Data Structures

"Programming in the Large" Steps

Design & Implement

- Program & programming style (done)
- Common data structures and algorithms **<--** we are here
- Modularity
- Building techniques & tools (done)

Debug

- Debugging techniques & tools (done)

Test

- Testing techniques (done)

Maintain

- Performance improvement techniques & tools

Goals of this Lecture

Help you learn (or refresh your memory) about:

- Common data structures: linked lists and hash tables

Why? Deep motivation:

- Common data structures serve as "high level building blocks"
- A power programmer:
  - Rarely creates programs from scratch
  - Often creates programs using high level building blocks

Why? Shallow motivation:

- Provide background pertinent to Assignment 3
- … esp. for those who have not taken COS 226

Common Task

Maintain a collection of key/value pairs

- Each key is a **string**; each value is an **int**
- Unknown number of key-value pairs

Examples

- (student name, grade)
  - ("john smith", 84), ("jane doe", 93), ("bill clinton", 81)
- (baseball player, number)
  - ("Ruth", 3), ("Gehrig", 4), ("Mantle", 7)
- (variable name, value)
  - ("maxLength", 2000), ("i", 7), ("j", -10)

Agenda

Linked lists

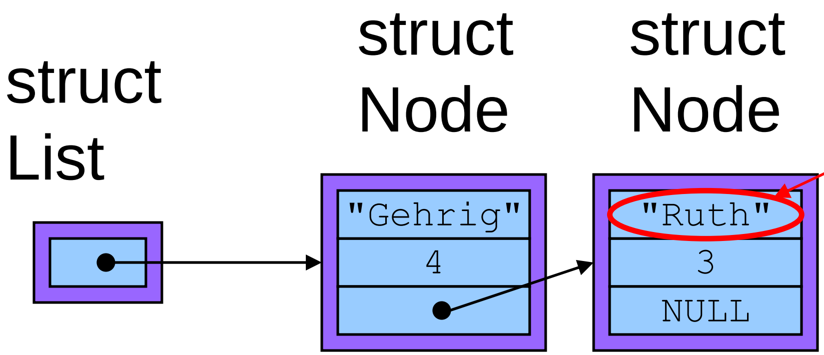
Hash tables

Hash table issues

Linked List Data Structure

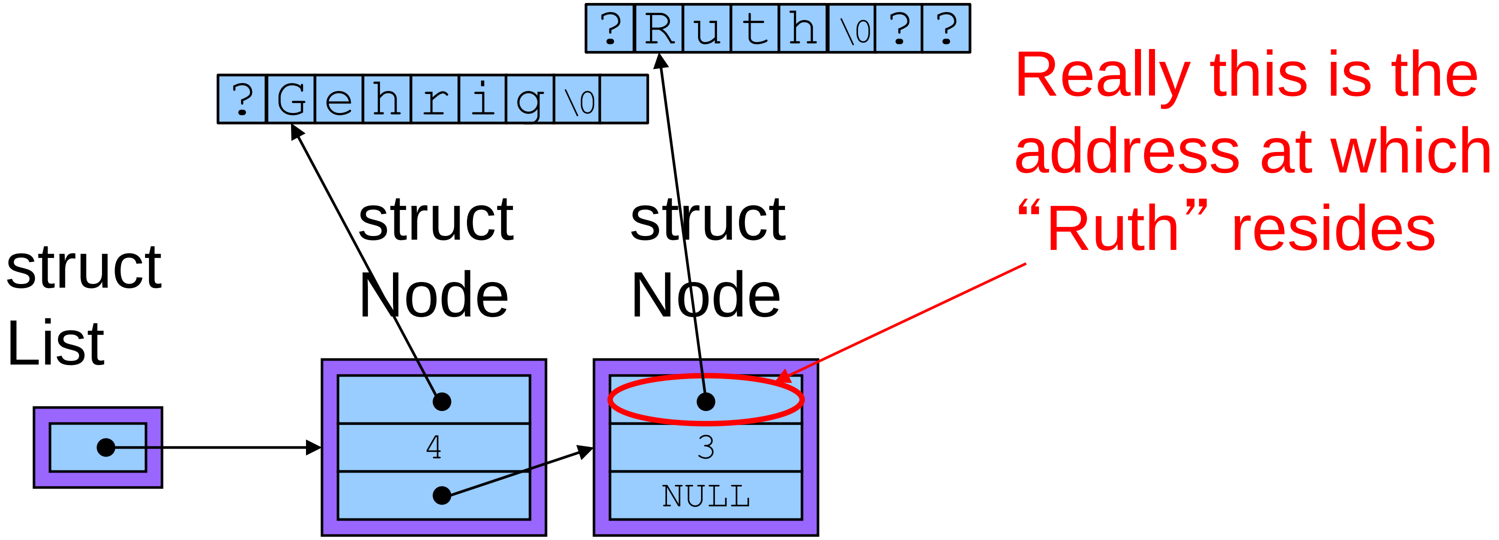
```
struct Node
{   const char *key;
    int value;
    struct Node *next;
};

struct List
{   struct Node *first;
};
```



Really this is the address at which “Ruth” resides

Linked List Data Structure



Linked List Algorithms

Create

- Allocate **`List`** structure; set **`first`** to **`NULL`**
- Performance: O(1) ⇒ fast

Add (no check for duplicate key required)

- Insert new node containing key/value pair at front of list
- Performance: O(1) ⇒ fast

Add (check for duplicate key required)

- Traverse list to check for node with duplicate key
- Insert new node containing key/value pair into list
- Performance: O(n) ⇒ slow

Linked List Algorithms

Search

- Traverse the list, looking for given key
- Stop when key found, or reach end
- Performance: O(n) ⇒ slow

Free

- Free **`Node`** structures while traversing
- Free **`List`** structure
- Performance: O(n) ⇒ slow

Would it be better to keep the nodes sorted by key?

Agenda

Linked lists

Hash tables

Hash table issues

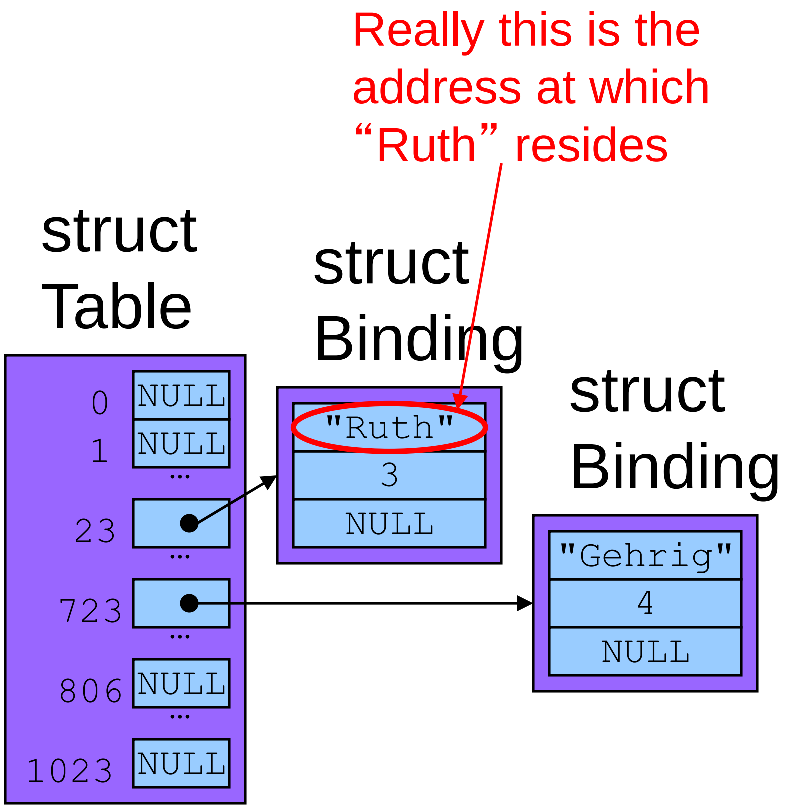
Hash Table Data Structure

Array of linked lists

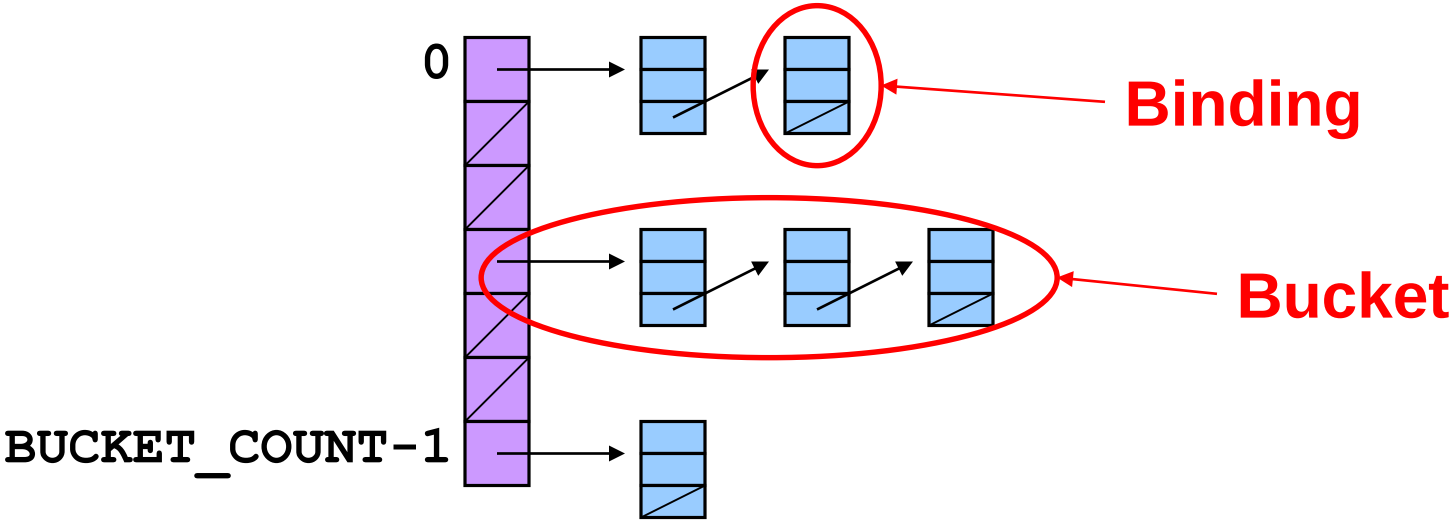
```
enum {BUCKET_COUNT = 1024};

struct Binding
{   const char *key;
    int value;
    struct Binding *next;
};

struct Table
{   struct Binding *buckets[BUCKET_COUNT];
};
```



Hash Table Data Structure



Hash function maps given key to an integer

Mod integer by `BUCKET_COUNT` to determine proper bucket

Hash Table Example

Example: **`BUCKET_COUNT`** = 7

Add (if not already present) bindings with these keys:

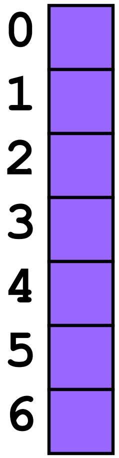
- the, cat, in, the, hat

Hash Table Example (cont.)

First key: "the"

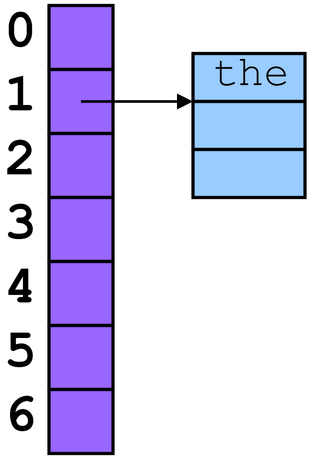
- hash("the") = 965156977; 965156977 % 7 = 1

Search **`buckets[1]`** for binding with key "the"; not found



Hash Table Example (cont.)

Add binding with key "the" and its value to **`buckets[1]`**

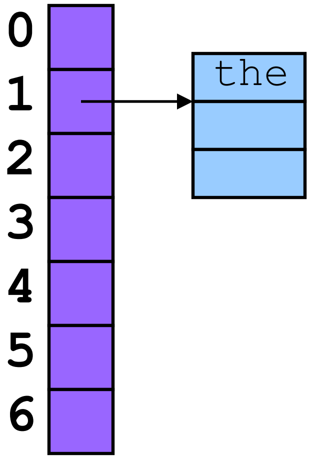


Hash Table Example (cont.)

Second key: "cat"

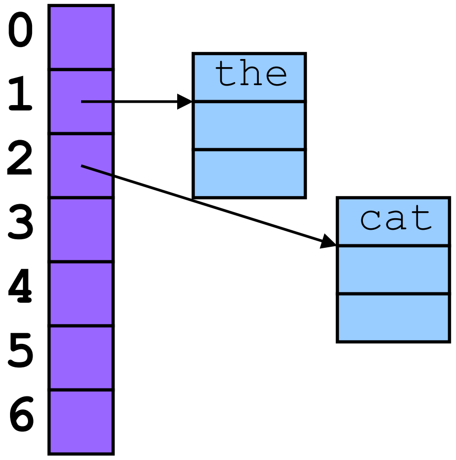
- hash("cat") = 3895848756; 3895848756 % 7 = 2

Search **`buckets[2]`** for binding with key "cat"; not found



Hash Table Example (cont.)

Add binding with key "cat" and its value to **`buckets[2]`**

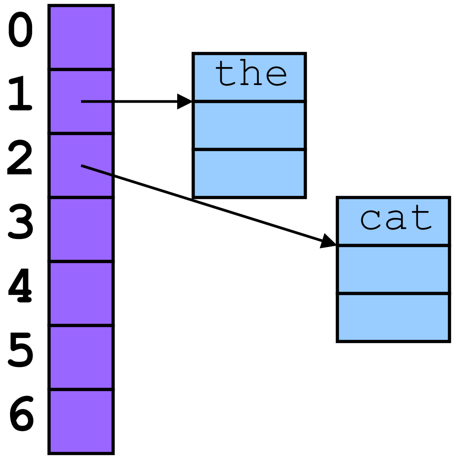


Hash Table Example (cont.)

Third key: "in"

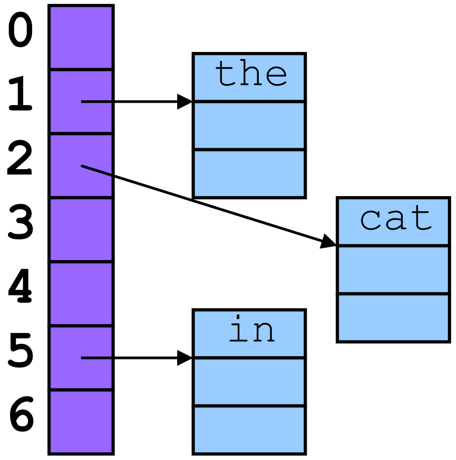
- hash("in") = 6888005; 6888005% 7 = 5

Search **`buckets[5]`** for binding with key "in"; not found



Hash Table Example (cont.)

Add binding with key “in” and its value to **`buckets[5]`**



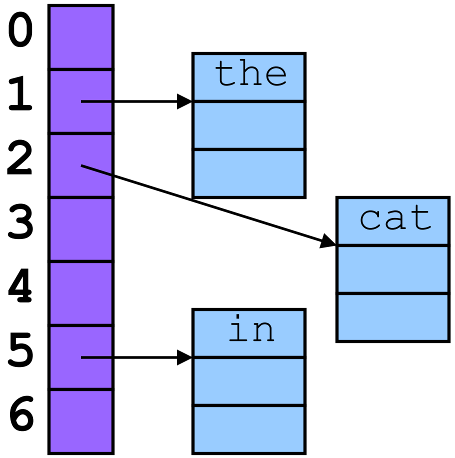
Hash Table Example (cont.)

Fourth word: "the"

- hash("the") = 965156977; 965156977 % 7 = 1

Search **`buckets[1]`** for binding with key "the"; found it!

- Don't change hash table

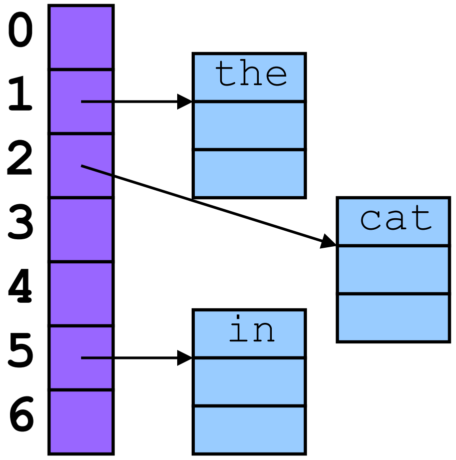


Hash Table Example (cont.)

Fifth key: "hat"

- hash("hat") = 865559739; 865559739 % 7 = 2

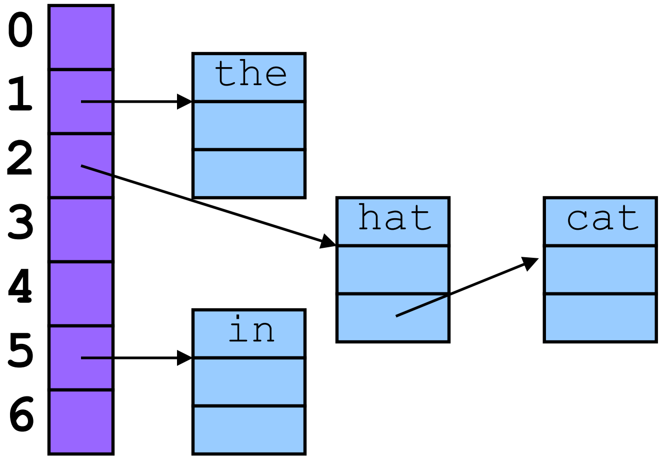
Search **`buckets[2]`** for binding with key "hat"; not found



Hash Table Example (cont.)

Add binding with key “hat” and its value to **`buckets[2]`**

- At front or back? Doesn’t matter
- Inserting at the front is easier, so add at the front



Hash Table Algorithms

Create

- Allocate **`Table`** structure; set each bucket to **`NULL`**
- Performance: O(1) ⇒ fast

Add

- Hash the given key
- Mod by **`BUCKET_COUNT`** to determine proper bucket
- Traverse proper bucket to make sure no duplicate key
- Insert new binding containing key/value pair into proper bucket
- Performance: O(1) ⇒ fast

Is the add performance always fast?

Hash Table Algorithms

Search

- Hash the given key
- Mod by **`BUCKET_COUNT`** to determine proper bucket
- Traverse proper bucket, looking for binding with given key
- Stop when key found, or reach end
- Performance: O(1) ⇒ fast

Is the search performance always fast?

Free

- Traverse each bucket, freeing bindings
- Free **`Table`** structure
- Performance: O(n) ⇒ slow

Agenda

Linked lists

Hash tables

Hash table issues

How Many Buckets?

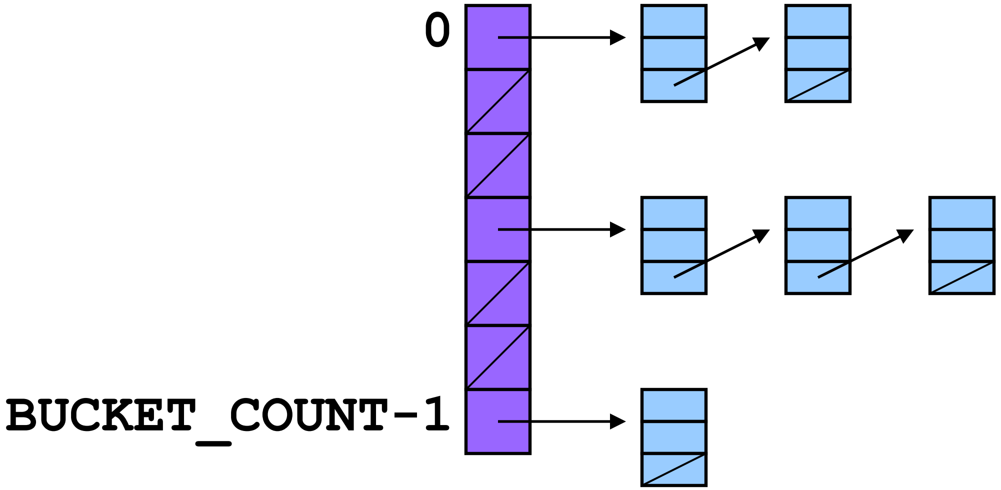
Many!

- Too few ⇒ large buckets ⇒ slow add, slow search

But not too many!

- Too many ⇒ memory is wasted

This is OK:

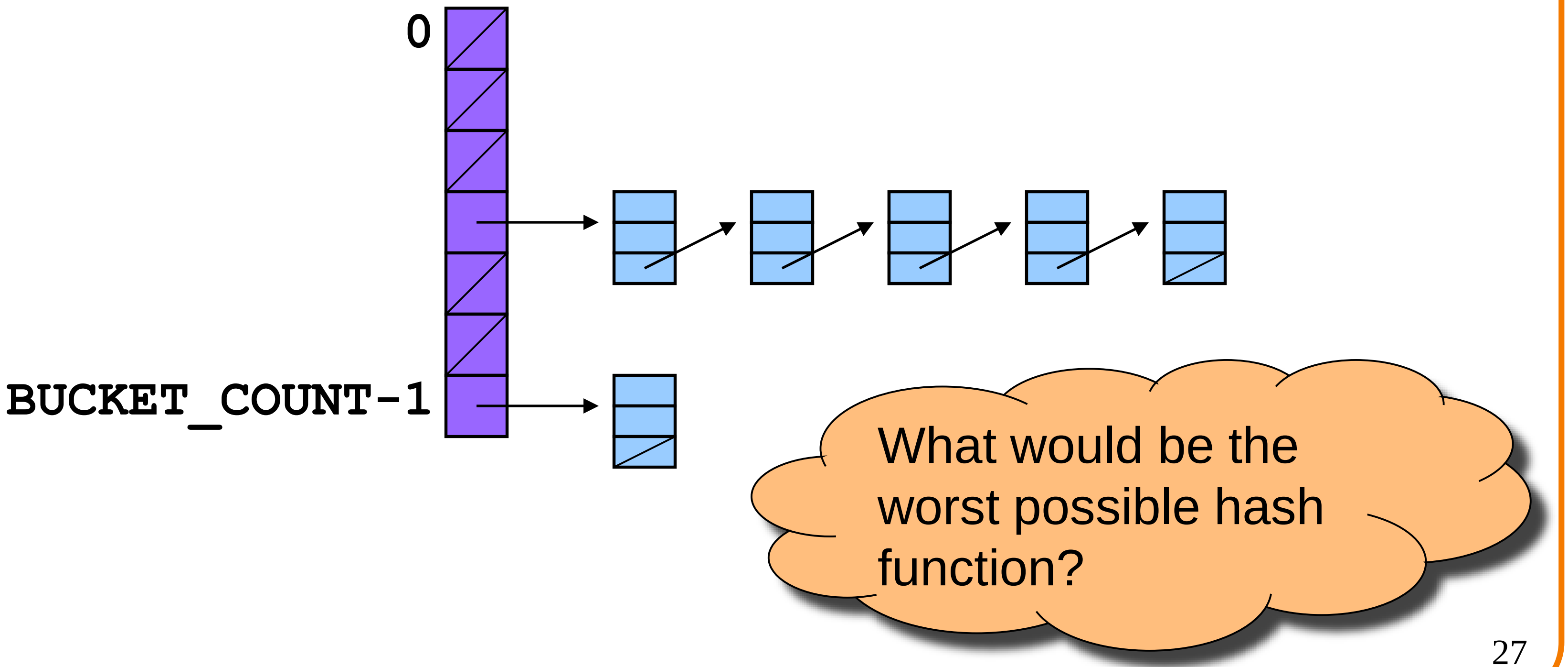


What Hash Function?

Should distribute bindings across the buckets well

- Distribute bindings over the range **`0, 1, …, BUCKET_COUNT-1`**
- Distribute bindings *evenly* to avoid very long buckets

This is not so good:



How to Hash Strings?

Simple hash schemes don't distribute the keys evenly enough

- Number of characters, mod **BUCKET_COUNT**
- Sum the numeric codes of all characters, mod **BUCKET_COUNT**
- …

A reasonably good hash function:

- Weighted sum of characters s_i in the string **s**
  - $(\Sigma\ a^i s_i) \bmod \mathrm{BUCKET\_COUNT}$
- Best if **a** and **BUCKET_COUNT** are relatively prime
  - E.g., **a** = 65599, **BUCKET_COUNT** = 1024

Why?

Footnote: I originally designed this homework so that BUCKET_COUNT is a prime number. In 2016 I wondered, "wouldn't it work just as well if a and BUCKET_COUNT are just relatively prime? Measurements show no: using a prime number of buckets leads to more even distribution of bucket contents."

How to Hash Strings?

Potentially expensive to compute $\Sigma\ a^i s_i$

So let's do some algebra ("Horner's rule")

- (by example, for string **s** of length 5, **a**=65599):

$h = \Sigma 65599^i * s_i$

$h = 65599^0 * s_0 + 65599^1 * s_1 + 65599^2 * s_2 + 65599^3 * s_3 + 65599^4 * s_4$

Direction of traversal of s doesn't matter, so…

$h = 65599^0 * s_4 + 65599^1 * s_3 + 65599^2 * s_2 + 65599^3 * s_1 + 65599^4 * s_0$

$h = 65599^4 * s_0 + 65599^3 * s_1 + 65599^2 * s_2 + 65599^1 * s_3 + 65599^0 * s_4$

$h = (((((s_0) * 65599 + s_1) * 65599 + s_2) * 65599 + s_3) * 65599) + s_4$

How to Hash Strings?

Yielding this function

```
size_t hash(const char *s, size_t bucketCount)
{  size_t i;
   size_t h = 0;
   for (i=0; s[i]!='\0'; i++)
      h = h * 65599 + (size_t)s[i];
   return h % bucketCount;
}
```

How to Protect Keys?

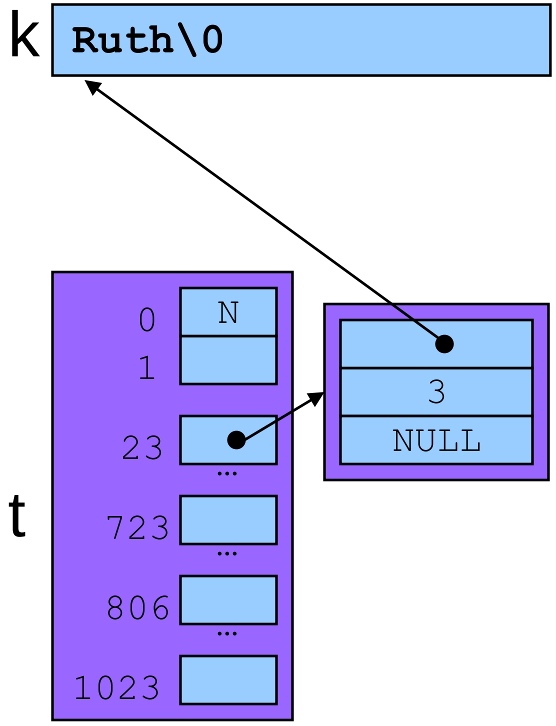
Suppose **`Table_add()`** function contains this code:

```
void Table_add(struct Table *t, const char *key, int value)
{   …
    struct Binding *p =
        (struct Binding*)malloc(sizeof(struct Binding));
    p->key = key;
    …
}
```

How to Protect Keys?

Problem: Consider this calling code:

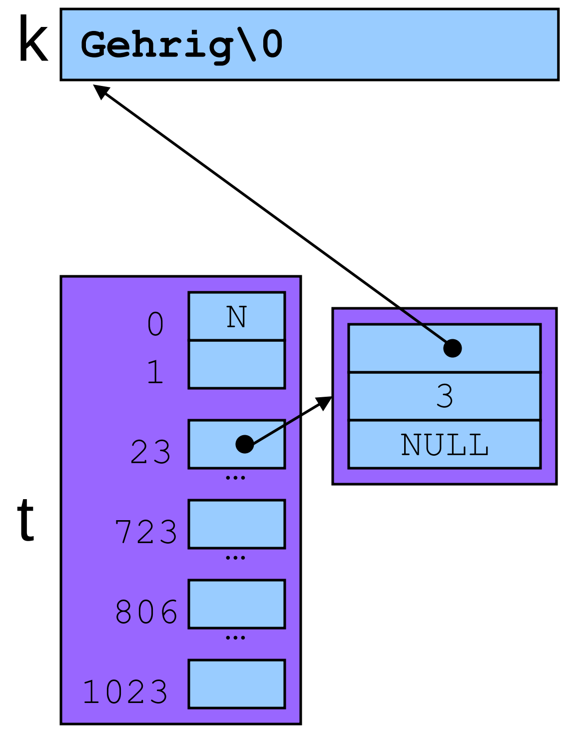
```
struct Table *t;
char k[100] = "Ruth";
…
Table_add(t, k, 3);
```



How to Protect Keys?

Problem: Consider this calling code:

```
struct Table *t;
char k[100] = "Ruth";
…
Table_add(t, k, 3);
strcpy(k, "Gehrig");
```



What happens if the client searches t for "Ruth"? For Gehrig?

How to Protect Keys?

Solution: **`Table_add()`** saves a **defensive copy** of the given key

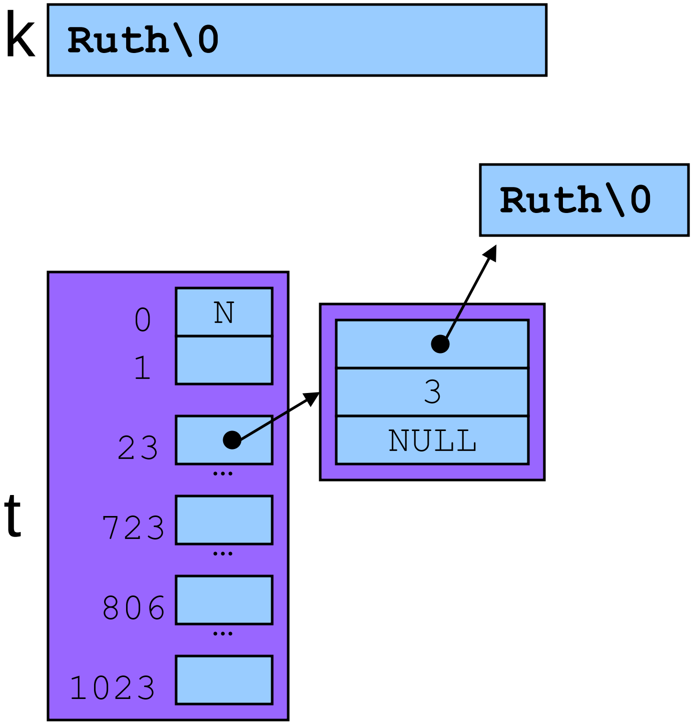
```
void Table_add(struct Table *t, const char *key, int value)
{   …
    struct Binding *p =
        (struct Binding*)malloc(sizeof(struct Binding));
    p->key = (const char*)malloc(strlen(key) + 1);
    strcpy((char*)p->key, key);
    …
}
```

Why add 1?

How to Protect Keys?

Now consider same calling code:

```
struct Table *t;
char k[100] = "Ruth";
…
Table_add(t, k, 3);
```

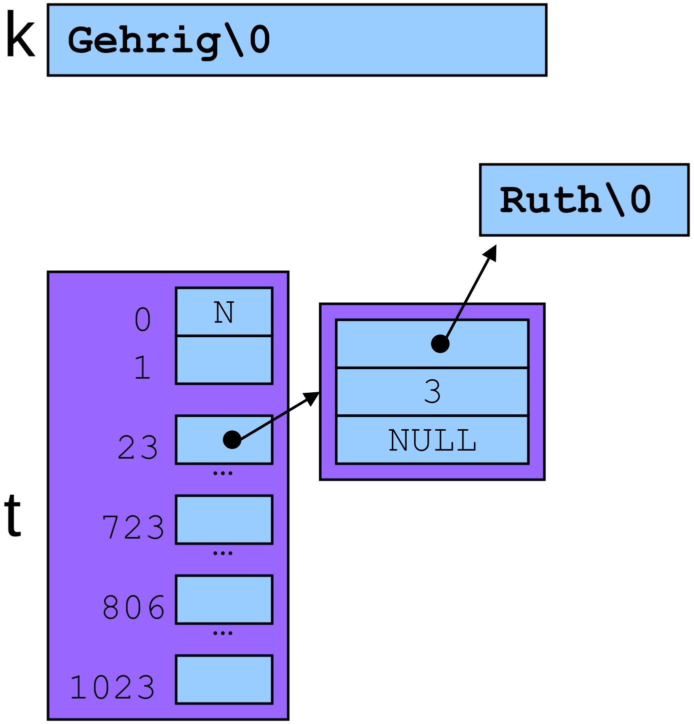


How to Protect Keys?

Now consider same calling code:

```
struct Table *t;
char k[100] = "Ruth";
…
Table_add(t, k, 3);
strcpy(k, "Gehrig");
```

Hash table is
not corrupted



Who Owns the Keys?

Then the hash table **owns** its keys

- That is, the hash table owns the memory in which its keys reside
- **`Hash_free()`** function must free the memory in which the key resides

Summary

Common data structures and associated algorithms

- Linked list
  - (Maybe) fast add
  - Slow search
- Hash table
  - (Potentially) fast add
  - (Potentially) fast search
  - Very common

Hash table issues

- Hashing algorithms
- Defensive copies
- Key ownership

Debugging (Part 2)

"Programming in the Large" Steps

Design & Implement

- Program & programming style (done)
- Common data structures and algorithms
- Modularity
- Building techniques & tools (done)

Test

- Testing techniques (done)

Debug

- Debugging techniques & tools **<--** we are still here

Maintain

- Performance improvement techniques & tools

Goals of this Lecture

Help you learn about:

- Debugging strategies & tools related to **dynamic memory management (DMM)** *

Why?

- Many bugs occur in code that does DMM
- DMM errors can be difficult to find
  - DMM error in one area can manifest itself in a distant area
- A power programmer knows a wide variety of DMM debugging **strategies**
- A power programmer knows about **tools** that facilitate DMM debugging

* Management of heap memory via **`malloc()`**, **`calloc()`**, **`realloc()`**, and **`free()`**

Agenda

(9) Look for common DMM bugs

(10) Diagnose seg faults using gdb

(11) Manually inspect malloc calls

(12) Hard-code malloc calls

(13) Comment-out free calls

(14) Use Meminfo

(15) Use Valgrind

Look for Common DMM Bugs

Some of our favorites:

```
int *p; /* value of p undefined */
…
*p = somevalue;
```

```
char *p; /* value of p undefined */
…
fgets(p, 1024, stdin);
```

```
int *p;
…
p = (int*)malloc(sizeof(int));
…
*p = 5;
…
free(p);
…
*p = 6;
```

What are the errors?

Look for Common DMM Bugs

Some of our favorites:

```
int *p;
…
p = (int*)malloc(sizeof(int));
…
*p = 5;
…
p = (int*)malloc(sizeof(int));
```

```
int *p;
…
p = (int*)malloc(sizeof(int));
…
*p = 5;
…
free(p);
…
free(p);
```

What are the errors?

Agenda

(9) Look for common DMM bugs

(10) Diagnose seg faults using gdb

(11) Manually inspect malloc calls

(12) Hard-code malloc calls

(13) Comment-out free calls

(14) Use Meminfo

(15) Use Valgrind

Diagnose Seg Faults Using GDB

Segmentation fault => make it happen in gdb

- Then issue the gdb **`where`** command
- Output will lead you to the line that caused the fault
  - But that line may not be where the error resides!

Agenda

(9) Look for common DMM bugs

(10) Diagnose seg faults using gdb

(11) Manually inspect malloc calls

(12) Hard-code malloc calls

(13) Comment-out free calls

(14) Use Meminfo

(15) Use Valgrind

Manually Inspect Malloc Calls

Manually inspect each call of `malloc()`

- Make sure it allocates enough memory

Do the same for `calloc()` and `realloc()`

Manually Inspect Malloc Calls

Some of our favorites:

```
char *s1 = "hello, world";
char *s2;
s2 = (char*)malloc(strlen(s1));
strcpy(s2, s1);
```

```
char *s1 = "Hello";
char *s2;
s2 = (char*)malloc(sizeof(s1));
strcpy(s2, s1);
```

```
long double *p;
p = (long double*)malloc(sizeof(long double*));
```

```
long double *p;
p = (long double*)malloc(sizeof(p));
```

What are the errors?

Agenda

(9) Look for common DMM bugs

(10) Diagnose seg faults using gdb

(11) Manually inspect malloc calls

(12) Hard-code malloc calls

(13) Comment-out free calls

(14) Use Meminfo

(15) Use Valgrind

Hard-Code Malloc Calls

Temporarily change each call of **`malloc()`** to request a large number of bytes

- Say, 10000 bytes
- If the error disappears, then at least one of your calls is requesting too few bytes

Then incrementally restore each call of **`malloc()`** to its previous form

- When the error reappears, you might have found the culprit

Do the same for **`calloc()`** and **`realloc()`**

Agenda

(9) Look for common DMM bugs

(10) Diagnose seg faults using gdb

(11) Manually inspect malloc calls

(12) Hard-code malloc calls

(13) Comment-out free calls

(14) Use Meminfo

(15) Use Valgrind

Comment-Out Free Calls

Temporarily comment-out every call of **`free()`**

- If the error disappears, then program is
  - Freeing memory too soon, or
  - Freeing memory that already has been freed, or
  - Freeing memory that should not be freed,
  - Etc.

Then incrementally "comment-in" each call of **`free()`**

- When the error reappears, you might have found the culprit

Agenda

(9) Look for common DMM bugs

(10) Diagnose seg faults using gdb

(11) Manually inspect malloc calls

(12) Hard-code malloc calls

(13) Comment-out free calls

(14) Use Meminfo

(15) Use Valgrind

Use Meminfo

Use the **Meminfo** tool

- Simple tool
- Initial version written by Dondero
- Current version written by COS 217 alumnus RJ Liljestrom
- Reports errors **after** program execution
  - Memory leaks
  - Some memory corruption
- User-friendly output

Appendix 1 provides example buggy programs

Appendix 2 provides Meminfo analyses

Agenda

(9) Look for common DMM bugs

(10) Diagnose seg faults using gdb

(11) Manually inspect malloc calls

(12) Hard-code malloc calls

(13) Comment-out free calls

(14) Use Meminfo

(15) Use Valgrind

Use Valgrind

Use the **Valgrind** tool

- Complex tool
- Written by multiple developers, worldwide
  - See www.valgrind.org
- Reports errors **during** program execution
  - Memory leaks
  - Multiple frees
  - Dereferences of dangling pointers
  - Memory corruption
- Comprehensive output
  - But not always user-friendly

Use Valgrind

Appendix 1 provides example buggy programs

Appendix 3 provides Valgrind analyses

Summary

Strategies and tools for debugging the DMM aspects of your code:

- Look for common DMM bugs
- Diagnose seg faults using gdb
- Manually inspect malloc calls
- Hard-code malloc calls
- Comment-out free calls
- Use Meminfo
- Use Valgrind

Appendix 1: Buggy Programs

leak.c

```
 1. #include <stdio.h>
 2. #include <stdlib.h>
 3. int main(void)
 4. {   int *pi;
 5.     pi = (int*)malloc(sizeof(int));
 6.     *pi = 5;
 7.     printf("%d\n", *pi);
 8.     pi = (int*)malloc(sizeof(int));
 9.     *pi = 6;
10.     printf("%d\n", *pi);
11.     free(pi);
12.     return 0;
13. }
```

Memory leak:

Memory allocated at line 5 is leaked

Appendix 1: Buggy Programs

doublefree.c

```
 1. #include <stdio.h>
 2. #include <stdlib.h>
 3. int main(void)
 4. {   int *pi;
 5.     pi = (int*)malloc(sizeof(int));
 6.     *pi = 5;
 7.     printf("%d\n", *pi);
 8.     free(pi);
 9.     free(pi);
10.     return 0;
11. }
```

Multiple free:

Memory allocated at line 5 is freed twice

Appendix 1: Buggy Programs

danglingptr.c

```
 1. #include <stdio.h>
 2. #include <stdlib.h>
 3. int main(void)
 4. {   int *pi;
 5.     pi = (int*)malloc(sizeof(int));
 6.     *pi = 5;
 7.     printf("%d\n", *pi);
 8.     free(pi);
 9.     printf("%d\n", *pi);
10.     return 0;
11. }
```

Dereference of dangling pointer:

Memory accessed at line 9 already was freed

Appendix 1: Buggy Programs

toosmall.c

```
 1. #include <stdio.h>
 2. #include <stdlib.h>
 3. int main(void)
 4. {   int *pi;
 5.     pi = (int*)malloc(1);
 6.     *pi = 5;
 7.     printf("%d\n", *pi);
 8.     free(pi);
 9.     return 0;
10. }
```

Memory corruption:

Too little memory is allocated at line 5

Line 6 corrupts memory

Appendix 2: Meminfo

Meminfo can detect memory leaks:

```
$ gcc217m leak.c -o leak
$ leak
5
6
$ ls
.  ..  leak.c  leak  meminfo30462.out
$ meminforeport meminfo30462.out
Errors:
   ** 4 un-freed bytes (1 block) allocated at leak.c:5
Summary Statistics:
   Maximum bytes allocated at once: 8
   Total number of allocated bytes: 8
Statistics by Line:
          Bytes   Location
             -4   leak.c:11
              4   leak.c:5
              4   leak.c:8
              4   TOTAL
Statistics by Compilation Unit:
              4   leak.c
              4   TOTAL
```

Appendix 2: Meminfo

Meminfo can detect memory corruption:

```
$ gcc217m toosmall.c -o toosmall
$ toosmall
5
$ ls
.  ..  toosmall.c  toosmall  meminfo31891.out
$ meminforeport meminfo31891.out
Errors:
   ** Underflow detected at toosmall.c:8 for memory allocated at toosmall.c:5
Summary Statistics:
   Maximum bytes allocated at once: 1
   Total number of allocated bytes: 1
Statistics by Line:
          Bytes   Location
              1   toosmall.c:5
             -1   toosmall.c:8
              0   TOTAL
Statistics by Compilation Unit:
              0   toosmall.c
              0   TOTAL
```

Appendix 2: Meminfo

Meminfo caveats:

- Don't mix .o files built with **`gcc217`** and **`gcc217m`**
- **`meminfo*.out`** files can be large
  - Should delete frequently
- Programs built with **`gcc217m`** run slower than those built with **`gcc217`**
  - Don't build with **`gcc217m`** when doing timing tests

Appendix 3: Valgrind

Valgrind can detect memory leaks:

```
$ gcc217 leak.c -o leak
$ valgrind leak
==31921== Memcheck, a memory error detector
==31921== Copyright (C) 2002-2012, and GNU GPL'd, by Julian Seward et al.
==31921== Using Valgrind-3.8.1 and LibVEX; rerun with -h for copyright info
==31921== Command: leak
==31921==
5
6
==31921==
==31921== HEAP SUMMARY:
==31921==     in use at exit: 4 bytes in 1 blocks
==31921==   total heap usage: 2 allocs, 1 frees, 8 bytes allocated
==31921==
==31921== LEAK SUMMARY:
==31921==    definitely lost: 4 bytes in 1 blocks
==31921==    indirectly lost: 0 bytes in 0 blocks
==31921==      possibly lost: 0 bytes in 0 blocks
==31921==    still reachable: 0 bytes in 0 blocks
==31921==         suppressed: 0 bytes in 0 blocks
==31921== Rerun with --leak-check=full to see details of leaked memory
==31921==
==31921== For counts of detected and suppressed errors, rerun with: -v
==31921== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 6 from 6)
```

Appendix 3: Valgrind

Valgrind can detect memory leaks:

```
$ valgrind --leak-check=full leak
==476== Memcheck, a memory error detector
==476== Copyright (C) 2002-2012, and GNU GPL'd, by Julian Seward et al.
==476== Using Valgrind-3.8.1 and LibVEX; rerun with -h for copyright info
==476== Command: leak
==476==
5
6
==476==
==476== HEAP SUMMARY:
==476==     in use at exit: 4 bytes in 1 blocks
==476==   total heap usage: 2 allocs, 1 frees, 8 bytes allocated
==476==
==476== 4 bytes in 1 blocks are definitely lost in loss record 1 of 1
==476==    at 0x4A069EE: malloc (vg_replace_malloc.c:270)
==476==    by 0x400565: main (leak.c:5)
==476==
==476== LEAK SUMMARY:
==476==    definitely lost: 4 bytes in 1 blocks
==476==    indirectly lost: 0 bytes in 0 blocks
==476==      possibly lost: 0 bytes in 0 blocks
==476==    still reachable: 0 bytes in 0 blocks
==476==         suppressed: 0 bytes in 0 blocks
==476==
==476== For counts of detected and suppressed errors, rerun with: -v
==476== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 6 from 6)
```

Appendix 3: Valgrind

Valgrind can detect multiple frees:

```
$ gcc217 doublefree.c -o doublefree
$ valgrind doublefree
==31951== Memcheck, a memory error detector
==31951== Copyright (C) 2002-2012, and GNU GPL'd, by Julian Seward et al.
==31951== Using Valgrind-3.8.1 and LibVEX; rerun with -h for copyright info
==31951== Command: doublefree
==31951==
5
==31951== Invalid free() / delete / delete[] / realloc()
==31951==    at 0x4A063F0: free (vg_replace_malloc.c:446)
==31951==    by 0x4005A5: main (doublefree.c:9)
==31951==  Address 0x4c2a040 is 0 bytes inside a block of size 4 free'd
==31951==    at 0x4A063F0: free (vg_replace_malloc.c:446)
==31951==    by 0x400599: main (doublefree.c:8)
==31951==
==31951==
==31951== HEAP SUMMARY:
==31951==     in use at exit: 0 bytes in 0 blocks
==31951==   total heap usage: 1 allocs, 2 frees, 4 bytes allocated
==31951==
==31951== All heap blocks were freed -- no leaks are possible
==31951==
==31951== For counts of detected and suppressed errors, rerun with: -v
==31951== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 6 from 6)
```

Appendix 3: Valgrind

Valgrind can detect dereferences of dangling pointers:

```
$ gcc217 danglingptr.c -o danglingptr
$ valgrind danglingptr
==336== Memcheck, a memory error detector
==336== Copyright (C) 2002-2012, and GNU GPL'd, by Julian Seward et al.
==336== Using Valgrind-3.8.1 and LibVEX; rerun with -h for copyright info
==336== Command: danglingptr
==336==
5
==336== Invalid read of size 4
==336==    at 0x40059E: main (danglingptr.c:9)
==336==  Address 0x4c2a040 is 0 bytes inside a block of size 4 free'd
==336==    at 0x4A063F0: free (vg_replace_malloc.c:446)
==336==    by 0x400599: main (danglingptr.c:8)
==336==
5
==336==
==336== HEAP SUMMARY:
==336==     in use at exit: 0 bytes in 0 blocks
==336==   total heap usage: 1 allocs, 1 frees, 4 bytes allocated
==336==
==336== All heap blocks were freed -- no leaks are possible
==336==
==336== For counts of detected and suppressed errors, rerun with: -v
==336== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 6 from 6)
```

Appendix 3: Valgrind

Valgrind can detect memory corruption:

```
$ gcc217 toosmall.c -o toosmall
$ valgrind toosmall
==436== Memcheck, a memory error detector
==436== Copyright (C) 2002-2012, and GNU GPL'd, by Julian Seward et al.
==436== Using Valgrind-3.8.1 and LibVEX; rerun with -h for copyright info
==436== Command: toosmall
==436==
==436== Invalid write of size 4
==436==    at 0x40056E: main (toosmall.c:6)
==436==  Address 0x4c2a040 is 0 bytes inside a block of size 1 alloc'd
==436==    at 0x4A069EE: malloc (vg_replace_malloc.c:270)
==436==    by 0x400565: main (toosmall.c:5)
==436==
==436== Invalid read of size 4
==436==    at 0x400578: main (toosmall.c:7)
==436==  Address 0x4c2a040 is 0 bytes inside a block of size 1 alloc'd
==436==    at 0x4A069EE: malloc (vg_replace_malloc.c:270)
==436==    by 0x400565: main (toosmall.c:5)
==436==
5
```

Continued on next slide

Appendix 3: Valgrind

Valgrind can detect memory corruption (cont.):

Continued from previous slide

```
==436==
==436== HEAP SUMMARY:
==436==     in use at exit: 0 bytes in 0 blocks
==436==   total heap usage: 1 allocs, 1 frees, 1 bytes allocated
==436==
==436== All heap blocks were freed -- no leaks are possible
==436==
==436== For counts of detected and suppressed errors, rerun with: -v
==436== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 6 from 6)
```

Appendix 3: Valgrind

Valgrind caveats:

- Not intended for programmers who are new to C
  - Messages may be cryptic
- Suggestion:
  - Observe line numbers referenced by messages
  - Study code at those lines
  - Infer meanings of messages