile, (buffer+total_readl), (reading – total_read)

Total_read = total_read + read_bytes

While:

Read_bytes > 0 && total_read != reading

Return Total_read

End read_bytes

<u>Define write_bytes</u>

______write bytes is exactly the same a read_bytes but using the write() syscall and an outfile exact same implementation basically

Define read_header

Reads the header bits aka the magic number

*Read_header(intfile, Fileheader*header)*

Increment the compressed bits *//8 bits * sizeof Fileheader*

Read_bytes (infile, header, sizeof(Fileheader) *//needs to be casted*

End read+header

Define write_header

Write header replaces read_bytes with write_bytes(outfile, ….)

Define Read_sym

Reads symbols from input file into a buffer

*Read_sym (infile, *sym)*

Static int remaining_sym = 0 *//track the symbols read*

If (remaining_sym == 0) *//if empty fill it with infile*

Remaining_sym = read_bytes(infile, sym_buff, BLOCK)

//check to see if theres symbols left

If (!remaining_sym) *//we reached the end*

Return false

End if

End if

**sym = sym_buff[sym_index]* *//set the curr sym from the buffer index*

Sym_index = (sym_index +1) MOD BLOCK *//set the curr sym index*

Incrememnt the compressed bits by 8 *//8 bytes*

Remaining_syms --; *//decrement the symbols as they are set*

Return true *// Iterate through the whole buffer*

End read_sym

Define buffer_pair

buffer pair will take the input CODE buffered first and the buffer symbol. Note this is done in LSB so flipping is required. When buffer fills up write out the buffer and reset to read > 4kb otherwise only one block will be read☹

```
Buffer_pair( outfile, code, sym, bitlen)

    Iterate from I [0,bitlen)

        If ( BLOCK == bitindex / 8 )            //check if the buffer is full

            Flush_pairs(outfile)                //if its full reset first then continue

        //continue as normal

        Byte_index = bitindex/8                     //to access the bit

        Norm_bit_index = bitindex MOD 8             //to access the correct bit

        Int Get_bit = code & 1

        Code = code >> 1                            //shift for the next iteration

        Bitbuff[byte_index]|= get_bit << norm_bit_index     //add code to buffer

    End iteration
    //repeat the code iteration for the symbol

    //since we know symbol is 8 bytes always, we only traverse loop 8 times

    //everything else is exactly the same but substitute code for sym

End buffer_pair
```

Define flush pairs

flush pairs will flush the bitbuffer when it reaches capacity and reset the index to 0

```
Flush_pairs(outfile)

    Total = 0                   //figure out how much to incremement compressed bits

    If (bitindex != 0)                          //0 means theres nothing to flush

        Total = write_bytes(outfile, bitbuff, to_bytes(bitindex)        // byte length

        Reset bitindex to 0
```

*Increment compressed_bits by itself + (total * 8)   //bytes*

End flush_pairs

<u>Define read_pair</u>

basically undo buffer_pair it's the opposite for decode make sure to initialize code and sym when starting to 0. Similar to buffer_pair but in reverse order. SIMPLE. Also must check if stopcode was found to discontinue reading the pairs

Read_pair(infile, *code, *sym, bit_len)

*****Code = 0, *sym = 0

Check if bitindex is 0

if it is read_bytes into bitbuff

Doing so will cause compressed bits to increment

Iterate from i [0,bit_len) incrementing bitindex

Get the byte index (bitindex / 8)

Get the correct bit (bitindex MOD 8)

Get_bit = ((bitbuff[byte_index] >> correct bit) & 1)

Code |= get_bit << i

End iteration

//check the stop code if code == stopcode return false

//repeat the code iteration for the symbol

//since we know symbol is 8 bytes always, we only traverse loop 8 times

//everything else is exactly the same but substitute code for sym

Return true End

read_pair

//buffers for the word

Static word_buffer[BLOCK]
Static wordindex

<u>Define buffer_word</u>

______buffer word will flush words in that if index reaches the limit it will flush its contents to outfile and reset its index.
Buffers populates from buffer_sym into word_buffer

```
Buffer_word(outfile, *w)

    Iterate from I: [0, w->length) i++

        Word_buffer[wordindex] = w->syms[i]

        Word_index++

        If (word_index == BLOCK)

            Flush_words(outfile)

        End if

    End iteration

End buffer_word
```

<u>Flush_words</u>

______Similar to flush_pairs it is its 'decoding' polar opposite.
exact same implementation as flush_pairs but with word buffer instead

<u>To_bytes</u>

______takes in a # of bits and spits out its floored bytes equivalent

```
To_bytes(bits)

If bits MOD 8 == 0

    Return bits/8

Otherwise

    return bits/8 +1

end to_bytes
```

Shout out to Oly and Eugene for helping understand file protection and file headers. Implementation based on your recommendations. Thank you TAs for your time and dedication to us throughout the quarter.

Main implementation DECODE

Define getopt options

Main

Define infile = STDIN_FILENO

Define outfile = STDOUT_FILENO

Getopt loop

Switch case for argc

-v bool toggles to turn on the stats output

-I changes input default form stdin to optgarg don't forget O_RDONLY

-o stdin to optarg don't forget O_WRONLOY | O_CREAT| O_TRUNC

End getopt loop

Define fileheader hd set fields to 0

Read_header(infile, &fileheader)

Check the magic number is set properly

Fschmod(outfile, hd.protection) //set outfile with same permissions as infile

.//commence DECOMPRESSION pseudocode for decode based on lab manual

Print stats if applicable

//clean up the nasty allocs wt and infile / outfile

Main implementation DECODE

Exactly the same but with fstat function to set the permissions

And of course the compression algorithm for encoding

Don't forget to clean up memory