CSE 221: Algorithms and Data Structures Lecture #8 The Constant Struggle for Hash

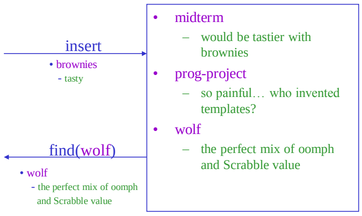
Steve Wolfman
2014W1

Today's Outline

- Constant-Time Dictionaries?
- Getting to Hash Tables by Doing Everything Wrong
  - First Pass: Plain Vectors
  - Second Pass: A Size Problem Resolved?
  - Third Pass: Crowding Resolved?
  - Fourth Pass: Allowing Diverse Keys?
  - Fifth Pass: Where the "Hash" Comes From
  - Third Pass, Take Two: Crowding Resolved Again?
- Hash Tables…

Reminder: Dictionary ADT

- Dictionary operations
  - create
  - destroy
  - insert
  - find
  - delete



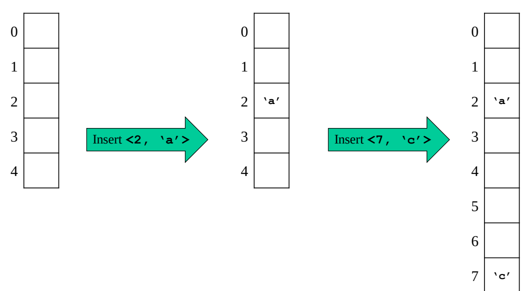
- Stores *values* associated with user-specified *keys*
  - values may be any (homogenous) type
  - keys may be any (homogenous) comparable type

Implementations So Far

	insert	find	delete
• Unsorted array	O(1)	O(n)	O(n)
• Balanced Trees	O(log n)	O(log n)	O(log n)

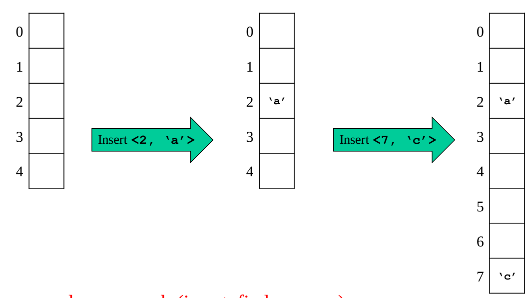
How about O(1) insert/find/delete for any key type?

First Pass: Resizable Vectors



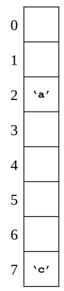
How will insert, find, and delete work?
What is an "empty" cell in the table?

What's Wrong with Our First Pass?

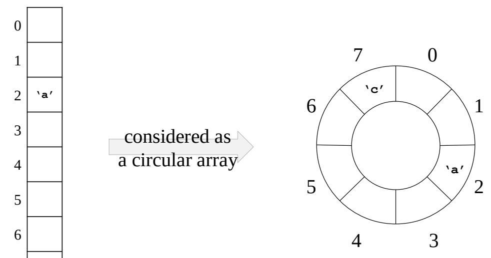


Give example commands (insert, find, remove) that illustrate what's wrong!

What is the 25th Element?

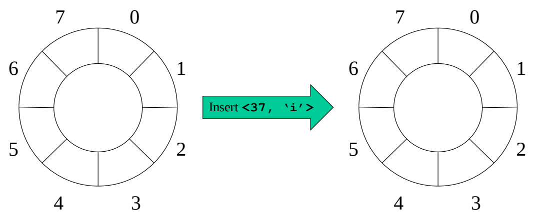


What is the 25th Element Now?



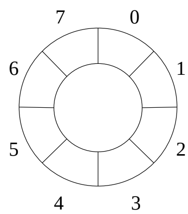
What's the largest possible element?

Second Pass: Circular Array (For the Win?)

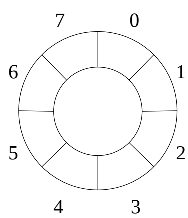


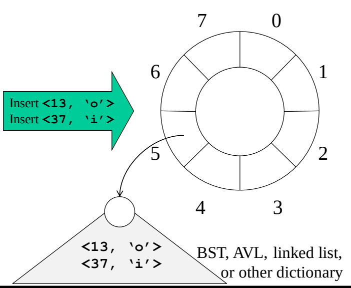
Does this solve our memory usage problem?

What's Wrong with our **Second** Pass?



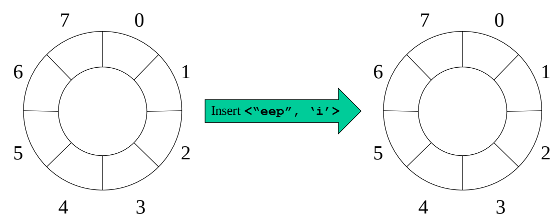
Third Pass: Punt to Another Dictionary?





When should we resize in this case?

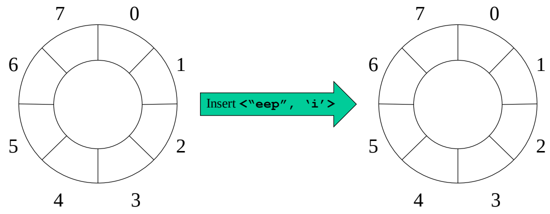
How Do We Turn Strings into Numbers?



What should we do?

Fourth Pass:
Strings **ARE** Numbers

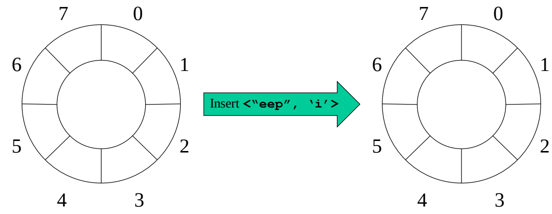
`e        e        p`
`01100101 01100101 01110000` = 6,645,104   6,645,104 % 8 = 0



13

What's Wrong with Our Fourth Pass?

`e        e        p`
`01100101 01100101 01110000` = 6,645,104   6,645,104 % 8 = 0

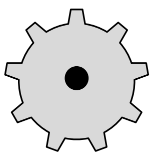


Antidisestablshmentarianism. Just saying.

14

Fifth Pass:
Hashing!

We only need perhaps a 64 (128?) bit number.
There's no point in forming a **huge** number.
We need a function to turn the strings into numbers, typically on a bounded range…

Antidisestablshmentarianism ⟶  ⟶ 1,097,757,801

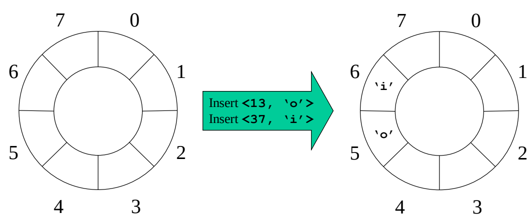
15

Schlemiel, Schlemazel,
Trouble for Our Hash Table?

- Let's try out:
  - "schlemiel" and "schlemazel"?
  - "microscopic" and "telescopic"?
  - "abcdefghijklmnopqrstuvwxyzyxwvutsrqponmlkjihgfedcba" and "abcdefghijklmnopqrstuvwxyzzyxwvutsrqponmlkjihgfedcba"

16

Third Pass, Take Two:
Punt to Another Slot?



Slot 5 is full, but no "dictionaries in each slot" this time.
Overflow to slot 6? When should we resize?

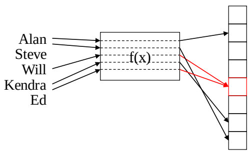
Today's Outline

- Constant-Time Dictionaries?
- Hash Table Outline
- Hash Functions
- Collisions and the Pigeonhole Principle
- Collision Resolution:
  - Chaining
  - Open-Addressing

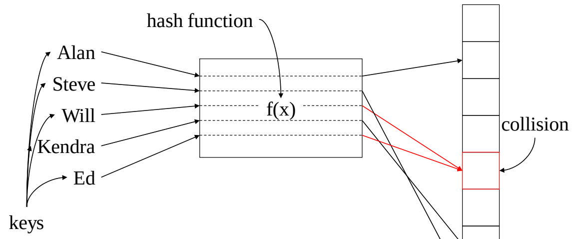
18

Hash Table
Dictionary Data Structure

- Hash function: maps keys to integers
  - result: can quickly find the right spot for a given entry
- Unordered and sparse table
  - result: cannot efficiently list all entries, list all entries between two keys (a "***range query***"), find minimum, find maximum, etc.



Hash Table Terminology



load factor $\lambda = \frac{\text{\# of entries in table}}{\text{tableSize}}$

Hash Table Code
First Pass

```
Value & find(Key & key) {
  int index = hash(key) % tableSize;
  return Table[index];
}
```

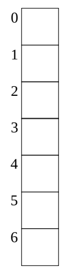
What should the hash function be?

How should we resolve collisions?

What should the table size be?

Practice

Insert 2
Insert 5
Insert 10
Think about inserting 9
Find 10
Insert 14
Insert -1
Insert 73



Today's Outline

- Constant-Time Dictionaries?
- Hash Table Outline
- Hash Functions
- Collisions and the Pigeonhole Principle
- Collision Resolution:
  - Chaining
  - Open-Addressing

A Good Hash Function…

…is easy (fast) to compute (O(1) *and* practically fast).

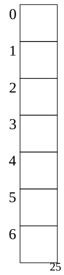
…distributes the data evenly (hash(a) ≠ hash(b), *probably*).

…uses the whole hash table (for all $0 \leq k <$ size, there's an i such that hash(i) % size = k).

Good Hash Function for Integers

- Choose
  - tableSize is
    - **prime** for good spread
    - **power of two** for fast calculations/convenient size
  - hash(n) = n (fast and good enough?)
- Example, tableSize = 7

  insert(4)
  insert(17)
  find(12)
  insert(9)
  delete(17)



25

Good Hash Function for Strings?

- Let s = $s_1 s_2 s_3 s_4 \ldots s_5$: choose
  - hash(s) = $s_1 + s_2 128 + s_3 128^2 + s_4 128^3 + \ldots + s_n 128^n$

    Think of the string as a base 128 number.

- Problems:
  - hash("really, really big") is really, really big!
  - hash("one thing") % 128 = hash("other thing") % 128

26

Making the String Hash Easy to Compute

- Use Horner's Rule (Qin's Rule?)

```
int hash(string s) {
  h = 0;
  for (i = s.length() - 1; i >= 0; i--) {
    h = (s_i + 31*h) % tableSize;
  }
  return h;
}
```

27

Hash Function Summary

- Goals of a hash function
  - reproducible mapping from key to table entry
  - evenly distribute keys across the table
  - separate commonly occurring keys (neighboring keys?)
  - complete quickly
- Sample hash functions:
  - h(n) = n % size
  - h(n) = string as base 31 number % size
  - *Multiplication hash: compute percentage through the table*
  - *Universal hash function #1: dot product with random vector*
  - *Universal hash function #2: next pseudo-random number*

28

How to Design a Hash Function

- Know what your keys are *or* Study how your keys are distributed.
- Try to include all important information in a key in the construction of its hash.
- Try to make "neighboring" keys hash to very different places.
- Balance complexity/runtime of the hash function against spread of keys (very application dependent).

29

Today's Outline

- Constant-Time Dictionaries?
- Hash Table Outline
- Hash Functions
- Collisions and the Pigeonhole Principle
- Collision Resolution:
  - Chaining
  - Open-Addressing

30

Collisions

- *Pigeonhole principle* says we can't avoid all collisions
  - try to hash without collision m keys into n slots with $m > n$
  - try to put 6 pigeons into 5 holes

31

The Pigeonhole Principle (informal)

You can't put k+1 pigeons into k holes without putting two pigeons in the same hole.

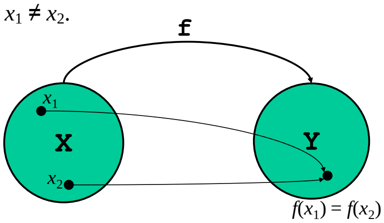
Image by en:User:McKay, used under CC attr/share-alike.

32

The Pigeonhole Principle (formal)

Let X and Y be finite sets where |X| > |Y|.

If f : X⟶Y, then f(x_1) = f(x_2) for some x_1, x_2 ∈ X, where $x_1 \neq x_2$.



33

The Pigeonhole Principle (Example #1)

Suppose we have 5 colours of Halloween candy, and that there's lots of candy in a bag. How many pieces of candy do we have to pull out of the bag if we want to be sure to get 2 of the same colour?

a. 2
b. 4
c. 6
d. 8
e. None of these

34

The Pigeonhole Principle (Example #2)

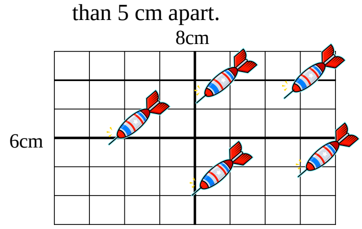
If there are 1000 pieces of each colour, how many do we need to pull to guarantee that we'll get 2 *black* pieces of candy (assuming that black is one of the 5 colours)?

a. 2
b. 4
c. 6
d. 8
e. None of these

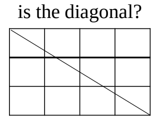
35

The Pigeonhole Principle (Example #3)

If 5 points are placed in a 6cm x 8cm rectangle, argue that there are two points that are not more than 5 cm apart.



Hint: How long is the diagonal?



36

The Pigeonhole Principle (Example #4)

For $a, b \in \mathbf{Z}$, we write *a divides b* as $a|b$, meaning $\exists\, c \in \mathbf{Z}$ such that $b = ac$.

Consider n +1 distinct positive integers, each $\leq 2n$. Show that one of them must divide on of the others.

For example, if $n = 4$, consider the following sets:

{1, 2, 3, 7, 8} {2, 3, 4, 7, 8} {2, 3, 5, 7, 8}

Hint: Any integer can be written as $2^k * q$ where k is an integer and q is odd. E.g., $129 = 2^0 * 129$; $60 = 2^2 * 15$.

The Pigeonhole Principle (Full Glory)

Let X and Y be finite sets with $|X| = n$, $|Y| = m$, and $k = \lceil n/m \rceil$.

If $f : X \rightarrow Y$, then $\exists\, k$ values $x_1, x_2, \ldots, x_k \in X$ such that $f(x_1) = f(x_2) = \ldots f(x_k)$.

Informally: If n pigeons fly into m holes, at least 1 hole contains at least $k = \lceil n/m \rceil$ pigeons.

Proof: Assume there's no such hole. Then, there are at most $(\lceil n/m \rceil - 1)*m$ pigeons in all the holes, which is fewer than $(n/m + 1 - 1)*m = n/m*m = n$, but that is a contradiction. QED

38

Today's Outline

- Constant-Time Dictionaries?
- Hash Table Outline
- Hash Functions
- Collisions and the Pigeonhole Principle
- Collision Resolution:
  - Chaining
  - Open-Addressing

39

Collision Resolution

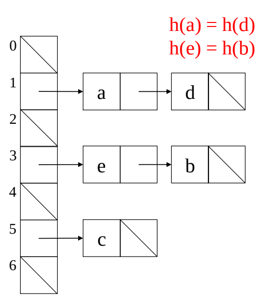
- *Pigeonhole principle* says we can't avoid all collisions
  - try to hash without collision m keys into n slots with $m > n$
  - try to put 6 pigeons into 5 holes
- What do we do when two keys hash to the same entry?
  - chaining: put little dictionaries in each entry

    shove extra pigeons in one hole!
  - open addressing: pick a next entry to try

40

Hashing with Chaining

- Put a little dictionary at each entry
  - choose type as appropriate
  - common case is unordered move-to-front linked list (chain)
- Properties
  - λ can be greater than 1
  - performance degrades with length of chains



41

Chaining Code

```
Dictionary & findBucket(const Key & k) {
  return table[hash(k)%table.size];
}

void insert(const Key & k,
            const Value & v)
{
  findBucket(k).insert(k,v);
}

void delete(const Key & k)
{
  findBucket(k).delete(k);
}

Value & find(const Key & k)
{
  return findBucket(k).find(k);
}
```

42

Load Factor in Chaining

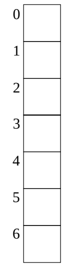
- Search cost
  - unsuccessful search:
  - successful search:
- Desired load factor:

43

Practice: Chaining

(Use a move-to-front list.)

Insert 2
Insert 5
Insert 10
Insert 73
Find 10
Insert 14
Insert -1
Insert 3



44

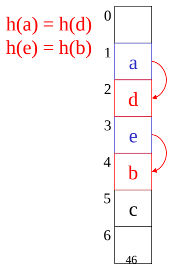
Today's Outline

- Constant-Time Dictionaries?
- Hash Table Outline
- Hash Functions
- Collisions and the Pigeonhole Principle
- Collision Resolution:
  - Chaining
  - Open-Addressing

45

Open Addressing

What if we only allow one Key at each entry?
  - two objects that hash to the same spot can't both go there
  - first one there gets the spot
  - next one must *go in another spot*
- Properties
  - $\lambda \leq 1$
  - performance degrades with difficulty of finding right spot



46

Probing

- Probing how to:
  - First probe - given a key k, hash to h(k)
  - Second probe - if h(k) is occupied, try h(k) + f(1)
  - Third probe - if h(k) + f(1) is occupied, try h(k) + f(2)
  - And so forth
- Probing properties
  - the i^{th} probe is to (h(k) + f(i)) mod size   where f(0) = 0
  - if i reaches size, the insert has failed
  - depending on f(), the insert may fail sooner
  - long sequences of probes are costly!

47

Linear Probing

f(i) = i

- Probe sequence is
  - h(k) mod size
  - h(k) + 1 mod size
  - h(k) + 2 mod size
  - …
- findEntry using linear probing:

```
bool findEntry(const Key & k, Entry *& entry) {
  int probePoint = hash1(k);
  do {
    entry = &table[probePoint];
    probePoint = (probePoint + 1) % size;
  } while (!entry->isEmpty() && entry->key != k);
  return !entry->isEmpty();
}
```

48

Linear Probing Example

	insert(76) 76%7 = 6	insert(93) 93%7 = 2	insert(40) 40%7 = 5	insert(47) 47%7 = 5	insert(10) 10%7 = 3	insert(55) 55%7 = 6
0				47	47	47
1						55
2		93	93	93	93	93
3					10	10
4						
5			40	40	40	40
6	76	76	76	76	76	76
probes:	1	1	1	3	1	3

49

Load Factor in Linear Probing

- For *any* $\lambda < 1$, linear probing will find an empty slot
- Search cost (for large table sizes)
  - successful search: $\frac{1}{2}\left(1+\frac{1}{(1-\lambda)}\right)$
  - unsuccessful search: $\frac{1}{2}\left(1+\frac{1}{(1-\lambda)^2}\right)$
- Linear probing suffers from *primary clustering*
- Performance quickly degrades for $\lambda > 1/2$

Values hashed close to each other probe the same slots.

50

Quadratic Probing

$f(i) = i^2$

- Probe sequence is
  - h(k) mod size
  - (h(k) + 1) mod size
  - (h(k) + 4) mod size
  - (h(k) + 9) mod size
  - …
- findEntry using quadratic probing:

```
bool findEntry(const Key & k, Entry *& entry) {
  int probePoint = hash1(k), numProbes = 0;
  do {
    entry = &table[probePoint];
    numProbes++;
    probePoint = (probePoint + 2*numProbes - 1) % size;
  } while (!entry->isEmpty() && entry->key != key);
  return !entry->isEmpty();
}
```

51

Quadratic Probing Example ☺

	insert(76) 76%7 = 6	insert(40) 40%7 = 5	insert(48) 48%7 = 6	insert(5) 5%7 = 5	insert(55) 55%7 = 6
0			48	47	47
1					
2				5	5
3					55
4					
5		40	40	40	40
6	76	76	76	76	76
probes:	1	1	2	3	3

52

Quadratic Probing Example ☹

	insert(76) 76%7 = 6	insert(93) 93%7 = 2	insert(40) 40%7 = 5	insert(35) 35%7 = 0	insert(47) 47%7 = 5
0				35	35
1					
2		93	93	93	93
3					
4					
5			40	40	40
6	76	76	76	76	76
probes:	1	1	1	1	

53

Quadratic Probing Succeeds (for $\lambda \le ½$)

- If size is prime and $\lambda \le ½$, then quadratic probing will find an empty slot in size/2 probes or fewer.
  - show for all `0 ≤ i, j ≤ size/2` and `i ≠ j`
    `(h(x) + i²) mod size ≠ (h(x) + j²) mod size`
  - by contradiction: suppose that for some i, j:
    `(h(x) + i²) mod size = (h(x) + j²) mod size`
    `i² mod size = j² mod size`
    `(i² - j²) mod size = 0`
    `[(i + j)(i - j)] mod size = 0`
  - but how can `i + j = 0` or `i + j = size` when `i ≠ j` and `i,j ≤ size/2`?
  - same for `i - j mod size = 0`

54

Quadratic Probing May Fail (for $\lambda > ½$)

- For any i larger than size/2, there is some j smaller than i that adds with i to equal size (or a multiple of size). D'oh!

Load Factor in Quadratic Probing

- For *any* $\lambda \leq ½$, quadratic probing will find an empty slot; for greater λ, quadratic probing *may* find a slot
- Quadratic probing does not suffer from primary clustering
- Quadratic probing *does* suffer from *secondary* clustering
  - How could we possibly solve this?

Values hashed to the SAME index probe the same slots.

Double Hashing

$f(i) = i \cdot hash_2(x)$

- Probe sequence is
  - $h_1(k)$ mod size
  - $(h_1(k) + 1 \cdot h_2(x))$ mod size
  - $(h_1(k) + 2 \cdot h_2(x))$ mod size
  - …
- Code for finding the next linear probe:

```
bool findEntry(const Key & k, Entry *& entry) {
  int probePoint = hash1(k), hashIncr = hash2(k);
  do {
    entry = &table[probePoint];
    probePoint = (probePoint + hashIncr) % size;
  } while (!entry->isEmpty() && entry->key != k);
  return !entry->isEmpty();
}
```


A Good Double Hash Function…

…is quick to evaluate.

…differs from the original hash function.

…never evaluates to 0 (mod size).

One good choice is to choose a prime R < size and:

$hash_2(x) = R - (x \bmod R)$

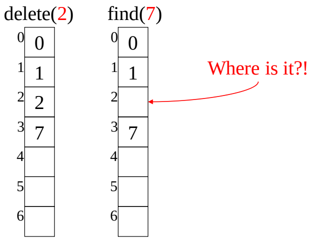
Double Hashing Example

	insert(76) 76%7 = 6	insert(93) 93%7 = 2	insert(40) 40%7 = 5	insert(47) 47%7 = 5, 5 - (47%5) = 3	insert(10) 10%7 = 3	insert(55) 55%7 = 6, 5 - (55%5) = 5
0						
1				47	47	47
2		93	93	93	93	93
3					10	10
4						55
5			40	40	40	40
6	76	76	76	76	76	76
probes:	1	1	1	2	1	2

Load Factor in Double Hashing

- For *any* $\lambda < 1$, double hashing will find an empty slot (given appropriate table size and $hash_2$)
- Search cost appears to approach optimal (random hash):
  - successful search: $\frac{1}{\lambda} \ln \frac{1}{1-\lambda}$
  - unsuccessful search: $\frac{1}{1-\lambda}$
- No primary clustering and no secondary clustering
- One extra hash calculation

Deletion in Open Addressing



- **Must** use lazy deletion!
- On insertion, treat a deleted item as an empty slot

The Squished Pigeon Principle

- An insert using open addressing *cannot* work with a load factor of 1 or more.
- An insert using open addressing with quadratic probing may not work with a load factor of ½ or more.
- Whether you use chaining or open addressing, large load factors lead to poor performance!
- How can we relieve the pressure on the pigeons?

Hint: think resizable arrays!

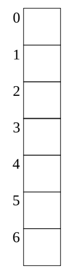
Rehashing

- When the load factor gets "too large" (over a constant threshold on λ), rehash all the elements into a new, larger table:
  - takes O(n), but amortized O(1) as long as we (just about) double table size on the resize
  - spreads keys back out, may drastically improve performance
  - gives us a chance to retune parameterized hash functions
  - avoids failure for open addressing techniques
  - allows arbitrarily large tables starting from a small table
  - clears out lazily deleted items

Practice: Open Addressing

(Try linear, quadratic, %7/(1-%5) double hashing.)

Insert 2
Insert 5
Insert 4
Insert 10
Insert 73
Find 10
Insert 14
Resize/Rehash
Insert -1
Insert 3



Coming Up

- Parallelism and/or Graphs

Extra Slides:
Some Other Hashing Methods

These are parameterized methods, which is handy if you **know** the keys in advance. In that case, you can randomly set the parameters a few times and pick a hash function that performs well. (Would that ever happen? How about when building a spell-check dictionary?)

Good Hashing: Multiplication Method

- Hash function is defined by size plus a parameter *A*
  $h_A(k) = \lfloor size * (k*A \bmod 1) \rfloor$ where $0 < A < 1$
- Example: size = 10, A = 0.485
  $h_A(50) = \lfloor 10 * (50*0.485 \bmod 1) \rfloor$
  $= \lfloor 10 * (24.25 \bmod 1) \rfloor = \lfloor 10 * 0.25 \rfloor = 2$
  - no restriction on size!
  - if we're building a static table, we can try several As
  - more computationally intensive than a single mod

Good Hashing: Universal Hash Function

- Parameterized by prime size and vector:
  $a = \langle a_0\ a_1 \ldots a_r \rangle$ where $0 \le a_i <$ size
- Represent each key as r + 1 integers where $k_i <$ size
  - size = 11, key = 39752 ==> <3,9,7,5,2>
  - size = 29, key = "hello world" ==> <8,5,12,12,15,23,15,18,12,4>

$$h_a(k) = \left(\sum_{i=0}^{r} a_i k_i \right) \bmod size$$

Universal Hash Function: Example

- Context: hash strings of length 3 in a table of size 131
  let a = <35, 100, 21>
  $h_a(\text{"xyz"}) = (35*120 + 100*121 + 21*122) \% 131$
  $= 129$

Universal Hash Function

- Strengths:
  - works on any type as long as you can form k_i's
  - if we're building a static table, we can try many a's
  - a random a has guaranteed good properties no matter what we're hashing
- Weaknesses
  - must choose prime table size larger than any k_i

Alternate Universal Hash Function

- Parameterized by k, a, and b:
  - k * size should fit into an int
  - a and b must be less than size

$$H_{k,a,b}(x) = ((a \cdot x + b) \bmod k \cdot size) / k$$

Alternate Universe Hash Function: Example

- Context: hash integers in a table of size 16
  let k = 32, a = 100, b = 200
  $h_{k,a,b}(1000) = ((100*1000 + 200) \% (32*16)) / 32$
  $= (100200 \% 512) / 32$
  $= 360 / 32$
  $= 11$

Universal Hash Function

- Strengths:
  - if we're building a static table, we can try many a's
  - random a,b has guaranteed good properties no matter what we're hashing
  - can choose any size table
  - very efficient if k and size are powers of 2
- Weaknesses
  - still need to turn non-integer keys into integers

73