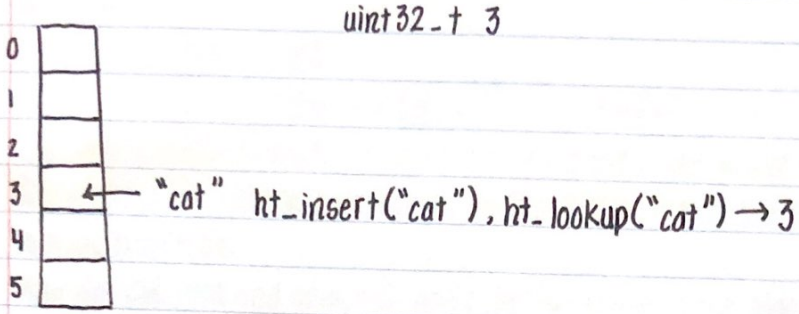
Assignment 6: Lab Section Eugene 2/16/2021

Bloom Filter: an array of bits, utilizes an underlying bit vector
Let's take an example: hash(salt, "cat")

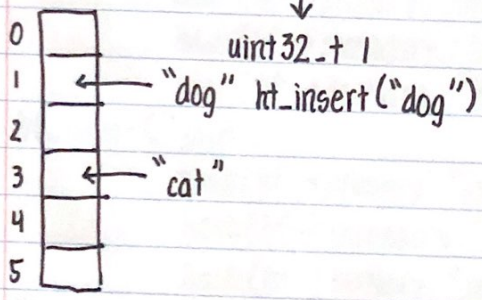
↓

uint32_t 3



Next, let's do hash(salt, "dog")

↓

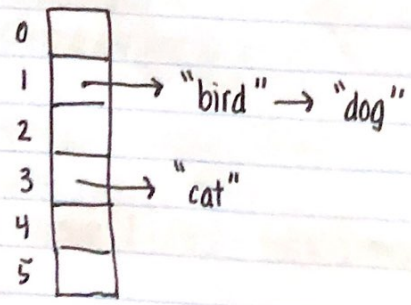


But, let's say we did hash(salt, "bird")

↓

uint32_t 1

This is what's known as a hash collision since "dog" is already @ the index
To solve this, we utilize a linked list
In the bloom filter, there will be 3 different salts, 3 different mappings
In the hashtable, there will be an array of linked lists
so now,



Each node in the linked list will contain 2 strings, oldspeak and newspeak
It also contains pointers to the next and previous node (implements a doubly linked list)
Each linked list will be initialized with two sentinel nodes (2 dummy nodes that exist when initialized)
ll_create()

LINKED LIST

NULL ← (H) ⇄ (T) → NULL

Inserting a node will mean to insert it at the head of the linked list
The program will read in badspeak words from badspeak.txt, add it to the bloom filter and the hash table
Add the oldspeak and newspeak pairs are also added to the Bloom Filter and hash table
bf_insert("meme")

hash(bf→primary, "meme") → 0
hash(bf→secondary, "meme") → 3
hash(bf→tertiary, "meme") → 5

bf_insert("java")

hash(bf→primary, "java") → 1
hash(bf→secondary, "java") → 2
hash(bf→tertiary, "java") → 5

Bloom Filter

0	1
1	1
2	1
3	1
4	0
5	1

bf_probe("meme")

hash 3 times → 0, 3, 5
"meme" most likely added to BF

bf_probe("java")

hash 3 times → 1, 2, 5

bf_probe("apple") → 0, 1, 4 most likely not added

BitVector : an array of bits
BitMatrix : each row was a row of bytes
BitVector represents a row like in BitMatrix
Set i^{th} bit:

```
byte = bv→vector[i/8]
mask = 1 << (i % 8)
bv_vector[i/8] = byte | mask
```

badspeak.txt
→ meme
→ java

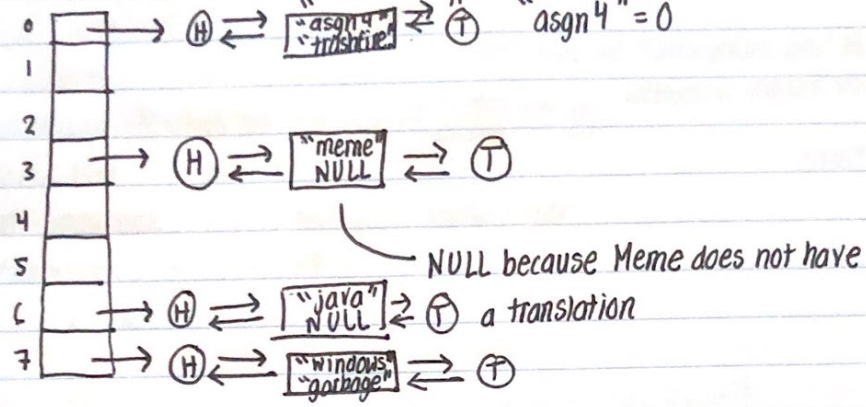
newspeak.txt
→ asgn 4 trashfire
→ windows garbage

oldspeak ⇒ newspeak translation

When there is a hash collision,

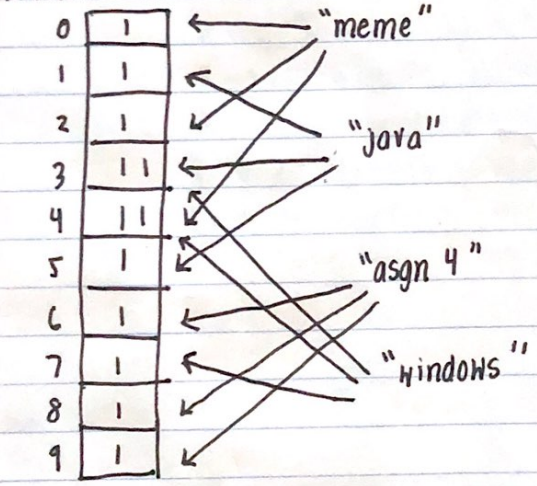
hash(ht → salt, "meme") = 3 " " "windows" = 7
" " "java" = 6
" " "asgn 4" = 0

Hash Table



NULL because Meme does not have a translation

Bloom Filter



Let's say user says "I used java for asgn 4"

bf_probe(... "i")
hash 1 = 2
hash 2 = 6
hash 3 = 9
⇓ since Bloom Filter full (worst case), we need to use hashtable and look it up
ht_lookup("i") = 3
did not pass the lookup, so we didn't add it

If the word does not pass the BloomFilter test, then we know its not a bad word
BloomFilter test and hashtable test are 2 checks for a word
let's say ht_lookup("java") hash → 6

returns 

← since no translation, add to forbidden linked list

Linked List *forbidden → (H) ⇄ [java NULL] ⇄ (T)

Linked List * to_revise

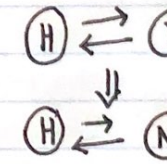
cat badspeak.txt | head | ./banhammer

./banhammer < input.txt

```
struct LinkedList {
    u32 length
    Node *head
    Node *tail
    bool mtf
};
```

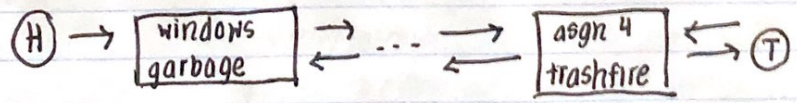
```
ll_create(bool mtf) {
```
↑ move to front

```
    LinkedList *ll = malloc(sizeof(LinkedList));
    ll→length = 0
    ll→head = node_create()
    ll→tail = node_create()
    ll→head→next = ll→tail
    ll→tail→prev = ll→head
    ll→mtf = mtf
}
```

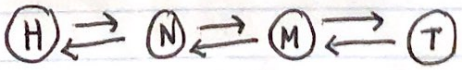


Inserting
- n→prev = head
- n→next = head→next
- head→next→prev = n
- head→next = n

Move to Front



If a word is used more often, we should move it to the front O(1) lookup



n→next = m→next
m→next→prev = m→prev
m→next = head→next
m→prev = head
head→next→prev = m
head→next = m

Lecture 3/02/2021

- Syntax errors
    - examine the syntax of your code
- Hard Bugs
    - location of the bug is hard to immediately pinpoint
    - finding these bugs
        - assert() statements
        - print statements
        - playing around with inputs
        - write test harnesses
            - test isolated functions

assert() is used to verify preconditions and postconditions

- precondition: condition that must be true before execution of some code
- postcondition: condition that must be true after execution of some code
- argument to assert() is a boolean expression
    - expression true? nothing happens

static analyzer: make sure the directory is clean, then run

```
$ scan-build make
```

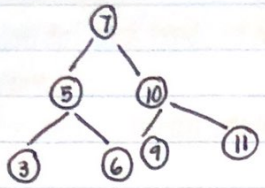
- fix any complaints that are issued
- possible for false positives to occur
- static analyzers work @ compile time, dynamic analyzers work @ runtime
    - can only catch things @ runtime (dynamic)
    - try to catch everything, even things that aren't bugs (static)

Lecture 3/04/2021:

- tree: nonlinear data structure, zero nodes or one node
- terms: root, parent, child, leaf, subtree, traversal, successor
- binary tree: node has at most, 2 children
  - left child (default Null)
  - reference to right child (default NULL)
- if a node has no children, it is considered to be a leaf
- 5 different types of binary trees
- binary search tree: left subtree of a node < than parent
  right subtree of a node > than parent
- runtime of finding is $O(\log n)$
- runtime of inserting is $O(\log n)$
- deleting a node is the hardest operation
  - if node is a leaf, remove leaf
  - if it has a child, child replaces it
  - 2 childs, order successor
- traversal: process of visiting each node in a tree
  - three diff. kinds of traversal, based on visiting order
- tries: digital search tree, similar to a tree data structure
  - ordered tree data structure
  - similar to tree data structure, nodes store entire alphabet
  - words are retrieved
- numerical quantities plots
- representing a graph: adjacency matrix ($n \times n$ matrix) - requires n^2 space
  - binary: edges present or absent - sparce matrix techniques improve it
  - weighted: $n \neq 0$
  - adjacency list
    - column array for nodes
    - linked list of edges from each node
    - may contain weights
- trees are a form of acyclic graphs (no loops)
  - DAG: directed acyclic graph
- routers are nodes with many edges

find while (not null & not found) ...

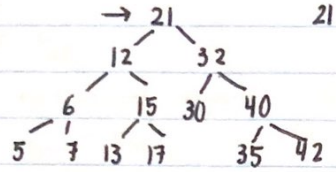
walk



preorder traversal: in order, left node, right

21, 12, 6, 5, 7, 15, 13, 17, 32, 30, 40, 35, 42

Preorder



to destroy a tree, walk the tree in post order traversal

Graphs

graph G = <V, E> (vertices and edges)

Node edge

edges = <A, B, N>

weight, capacity, anything you can quantify

DFS: deep first search

Eugene Lab Section 3/04/2021: Assignment 7

Lempel-Ziv Compression

- two different types of compression: lossy compression and lossless compression

abababab
1 2 3 4 2 16-bit unsigned integers uint16_t

0 is the stop code

"" + "a" = "a"

"ab" + "c" = "abc"

Example: cw (current word) = ""

abab cw + "a" = "a"

reset the current word back to the empty word

cw + "b" = "b"

Is "b" in the table? No? Add it to table then reset

cw + "a" = "a"

cw + "b" = "ab"

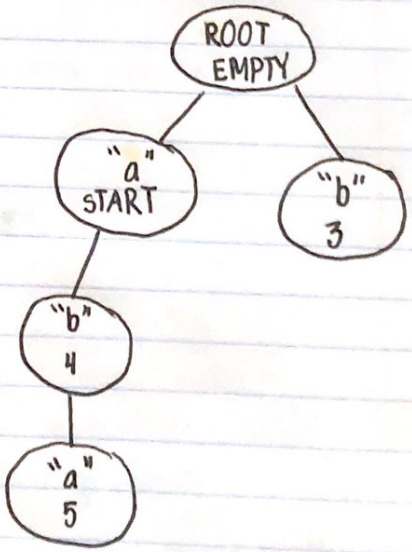
Add to dictionary

decoding: (empty_code, "a")
(empty_code, "b")
(start_code, "b")
} "abab"

"abab" + "aba"

WORD	CODE
""	EMPTY
"a"	START
"b"	3
"ab"	4
"aba"	5
	⋮

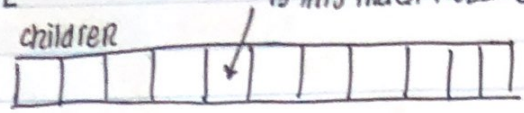
(EMPTY, "a")
(EMPTY, "b")
(START, "b")
(4, "a")




```
struct TrieNode {
        TrieNode *children[ALPHABET]; ← array of children
        uint16_t code;                  TrieNode pointers
};
```

NODE

children



To delete a trie: trie_del (root)
for each child
trie_del (child)

void trie_delete (TrieNode *n) is a recursive function
trie_reset() is not a recursive function (should use trie_delete())

```
trie_delete(Node *n)
        for c ∈ n.children
            trie_delete (c)
            n.children c = NULL
        }
        trie_node_delete (n)
```

```
struct Word {
        uint8_t *syms; ← array of symbols
        uint32_t len; ← # of symbols
};
```

A WordTable is an array of Words

word_append_sym (Word *w, uint8_t sym)
basically appends "ab" + "c" to "abc"

```
struct FileHeader {
        uint32_t magic;
        uint16_t protection; ← permissions
}
```

int read_bytes(int infile, uint8_t *buf, int to_read) using syscalls
open()
read() write()
close()
loop calls to read() until we have
either read all the bytes that were specified
(to_read) or there are no more bytes to read

u8 symbols [BLOCK] = 

int count = 0

count

```
read_sym (*sym)
        if count is 0 : {
                read_bytes (symbols, BLOCK)
        }
        *sym = symbols [index]
        index += 1
```

read_bytes() returns the # of bytes read

```
n = read_bytes (symbol, BLOCK)
if n < BLOCK {
    endOfBuffer = n + 1
If the endOf Buffer == index
    return false
while (read_sym (stdin, &sym)) {
        print (sym)
}
```

u8 bitsAndCodes [Block] bitAndCode

int bit index = 0

```
write_pair (outfile, code, sym, bitlen) {
        for i=0 ... bitlen
            is the ith bit of code set ?
                set the bit index bit
            else
                clear the bit Index bit
        for i=0 .. 8 (8 bits per symbol)
            is the ith bit of sym set ?
            .
            .
            .
If buffer is filled, then the value of bitIndex == BLOCK + 8
                write_bytes (bitsANDCode, BLOCK)
```

code = 13
bitlen 5

MSB 0000 1101 LSB
7654 3210

endian.rr

- is_big()
- is_little()
- swap16() ← swaps endianness of 16-bit type
- swap32()
- swap64()

header.magic is a u32 , header.protections is a u16

If (is_big() is true)

- header.magic = swap32(header.magic)
- header.protections = swap16(header.protections)

codes are u16