CPSC 221: Data Structures
Hashing

Alan J. Hu

(Using mainly Steve Wolfman's Old Slides)

Learning Goals

After this unit, you should be able to:

- Define various forms of the pigeonhole principle; recognize and solve the specific types of counting and hashing problems to which they apply.
- Provide examples of the types of problems that can benefit from a hash data structure.
- Compare and contrast open addressing and chaining.
- Evaluate collision resolution policies.
- Describe the conditions under which hashing can degenerate from O(1) expected complexity to O(n).
- Identify the types of search problems that do not benefit from hashing (e.g. range searching) and explain why.
- Manipulate data in hash structures both irrespective of implementation and also within a given implementation.

Outline

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

Reminder: Dictionary ADT

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

insert

- brownies
  - tasty

find(wolf)

- wolf
  - the perfect mix of oomph and Scrabble value

- midterm
  - would be tastier with brownies
- prog-project
  - so painful… who invented templates?
- wolf
  - the perfect mix of oomph and Scrabble value

- 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 list	O(1)	O(n)	O(n)
• Sorted Array	O(n)	O(log n)	O(n)
• AVL Trees	O(log n)	O(log n)	O(log n)
• B+Trees	O(log n)	O(log n)	O(log n)
• …			

Implementations So Far

	insert	find	delete
• Unsorted list	O(1)	O(n)	O(n)
• Sorted Array	O(n)	O(log n)	O(n)
• AVL Trees	O(log n)	O(log n)	O(log n)
• B+Trees	O(log n)	O(log n)	O(log n)
• Array:	O(1)	O(1)	O(1)

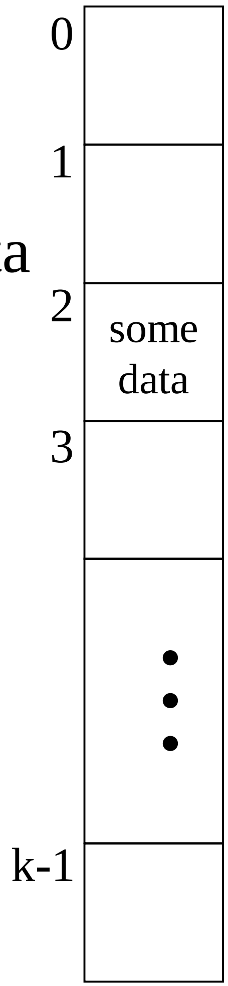
But only for the special case of integer keys between 0 and *size-1*

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

Hash Table Goal

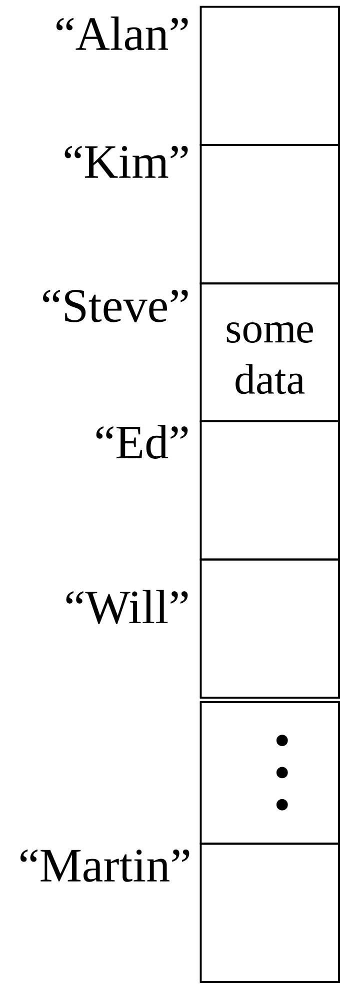
We can do:

a[2] = some data



We want to do:

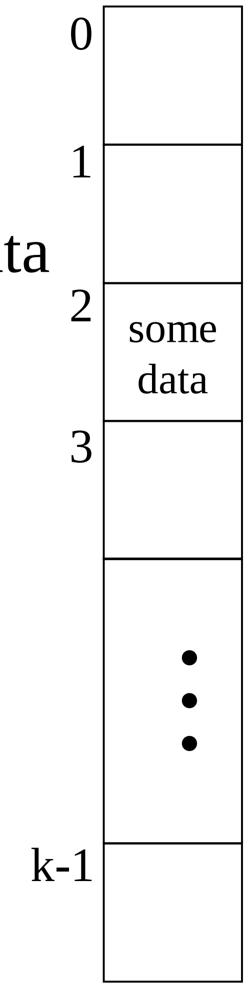
a[“Steve”] = some data



Aside: How do arrays do that?

We can do:

a[2] = some data



Q: If I know houses on a certain block in Vancouver are on 33-foot-wide lots, where is the 5th house?
A: It's from (5-1)*33 to 5*33 feet from the start of the block.

element_type a[SIZE];

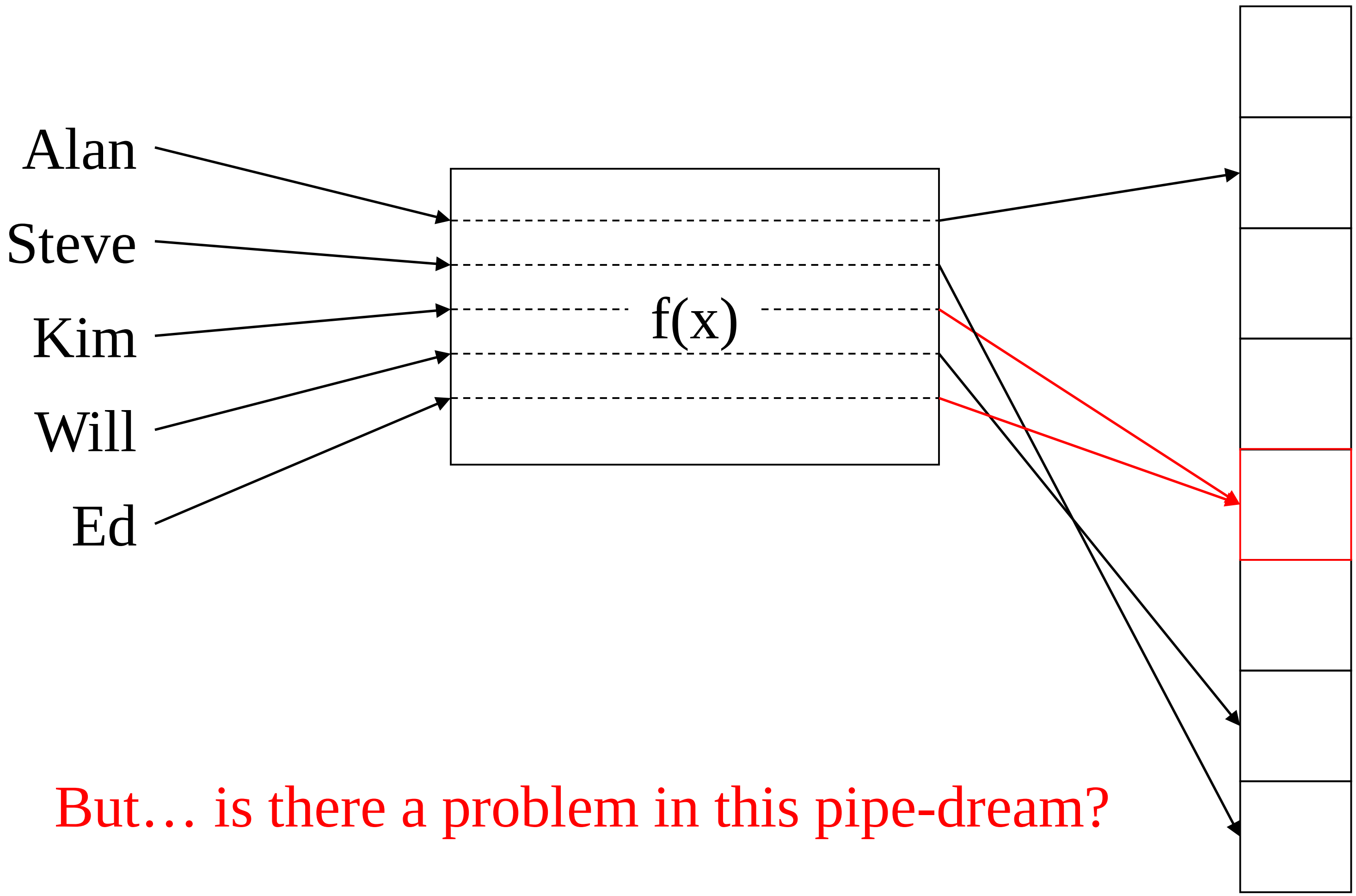
Q: Where is a[i]?
A: start of a + i*sizeof(element_type)

Aside: This is why array elements have to be the same size, and why we start the indices from 0.

Outline

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

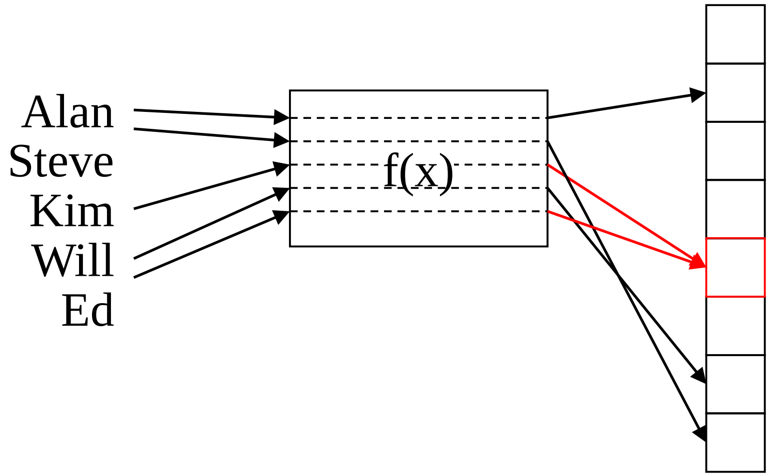
Hash Table Approach



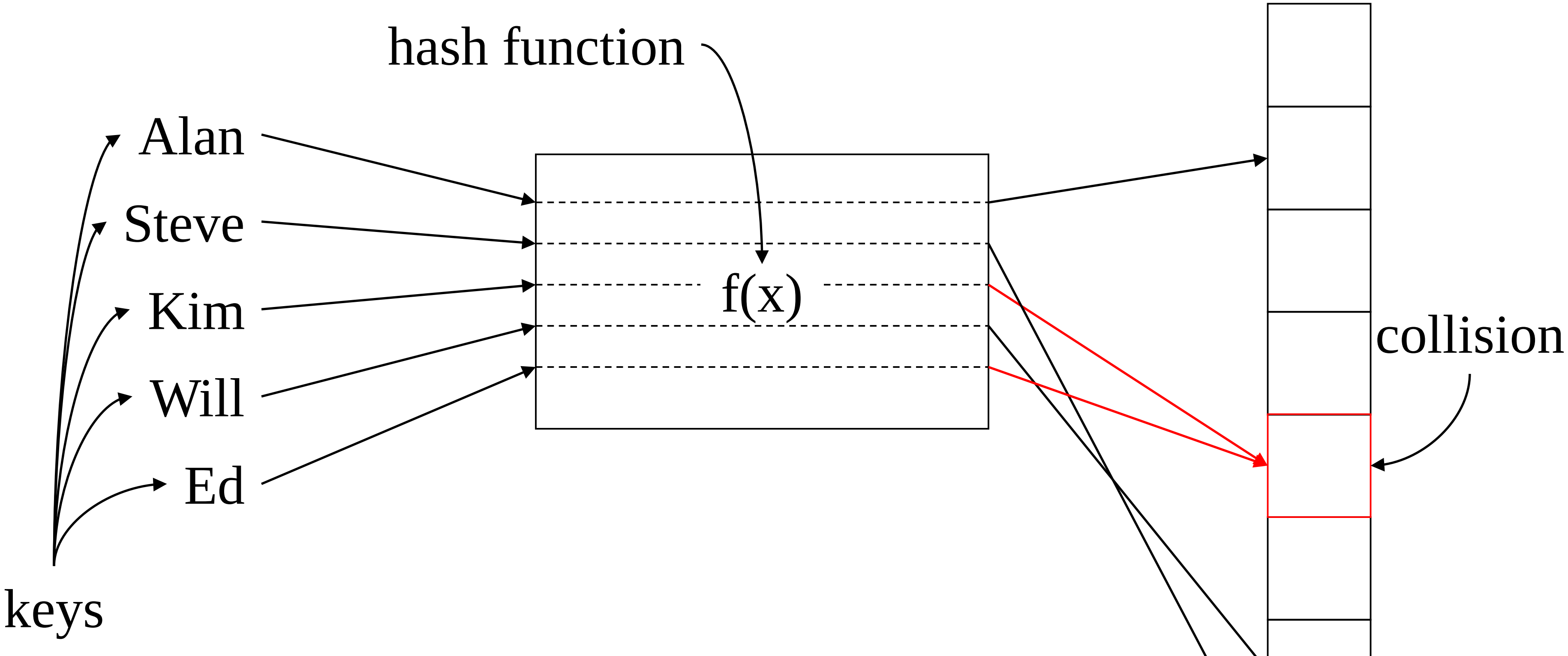
But… is there a problem in this pipe-dream?

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, ***definitely*** cannot efficiently list all entries in order or list entries between one value and another (a "range" query)



Hash Table Terminology



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

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?

Outline

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

A Good (Perfect?) Hash Function…

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

…distributes the data evenly (hash(a) % size ≠ hash(b) % size).

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

Aside: a Bit of 121 Theory

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

…distributes the data evenly (hash(a) % size ≠ hash(b) % size).

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

Onto (surjective)

Ideally, one-to-one (injective)

Good Hash Function for Integers

- Choose
  - tableSize is prime
  - hash(n) = n
- Example:
  - tableSize = 7

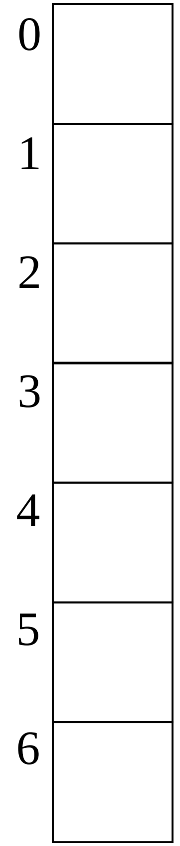
insert(4)

insert(17)

find(12)

insert(9)

delete(17)



Good Hash Function for Strings?

- Let $s = s_0 s_1 s_2 s_3 \ldots s_{n-1}$: choose
  - hash(s) = $s_0 + s_1 31 + s_2 31^2 + s_3 31^3 + \ldots + s_{n-1} 31^{n-1}$

Think of the string as a base 31 number.

- Problems:
  - hash(“really, really big”) = well… something really, really big
  - hash(“one thing”) % 31 = hash(“other thing”) % 31

Why 31? It’s prime. It’s **not** a power of 2. It works pretty well.

Making the String Hash Easy to Compute

- Use Horner’s Rule

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

Making the String Hash Cause Few Conflicts

- Ideas?

Making the String Hash Cause Few Conflicts

- Ideas?

  Make sure tableSize is not a multiple of 31.

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*

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.
- Prune the features used to create the hash until it runs “fast enough” (application dependent).

How to Design a Hash Function

- Know what your keys are *or*

In real life, use a standard hash function that people have already shown works well in practice!

different places.

- Prune the features used to create the hash until it runs “fast enough” (application dependent).

Extra Slides:
Some Other Hashing Methods

Good Hashing: Multiplication Method

- Hash function is defined by some positive number *A*

  $h_A(k) = (A * k) \% size$

- Example: A = 7, size = 10

  $h_A(50) = 7*50 \bmod 10 = 350 \bmod 10 = 0$

  - choose A to be relatively prime to size
  - more computationally intensive than a single mod
  - (This is simplified from a more general, theoretical case.)

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(“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
  - slower to compute than simpler hash functions

Alan's Aside: Bit-Level 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

Use the bits of the key!

Can use a power of 2

Good Hashing: Bit-Level 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 bits

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

Alternate Universal Hash Function

- Parameterized by p, a, and b:
  - p is a big prime (several times bigger than table size)
  - a and b are arbitrary integers in [1,p-1]

$$H_{p,a,b}(x) = \left(a \cdot x + b\right) \bmod p$$

Outline

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

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.

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

Collisions

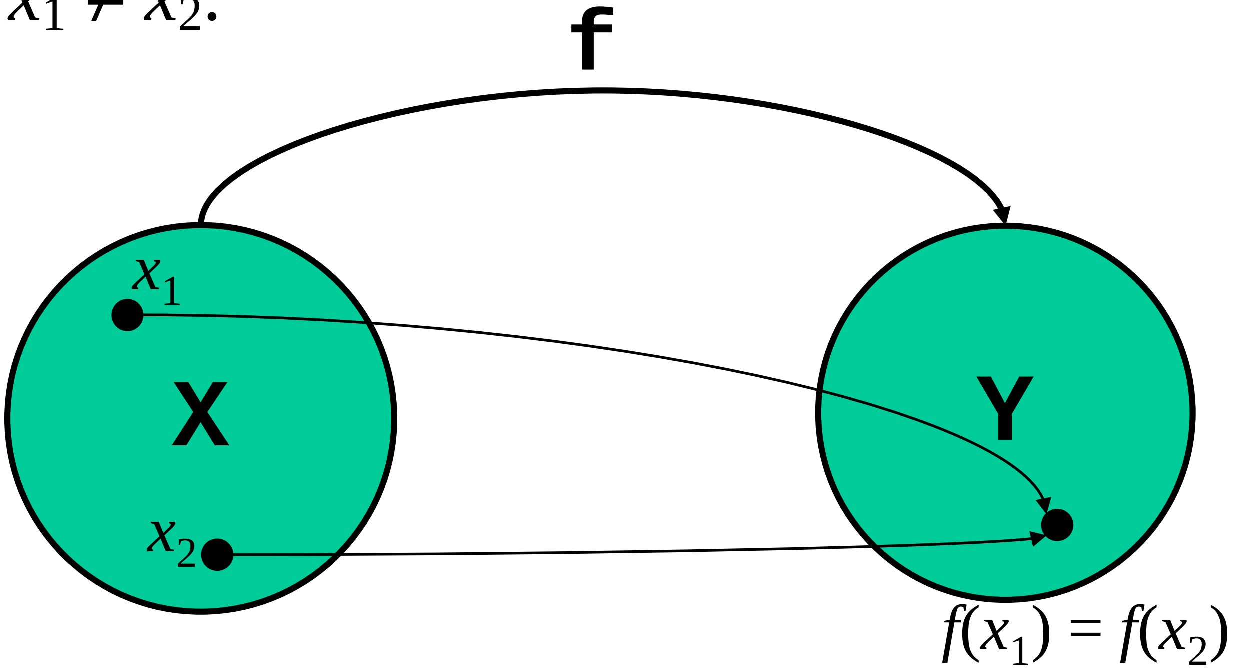
- *Pigeonhole principle* says we can’t avoid all collisions
  - try to hash without collision m keys into n slots with $m > n$
  - try to put 6 pigeons into 5 holes

Alan’s Aside: This is actually somewhat misleading.
Collisions are a problem even when $m < n$.
So this tie-in of collisions and the pigeonhole principle isn’t really fundamental. It’s just a nice chance to introduce the pigeonhole principle…

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 in X, where $x_1 \neq x_2$.



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

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. 6
c. 4002
d. 5001
e. None of these

The Pigeonhole Principle (No!) (Example #2)

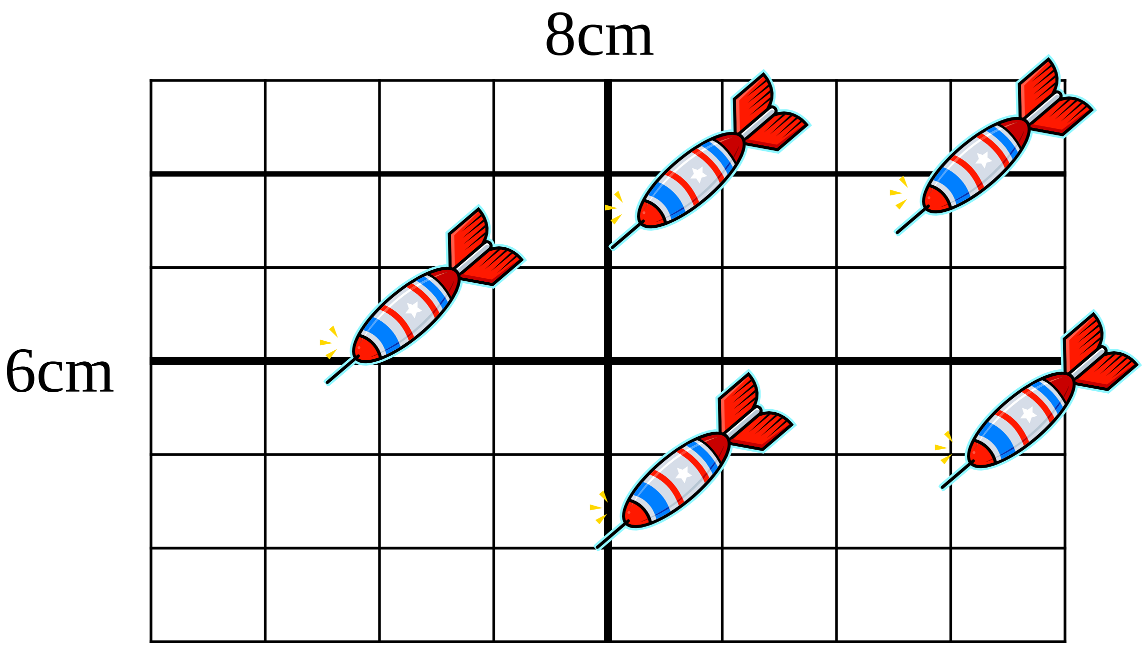
If there are 1000 pieces of each colour, how many do we need to pull to guarantee that we'll get 2 *black* pieces of candy (assuming that black is one of the 5 colours)?

a. 2
b. 6
c. 4002
d. 5001
e. None of these

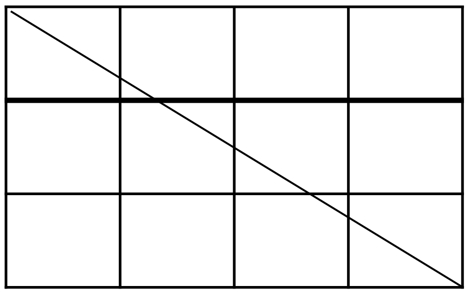
The PhP doesn't tell us *which* hole has two pigeons.

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?



The Pigeonhole Principle (Example #4)

For integers *a*, *b*, we write *a divides b* as $a|b$, meaning there exists integer *c* such that $b = ac$.

Consider n +1 distinct positive integers, each $\leq 2n$. Show that one of them must divide one 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 $q*2^k$ where k is a non-negative 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 there exist 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

Outline

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

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

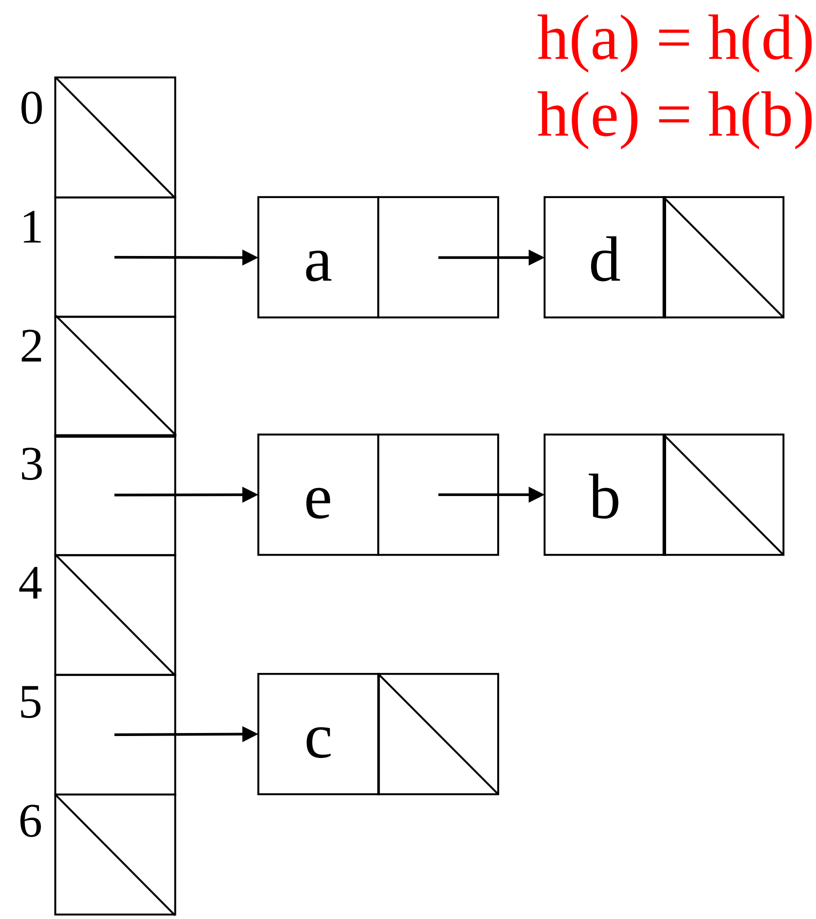
(Alan Aside) Collision Resolution

- *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 (AKA open hashing or closed addressing): put little dictionaries in each entry

    shove extra pigeons in one hole!
  - open addressing (AKA closed hashing): pick a next entry to try

Hashing with Chaining

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



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);
}
```

Load Factor in Chaining

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

Outline

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

Open Addressing / Closed Hashing

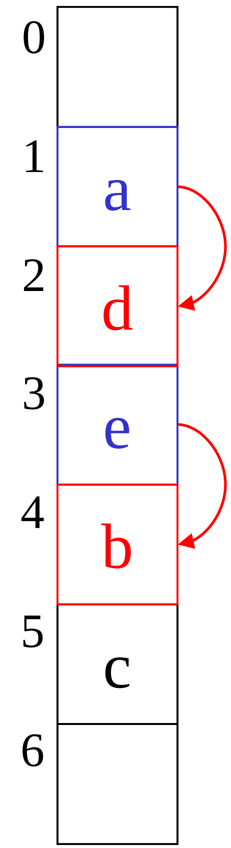
What if we only allow one key at each entry?

- 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

$h(a) = h(d)$

$h(e) = h(b)$



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!

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);
  int i=0;
  do {
    entry = &table[(probePoint+(i++)) % size];
  } while (!entry->isEmpty() && entry->key != k);
  return !entry->isEmpty();
}
```

Linear Probing (More Efficient Code)

$f(i) = i$

- 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();
}
```

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

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: $$\frac{1}{2}\left(1+\frac{1}{(1-\lambda)^2}\right)$$

Values hashed close to each other probe the same slots.

- Linear probing suffers from *primary clustering*
- Performance quickly degrades for $\lambda > 1/2$

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), i = 0;
  do {
    entry = &table[(probePoint + i*i) % size];
    i++;
  } while (!entry->isEmpty() && entry->key != key);
  return !entry->isEmpty();
}
```

Quadratic Probing (more efficient code)

$f(i) = i^2$

- 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), i = 0;
  do {
    entry = &table[probePoint];
    i++;
    probePoint = (probePoint + 2*i - 1) % size;
  } while (!entry->isEmpty() && entry->key != key);
  return !entry->isEmpty();
}
```

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

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	∞

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

- If size is prime and $\lambda \leq ½$, 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`**

    $$(h(x) + i^2) \bmod size \neq (h(x) + j^2) \bmod size$$
  - `this means that the size/2 probes must all land in different places, so at least one must succeed if` $\lambda \leq ½$

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

- If size is prime and $\lambda \leq ½$, 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`**

    $(h(x) + i^2) \bmod size \neq (h(x) + j^2) \bmod size$
  - by contradiction: suppose that for some i, j:

    $(h(x) + i^2) \bmod size = (h(x) + j^2) \bmod size$

    $i^2 \bmod size = j^2 \bmod size$

    $(i^2 - j^2) \bmod size = 0$

    $[(i + j)(i - j)] \bmod size = 0$
  - 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`**

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* $\lambda \leq \frac{1}{2}$, 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

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

- Probe sequence is
  - $h_1(k)$ mod size
  - $(h_1(k) + 1 \cdot h_2(k))$ mod size
  - $(h_1(k) + 2 \cdot h_2(k))$ 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:

$$\text{hash}_2(x) = R - (x \bmod R)$$

Double Hashing Example

	insert(76)	insert(93)	insert(40)	insert(47)	insert(10)	insert(55)
	76%7 = 6	93%7 = 2	40%7 = 5	47%7 = 5 5 - (47%5) = 3	10%7 = 3	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

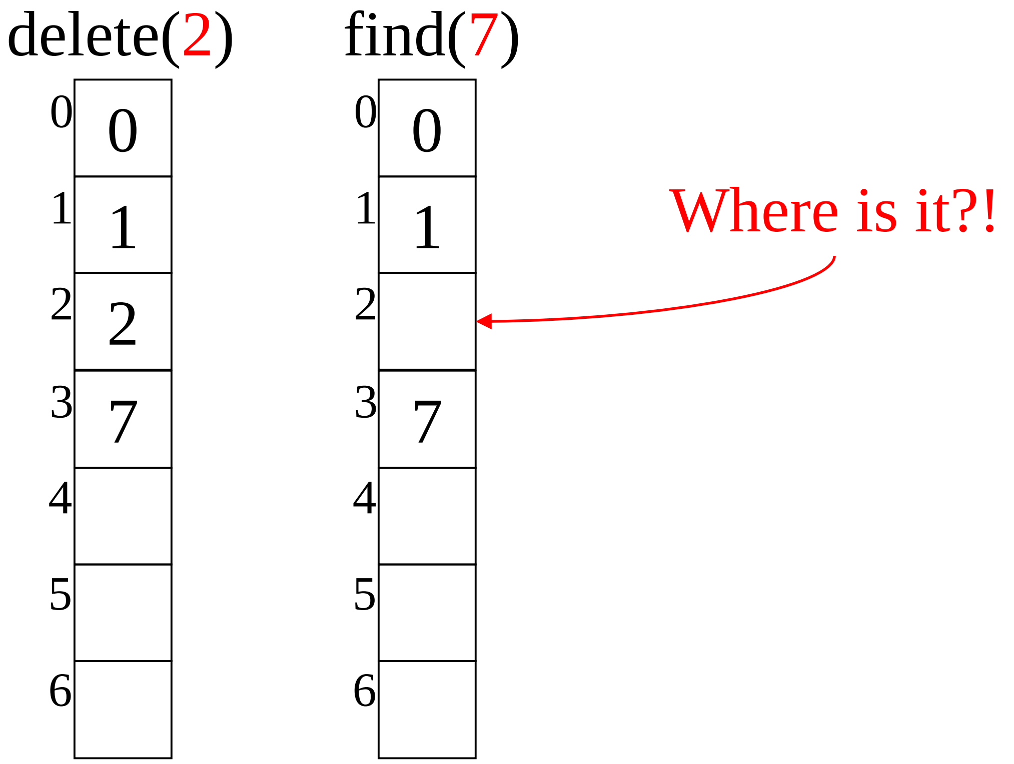
Load Factor in Double Hashing

- 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

Outline

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

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