ASGN 7 - COMPRESSION

LEMPEL-ZIV COMPRESSION

- REPRESENT REPEATED PATTERNS BY 32 BIT PAIRS
  - 16 BITS FOR CODE, 8 BITS FOR VALUE
  - CODE = UNSIGNED INT
  - $2^{16}-1$ CODES: RESERVE 1 FOR STOP CODE VAL $2^{16}-1$
  - EMPTY_CODE = 1 START_CODE = 2
  - NEW "WORDS" AKA CHARS ARE SET AS INDEPENDENT BRANCHES
  - CODES ARE ASSIGNED BASED ON UNIQUENESS ONLY, AND ARE USED AS KEYS/INDECIES FOR WORD DICT/LIST
    - WHEN A "WORD" ISN'T IN THE DICTIONARY, IT IS ASSIGNED CODE: (prev_code + 1) AND IS THEN ADDED TO THE DICTIONARY
  - PAIRS ARE OUTPUTTED BY THE CODE OF THE PREVIOUS WORD, AND THE CHARACTER ITSELF WHICH WILL BE APPENDED TO THE CHAR THAT THE CODE REPRESENTS
- LZ 78 COMPRESSION NOT INCREDIBLY EFFICIENT IN FILES WITH LITTLE PATTERNS →
  - BINARY.DECODE COMPRESSES WITH A NEGATIVE RATIO

TRIES

- AKA PREFIX TREE

- 256 POSSIBLE CHILDREN PER NODE (ASCII VALS/CHARACTERS)

```
struct TrieNode {
  TrieNode * children[ALPHABET];
  uint16_t code;
}
```

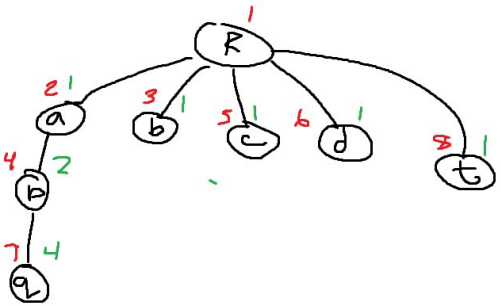
TrieNode *trie_node_create(uint16_t code):
- TrieNode *t = (TrieNode*) malloc(sizeof(TrieNode))
- t->children = (TrieNode*) malloc(ALPH * sizeof(Tri...)
  - SET ALL INDECIES OF T-> CHILDREN TO NULL
- t->code = code
- return t.

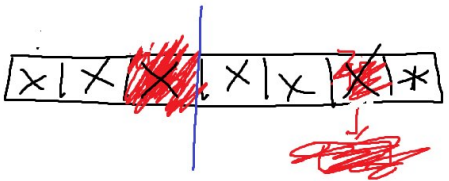
void trie_node_delete(TrieNode *n):
- free(n)

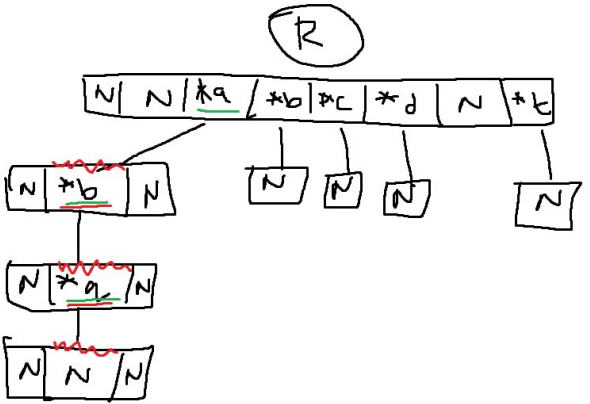
TrieNode *trie_create(void):
- TrieNode *t = trie_node_create(START_CODE)
- return t

void trie_reset(TrieNode *root):
- LOOP THROUGH TRIE FROM ROOT'S CHILDREN
- DELETE EACH CHILD AND ALL OF THEIR CHILDREN
  - USING trie_delete

void trie_delete(TrieNode *n):
- LOOP THROUGH ALL CHILDREN OF NODE
  - RECURSIVE CALL — DELETE ALL OF CHILD'S CHILDREN
  - SET THAT LOCATION IN CHILDREN TO NULL
- DELETE ITSELF


R
1
a
2 1
b
3 1
c
5 1
d
6 1
t
8 1
b
4 2
q
7 4


X X X X X *


R
N N *a *b *c *d N *t
N *b N
N *q N
N N N
N N N N N

abab c dab q t

WORD TABLES

- LOOKUP TABLE WHERE EACH INDEX CONTAINS A WORD

- WORDS HAVE SYMBOLS, AKA THE WORD ITSELF, STORED IN A BYTE ARRAY
  - WORDS ALSO HAVE A LENGTH THAT IS USED TO DETERMINE THE LENGTH OF THE BYTE ARRAY

Word * word_create (uint8_t * syms, uint64_t len):
- word * w = malloc ...
- w -> syms = malloc (len x uint8)
- Loop through and set w->syms = syms @ [i]
- w -> len = len

word * word_append_sym (word * w, uint8_t sym).
- word * new = malloc()...
- new -> len = (w -> len + 1)

- new -> syms = malloc (new->len * uint8)
- memcpy (new -> syms + w -> syms, w -> len)

- new -> syms [new->len] = sym

void word_delete (word * w):
- free (w -> syms)
- free (w)

WordTable * wt_create (void):
- WordTable * wt = calloc (MAX-CODE, sizeof(word))
- WORD * start = word_create (NULL, ZERO)
- wt[EMPTY_CODE] = start.
- return wt.

void wt_reset (WordTable * wt):
- loop from i = START_CODE -> MAX-CODE
  - if wt[i] ≠ NULL
    - word_delete (wt[i])
    - wt[i] = NULL

void wt_delete (WordTable * wt):
- for (i=0 -> MAX-CODE
  - if wt[i] ≠ NULL
    - word_delete (wt[i])
    - wt[i] = NULL

IO.H

// bit buf contains pairs
// sym buf contains pure translation of file

// READS SPECIFIED # OF BYTES (OR UNTIL EXHAUSTED) FROM SOURCE AND STORES IN BUFFER
int read_bytes(int infile, uint8 *buf, int to_read):

- int read_count, int total, = 0
- DO:
  read_count = read(infile, buf+total, to_read-total)
  total += read_count
  while(rbytes > 0)

(MOVE POINTER IN BUFFER; ACCOUNT FOR BYTES ALREADY READ)

- return total

// WRITES BYTES INTO BUFFER
int write_bytes int outfile, uint8 *buf, int to_write):

- SAME AS ABOVE JUST WITH write(outfile, buf+total, to_read-total)

// READS HEADER FROM FILE INTO OUR STRUCT
void read_header(int infile, FileHeader *header):

- read_bytes(infile, (uint8*)header, sizeof(FileHeader))

// WRITES FILE HEADER TO OUTPUT FILE
void write_header(int outfile, FileHeader *header):

- write_bytes(outfile, (uint8*)header, sizeof(FileHeader))

// RETURNS BY REFERENCE A SINGLE SYMBOL FROM UNPROCESSED FILE
// IF FIRST TIME CALLED, READS FILE INTO BUFFER, OTHERWISE ONLY UPDATES SYM + CHECKS END
bool read_sym(int infile, uint8_t *sym):

- static int end = 0
- if first time called: end = read_bytes(infile, buf.sym, BLOCK)
- update *sym / *sym = buf.sym[index++] // updates for next call
- if reached last index of buffer reset index + overwrite buffer
- if read in BLOCK bytes, maybe more to read, return true
  - else if index == # bytes read (end) return false
    - else return true

// STORES PAIR OF SYM AND BIT ADJUSTED CODE IN BUFFER, CODE FIRST, SYM NEXT
// BUFFER IS WRITTEN OUT TO OUTFILE IF BUFFER IS FULL, ELSE RELY ON FLUSH_PAIRS
void buffer_pair(int outfile, uint16 code, uint8 sym, u8 bitlen):

- total bits += 8 + bitlen // USED FOR STATS -V

// BUFFER IDEX/CODE/BITS

for(i=0, i < bitlen, i++):

- IF LSB == 1:
  - setbit(bitbuf, bitindex) // BIT = 1
- else (LSB == 0):
  - clrbit(bitbuf, bitindex) // BIT = 0

- bitindex ++ // SET BITLEN BITS, THEN MOVE TO SYM
- CODE >>= 1

- if bitindex == BLOCK * 8:
  - write_bytes(outfile, bitbuf, BLOCK)
  - bitindex = 0

// BUFFER SYM

- same as above, just replace bitlen with 8, CODE w/ sym

// WRITES OUT ANY REMAINING PAIRS TO OUTFILE
void flush_pairs(int outfile):

- if bitindex != 0:

  if (bitindex % 8 == 0):
    bytes = bitindex / 8
  else:
    bytes = bitindex / 8 + 1
- write_bytes(outfile, bit_buf, bytes)

// WRITE OUT REMAING SYMBOLS OF WORDS
void flush_words(int outfile):

REPRESENTS BYTES NO NEED TO TRANSLATE

- if symindex != 0:

  - write_bytes(outfile, symbuf, symindex)

// READS CODE AND SYMBOL FROM INPUT FILE -> PLACES OBJS IN RESPECTIVE *
// PROCESSES BLOCKS LIKE buffer_pair, CODE THEN SYMBOL
// RETURNS TRUE IF MORE TO READ IN BUF, ELSE FALSE

```
bool read_pair(int infile, u16* code, u8* sym, u8 bitlen):

  // READS CODE
  ^*code = 0

  - for(i=0, i<bitlen, i++):

    - if bitindex == 0 : read_bytes(infile, bitbuf, BLOCK)

    -if bitbuf[bitindex] == 1 :
        - set bit(*code, i)
    - else
        -clr bit(*code, i)

    -bitindex ++

    - if bitindex == BLOCK * 8 : bitindex = 0

  // READ SYM

    - same as above, replace bitlen w/ 8, code w/ sym

  // FINAL RETURN
   - return !(*code == STOP_CODE)


// FILLS BUFFER WITH SYMBOLS OF WORD
// WRITES TO OUTFILE IF FINISHED
void buffer_word(int outfile, Word *w):

  - total syms += w->len   // COUNT STATS

  - for(i=0, i<w->len, i++):

    - symbuf[symindex++] = w->syms[i]

    -if sym index == BLOCK
        - write_bytes(outfile, symbuf, BLOCK)
        - sym index = 0
```

COMPRESSION (FROM LAB PSUEDO CODE)

- getopt() loop -> taking user I/O IF THERE
- create header for encoded file using permissions from infile and given magic # 0x8badbeef
- initialize TrieNode root
  - initialize prev, curr, next values for reference
- while (reading syms from infile):
  - try to add sym to trie
    - add if new on specific branch -> back to root
    - if already present on branch make that node current
    - reset trie if it fills
- write out any remaining pairs
- produces LZ78 encode file that can be decoded with associated decode function if magic numbers are the same.

DECOMPRESSION

- getopt() -> look for file inputs/output
- read in header and ensure magic # is as expected
- initialize word table and necessary reference vars
- while (reading pairs from encoded file):
  - append word table with sym and code read in form of word
  - place word created into buffer to be written out
  - reset word table if ran out of space
- flush any remaining words

- FOR BOTH ENCODE + DECODE PROVIDE STATS FROM LAB MANUAL

CITATIONS

- Eugene's lab section 12/9
- Maxwell's lab section 12/9
- various piazza post on valgrind, infer, and some ADTs
- man7.org for stat struct
- linux.die.net for fchmod C)
- pubs.opengroup.org for fstat C)

EXTRA NOTES

- mane buffers static in io.C
- 1 in buffer one out buffer
- set trie locations to null after deletions
- open files outside of getopt for infer errors

- THANK YOU TA's! ☺