Unit #5: Hash Functions and the Pigeonhole Principle

CPSC 221: Basic Algorithms and Data Structures

Anthony Estey, Ed Knorr, and Mehrdad Oveisi

2016W2

Annotated Slides from Ed's Class

Unit Outline

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

Learning Goals

- Provide examples of the types of problems that can benefit from a hash data structure.
- Identify the types of search problems that do not benefit from hashing (e.g., range searching) and explain why.
- Evaluate collision resolution policies.
- Compare and contrast open addressing and chaining.
- Describe the conditions under which `find` using a hash table takes $\Omega(n)$ time.
- Describe and apply `insert`, `delete`, and `find` operations using various open addressing and chaining schemes.
- Define various forms of the pigeonhole principle; and recognize and solve the specific types of counting and hashing problems to which they apply.

Review: Dictionary ADT

Dictionary operations

- create
- destroy
- insert
- find
- delete

key	value
Multics	MULTiplexed Information and Computing Service
Unics	single-user Multics
Unix	multi-user Unics
GNU	GNU's Not Unix

- insert(Linux, Linus Torvald's Unix)
- find(Unix)

Purpose: Store *values* associated with user-specified *keys*.

Hash Table Goal

We can do:

a[2]= "GNU's Not Unix"

0	
1	
2	GNU's Not Unix
3	
	⋮
$m-1$	

We want to do:

a["GNU"]= "GNU's Not Unix"

Multics	
Linux	
GNU	GNU's Not Unix
Unix	
	⋮
Unics	

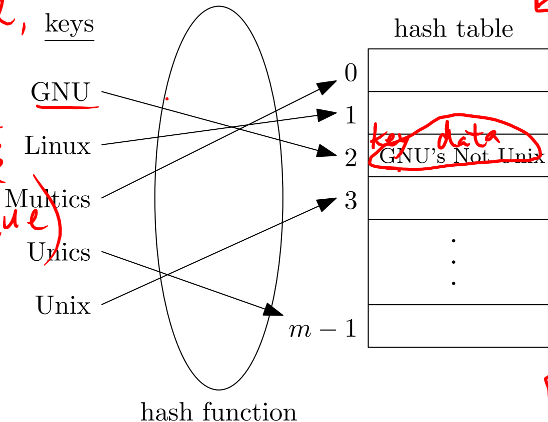
key

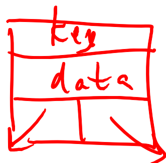
key

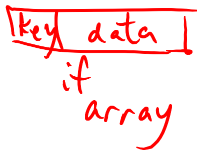
value

Hash Table Approach

Choose a **hash function** to map keys to indices.



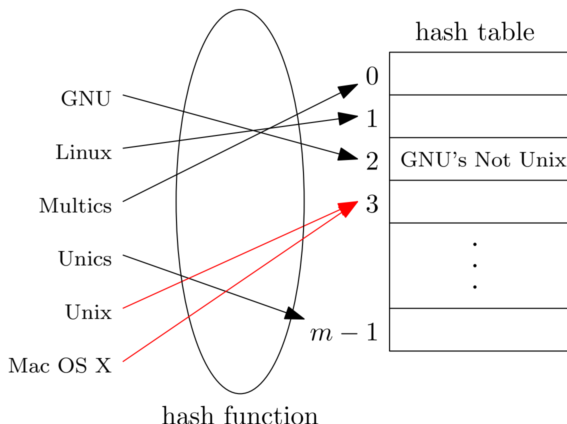




hash("GNU") $= 2$

Collisions

A **collision** occurs when two different keys x and y map to the same index (**slot**) in the hash table, i.e., $\text{hash}(x) = \text{hash}(y)$.



Any suggestions on how we can try to prevent collisions?

Simple, Naïve Hash Table Code

```
void insert(const Key & key ) {
  int index = hash(key) % m;
  HashTable[index] = key;
}

Value & find(const Key & key ) {
  int index = hash(key) % m;
  return HashTable[index];
}
```

What should the hash function, hash, be?

What should the table size, m, be?

What do we do about collisions?

see following slides

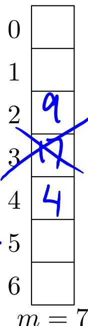
Good Hash Function Properties

Using knowledge of the kind and number of keys to be stored, we should choose our hash function so that it is:

- fast to compute, and
- causes few collisions (we hope).

Numeric keys We might use $\text{hash}(x) = x \bmod m$ with m larger than the number of keys we expect to store.

Example: $\text{hash}(x) = x \bmod 7$
insert(4)
insert(17) 17 % 7 = 3
find(12) 12 % 7 = 5 not there
insert(9)
delete(17)



if it were there, you'd still have to verify the key

Hashing String Keys

One option

Let string $s = s_0 s_1 s_2 \ldots s_{k-1}$ where each s_i is an 8-bit character.

$$\text{hash}(s) = s_0 + 256 s_1 + 256^2 s_2 + \cdots + 256^{k-1} s_{k-1}$$

This hash function treats the string as a base 256 number.

Problems

- hash("really, really big") = well... something really, really big
- hash("anything") mod 256 = hash("anything else") mod 256

Hashing String Keys Using *mod* and Horner's Rule

```
int hash( string s ) {
  int h = 0;
  for (i = s.length() - 1; i >= 0; i--) {
    h = (256 * h + s[i]) % m;
  }
  return h;
}
```

Compare that to the hash function from yacc:

```
#define TABLE_SIZE 1024 // must be a power of 2
int hash( char *s ) {
  int h = *s++;
  while( *s ) h = (31 * h + *s++) & (TABLE_SIZE - 1);
  return h;
}
```

What's different?

Hash Function Summary

Goals of a hash function

- Should be fast to compute O(1) expected
- Should cause few collisions generally want x & y to map to different hash table locations

Sample hash functions

- For numeric keys x, $\text{hash}(x) = x \bmod m$
- $\text{hash}(s) =$ string as base 256 number mod m
- Multiplicative hash: $\text{hash}(k) = \lfloor m \cdot \text{frac}(ka) \rfloor$ where $\text{frac}(x)$ is the fractional part of x and $a = 0.6180339887$ (for example).

- infinite #

on the first try
- not likely in general

Fixed Hash Functions are Dangerous

e.g., insert n keys

Good hash table performance depends on having few collisions.

worst-case: $O(n^2)$

If a user knows your hash function, she can cause many elements to hash to the same slot. Why would she want to do that?

Denial of Service

$1 + 2 + 3 + \dots + n \in O(n^2)$

Yacc hashes "XY" and "xy" to 769. How can you find many strings that yacc hashes to the same slot?

$\{XY, xy\}^k \; \forall k \in \mathbb{Z}^+$

first

Protection

- Choose a new hash function at random for every hash table.

all hit the same slot

- Use a cryptographically secure hash function, such as, Secure Hash Algorithm SHA-256 which results in hash values that are 256 bits long (i.e., 32 bytes).

Universal Hash Functions

A set $\mathcal{H}$ of hash functions is *universal* if the probability that $\text{hash}(x) = \text{hash}(y)$ is at most $1/m$ when hash() is chosen at random from $\mathcal{H}$.

Example: Let p be a prime number larger than any key. Choose a at random from $\{1, 2, \ldots, p-1\}$ and choose b at random from $\{0, 1, \ldots, p-1\}$.

$$\text{hash}(x) = ((a \cdot x + b) \bmod p) \bmod m$$

General Form of Hash Functions

number, string, matrix (10 x 10), image

1. Map the key to a sequence of bytes.
   - Two equal sequences occur iff the keys are equal.
   - Well, this is easy: the key probably is a sequence of bytes already.
2. Map the sequence of bytes to an integer x. hash code map (big range)
   - Changing even one byte should cause apparently **random** changes to x.
   - This is hard. It may be expensive (e.g., using a cryptographic hash function).
3. Map x to a table index using $x \bmod m$. compression map

only m spots in hash table

- want a good uniform distribution of keys

Collisions

Birthday Paradox

With probability > ______, two people, in a room of 23, have the same birthday. (Hash 23 people into $m = 365$ slots. Collision?)

General Birthday Paradox

If we *randomly* hash $\sqrt{2m}$ keys into m slots, we get a collision with probability > ______.

Collision

Unless we know all the keys in advance and design a perfect hash function, we must handle collisions.

What do we do when two keys hash to the same slot?

- In Separate Chaining: Store multiple items in each slot (e.g., via a chain of "overflow" elements)
- In Open Addressing: Pick a next slot to try

Hashing with Chaining

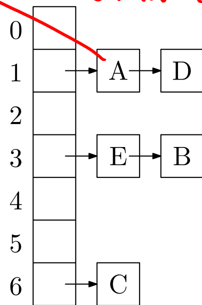
Store multiple items in each slot.

How?

- Common choice is an unordered linked list (a chain)
- Could use any dictionary ADT implementation

Result

- Can hash more than m items into a table of size m
- Performance depends on the length of the chains.
- Memory is allocated for each insertion.



$\text{hash}(A) = \text{hash}(D) = 1$
$\text{hash}(E) = \text{hash}(B) = 3$

Access Time for Chaining

Load Factor

$$\alpha = \frac{\#\text{ hashed items}}{\text{table size}} = \frac{n}{m}$$

Assume we have a uniform hash function (every item hashes to a uniformly distributed slot).

Search cost

On average,

- an unsuccessful search examines α items.
- a successful search examines $1 + \frac{n-1}{2m} = 1 + \frac{\alpha}{2} - \frac{\alpha}{2n}$ items.

We want the load factor to be small.

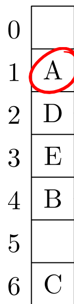
Open Addressing — no chaining

Allow only one item in each slot. The hash function specifies a *sequence* of slots to try.

Insert If the first slot is occupied, try the next, then the next, ... until an empty slot is found.

Find If the first slot doesn't match, try the next, then the next, ... until a match is found or an empty slot is reached (i.e., the key cannot be found).

suppose h(A) = 1
h(D) = 1



linear collision resolution policy

Result

- We cannot hash more than m items into a table of size m. [Pigeonhole Principle]
- Hash table memory is allocated once.
- Performance depends on the number of tries.

if ∃ 4 birds & only 3 nests ⇒ ≥ 2 birds per

Probe Sequence

The probe sequence is the sequence of slots that we examine when inserting (and finding) a key.

A probe sequence is a function $h(k, i)$ that maps a key k and an integer i to a table index.
Given key k:

- We first examine slot $h(k, 0)$.
- If it's full, we examine slot $h(k, 1)$.
- If it's full, we examine slot $h(k, 2)$.
- And so on ...

- try next cell
- if still collision, try next cell
etc.

If all the slots in the probe sequence are full, then we fail to insert the key.
The "time" to insert the key is the number of slots we must examine before finding an empty slot.

Linear Probing: $h(k, i) = (\text{hash}(k) + i) \bmod m$

Key's corresponding data

ref

```
Entry *find( const Key & k ) {
  int p = hash(k) % size;
  for( int i=1; i<=size; i++ ) {
    Entry *entry = &(table[p]);
    if( entry->isEmpty() ) return NULL;
    if( entry->key == k ) return entry;
    p = (p + 1) % size;
  }
  return NULL;
}
```

table location

not found

test it because it might not be the key you're looking for

Linear Probing 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	10%7 = 3	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 •

Access Time for Linear Probing

If $\alpha < 1$, linear probing will find an empty slot.

Linear probing suffers from **primary clustering**: the creation of long, consecutive sequences of filled slots. (They tend to get longer, and merge.)

Performance quickly degrades for $\alpha > 1/2$.

α	0.25	0.5	0.75	0.9
unsucc.	1.4	2.5	8.5	50.5
succ.	1.2	1.5	2.5	5.5

succ. $\longrightarrow \approx \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$

unsuccessful search

$$\approx \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right) \text{ slots}$$

$= 50.5$ when $\alpha = 0.9$

independent of m

Quadratic Probing: $h(k, i) = (\text{hash}(k) + i^2) \bmod m$

(assume unique keys for this Unit on hashing)

```
Entry *find( const Key & k ) {
  int p = hash(k) % size;
  for( int i=1; i<=size; i++ ) {
    Entry *entry = &(table[p]);
    if( entry->isEmpty() ) return NULL;
    if( entry->key == k ) return entry;
    p = (p + 2*i - 1) % size;
  }
  return NULL;
}
```

$i^2 = (i-1)^2 + 2i - 1$

try hash(k) first; if unsuccessful, try $[\text{hash}(k) + 1^2] \% m$

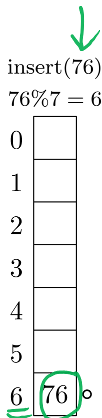
... then $[\text{hash}(k) + 2^2] \% m$

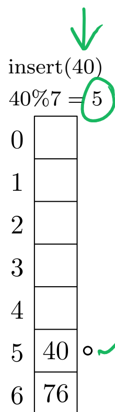
... then $[\text{hash}(k) + 3^2] \% m$

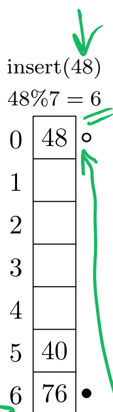
...

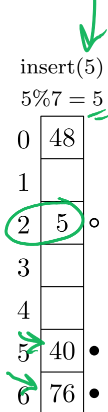
Quadratic Probing Example

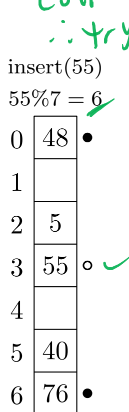
collision ∴ try 1^2 case
collision ∴ try 2^2 case











occupied
∴ $(6+1^2)\%7=0$

Quadratic Probing Example

	insert(76)	insert(93)	insert(40)	insert(35)	insert(47)
	$76\%7 = 6$	$93\%7 = 2$	$40\%7 = 5$	$35\%7 = 0$	$47\%7 = 5$
0				35 ∘	35 •
1					
2		93 ∘	93	93	93 •
3					
4					
5			40 ∘	40	40 •
6	76 ∘	76	76	76	76 •
					fail

can't find an empty slot

Quadratic Probing: First $\lceil m/2 \rceil$ Probes are Distinct

Claim: If m is prime, the first $\lceil m/2 \rceil$ probes are distinct.

Proof: (by contradiction) Suppose for some $0 \leq i < j \leq \lfloor m/2 \rfloor$,

$$
\begin{aligned}
& & (\text{hash}(k) + i^2) \bmod m &= (\text{hash}(k) + j^2) \bmod m \\
&\Leftrightarrow & i^2 \bmod m &= j^2 \bmod m \\
&\Leftrightarrow & (i^2 - j^2) \bmod m &= 0 \\
&\Leftrightarrow & (i - j)(i + j) \bmod m &= 0
\end{aligned}
$$

Since m is prime, one of $(i - j)$ and $(i + j)$ must be divisible by m. But $0 < i + j < m$ and $-\lfloor m/2 \rfloor \leq i - j < 0$ because $0 \leq i < j \leq \lfloor m/2 \rfloor$.

Result

If table size m is prime and there are $< \lceil m/2 \rceil$ full slots (i.e., $\alpha < 1/2$), then quadratic probing will find an empty slot.

Quadratic Probing: Only $\lceil m/2 \rceil$ Probes Are Distinct

Claim: For any $j \in \{\lceil m/2 \rceil, \lceil m/2 \rceil + 1, \ldots, m-1\}$, there is an $i \in \{1, 2, \ldots, \lfloor m/2 \rfloor\}$ such that $i^2 \bmod m = j^2 \bmod m$.

Proof: Let $i = m - j$.

$$i^2 = (m-j)^2 = m^2 - 2mj + j^2 = j^2 \mod m.$$

For example: $m = 7$

$$\begin{aligned}
\text{hash}(k) + 0^2 &= \text{hash}(k) + 0 \mod 7\\
\text{hash}(k) + 1^2 &= \text{hash}(k) + 1 \mod 7\\
\text{hash}(k) + 2^2 &= \text{hash}(k) + 4 \mod 7\\
\text{hash}(k) + 3^2 &= \text{hash}(k) + 2 \mod 7\\
\text{hash}(k) + 4^2 &= \text{hash}(k) + 2 \mod 7\\
\text{hash}(k) + 5^2 &= \text{hash}(k) + 4 \mod 7\\
\text{hash}(k) + 6^2 &= \text{hash}(k) + 1 \mod 7
\end{aligned}$$

Access Time for Quadratic Probing

Only the first $\lceil m/2 \rceil$ slots in a quadratic probe sequence are distinct — the rest are duplicates.

Quadratic probing doesn't suffer from primary clustering.

Quadratic probing suffers from **secondary clustering**: all items that initially hash to the same slot follow that same probe sequence.

less likely?

How could we avoid that?

- use a different hash function (i.e., 2 hash functions in all) then if a collision occurs with the first, it's unlikely to collide with second

Double Hashing: $h(k, i) = (\text{hash}(k) + i \cdot \text{hash}_2(k)) \bmod m$

```
Entry *find( const Key & k ) {
  int p = hash(k) % size, inc = hash2(k);
  for( int i=1; i<=size; i++ ) {
    Entry *entry = &(table[p]);
    if( entry->isEmpty() ) return NULL;
    if( entry->key == k ) return entry;
    p = (p + inc) % size;
  }
  return NULL;
}
```

Secondary hash function

if the second hash function collides, try i=2 ... 3 ... 4

Choosing $\text{hash}_2(k)$

$\text{hash}_2(k)$ should:

- be quick to evaluate $O(1)$
- differ from $\text{hash}(k)$
- never be 0 (mod m) make the result 1 if the sec. function = 0

We'll use:

$$\text{hash}_2(k) = r - (k \bmod r) \quad \text{this} \neq 0$$

for a prime number $r < m$.

Double Hashing Example

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

primary hash function

secondary hash function

occupied

$(5+3)\%7 = 1$

$(6+5)\%7 = 4$

Access Time for Double Hashing

For $\alpha < 1$, double hashing will find an empty slot (assuming *m* and hash_2 are well-chosen).

No primary or secondary clustering.

One extra hash calculation. $O(1)$

α = load factor

unsuccessful search $\frac{1}{1-\alpha}$ tries

successful search $\frac{1}{\alpha} \ln\left(\frac{1}{1-\alpha}\right)$

Searches:

α	0.25	0.5	0.75	0.9
unsucc.	1.3	2.0	4.0	10.0
succ.	1.2	1.4	1.8	2.6

Deletion in Open Addressing

Example: $\text{hash}(k) = k \bmod 7$

delete(2)

0	0
1	1
2	2
3	7
4	
5	
6	

find(7)

$7 \% 7 = 0$

0	0	← not here
1	1	← not here
2		← end of search?!
3	7	
4		
5		
6		

Put a tombstone in the slot.

Find Treat the tombstone as an occupied slot.

Insert Treat the tombstone as an empty slot.

However, you need to Find before you Insert because you want to avoid duplicate keys.

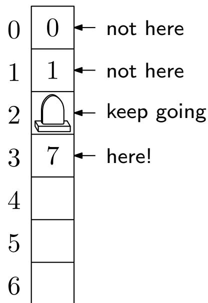
Deletion in Open Addressing

Example: hash(k) = k mod 7

delete(2)

0	0
1	1
2	2
3	7
4	
5	
6	

find(7)



Put a tombstone in the slot.

Find Treat the tombstone as an occupied slot.

Insert Treat the tombstone as an empty slot.

However, you need to Find before you Insert because you want to avoid duplicate keys.

Resizable Hash Tables

An insert using open addressing cannot succeed with a load factor of 1 or more. [Pigeonhole Principle]

An insert using open addressing with quadratic probing may not succeed with a load factor $> 1/2$.

Whether you use chaining or open addressing, large load factors lead to poor performance!

Hint: Think resizable arrays!

Rehashing

When the load factor gets "too large" ($\alpha >$ some constant threshold), rehash all the elements into a new, larger table:

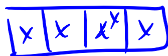
- takes $\Theta(m)$ time, but amortized $O(1)$ as long as we double table size on the resize ↳ but not good in a DBMS
- spreads keys back out, may drastically improve performance
- gives us a chance to change the hash function
- avoids failure for open addressing techniques
- allows arbitrarily large tables starting from a small table
- clears out tombstones

The Pigeonhole Principle

informal version

If more than m (m+1) pigeons fly into m pigeonholes then some pigeonhole contains at least two pigeons.

$m+1=5$
$m=4$



Corollary

If we hash $n > m$ keys into m slots, two keys will collide.

n: # of keys

m: size of table

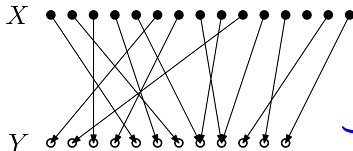
The Pigeonhole Principle

more formally

# of elements in set X

Let X and Y be finite sets where $|X| > |Y|$.
If $f : X \to Y$, then $f(x_1) = f(x_2)$ for some $x_1 \neq x_2$.

collision



function is "onto" (i.e., it's "surjective") but not injective

The Pigeonhole Principle: Example #1

1000 of each colour including green

Suppose we have 5 colours of Halloween candy, and there is a lot of candy in the 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?

guarantee

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

$6 > 5$

birds nests

What if you wanted a guarantee of getting 2 green candies?

4002

The Pigeonhole Principle: Example #2

Compression

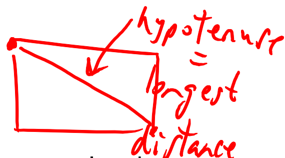
Any lossless compression algorithm (such as `zip`, `bzip2`, Huffman coding, Sequitur, etc.) will fail to compress some file.

How many files containing n bits are there?

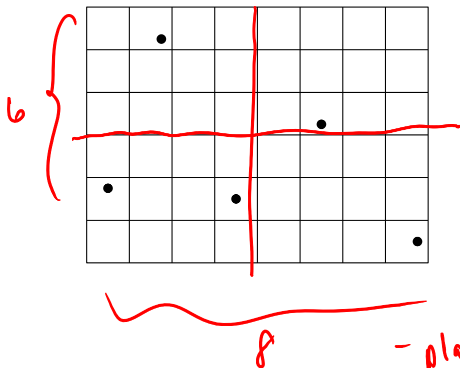
How many files containing fewer than n bits are there?

What are the pigeons and pigeonholes?

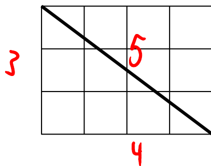
The Pigeonhole Principle: Example #3



Claim: If 5 points are placed in a 6 cm x 8 cm rectangle, there are two points that are ≤ 5 cm apart.



Hint: How long is this diagonal?



5 points, 4 quadrants

- place 4 points anywhere (worst case = corner)
- 5th point must be ≤ 5 cm

The Pigeonhole Principle: Example #4

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:

$$\{1, 2, 3, 7, 8\} \quad \{2, 3, 4, 7, 8\} \quad \{2, 3, 5, 7, 8\}$$

Hint: Any integer can be written as $2^k \cdot q$ where k is an integer and q is odd. For example, $129 = 2^0 \cdot 129$; and $60 = 2^2 \cdot 15$.

General Pigeonhole Principle

Let X and Y be finite sets with $|X| = n$, $|Y| = m$, and $k = \lceil n/m \rceil$. If $f : X \to Y$ then there exist k distinct values $x_1, x_2, \ldots, x_k \in X$ such that $f(x_1) = f(x_2) = \cdots = f(x_k)$.

Informally: If n pigeons fly into m pigeonholes, at least one pigeonhole contains at least $k = \lceil n/m \rceil$ pigeons.

e.g., $k = \left\lceil \frac{100}{6} \right\rceil = 17$

Proof: Assume there's no such pigeonhole. Then, there are at most $(\lceil n/m \rceil - 1)\, m < (n/m)m = n$ pigeons.

Pigeonhole Principle: Example #5

Ramsey's Theorem

In any group of 6 people, where each two people are either friends or enemies (i.e., they can't be "neutral"), there must be either 3 pairwise friends or 3 pairwise enemies.

Proof: Let A be one of the 6 people. A has at least 3 friends or at least 3 enemies by the General Pigeonhole Principle because $\lceil 5/2 \rceil = 3$. (5 people into 2 holes (friend/enemy).)
Without loss of generality, suppose A has ≥ 3 friends (the enemies case is similar). Call three of them B, C, and D.
If (B, C) or (C, D) or (B, D) are friends then we're done because those two friends with A forms a triple of friends.
Otherwise (B, C) and (C, D) and (B, D) are enemies and BCD forms a triple of enemies.

Pigeonhole Principle: Example #6

While on a 28-day vacation, Martina plays at least one set of tennis each day, but no more that 40 sets over all 28 days. Prove that there is a span of consecutive days in which she plays exactly 15 sets.

Proof: Let x_i be the total number of sets played up to and including day i (for $i = 1, 2, \ldots, 28$). Let $x_0 = 0$.
We need to show that there exist $0 \leq i < j < 28$ such that $x_j = x_i + 15$.
Consider $x_1, x_2, \ldots, x_{28}, x_0 + 15, x_1 + 15, \ldots, x_{27} + 15$. These are 56 integers (pigeons) in the range $[1, 39 + 15]$ (i.e., 54 pigeonholes). Two of these integers are equal by the Pigeonhole Principle. Since $x_i < x_j$ for $i < j$ (because Martina plays ≥ 1 set per day), the two that are equal must be $x_j = 15 + x_i$. So from day $i + 1$ to day j, Martina plays 15 sets.

Pigeonhole Principle: Example #7

Erdös-Szekeres Theorem

Any sequence $x_1, x_2, \ldots, x_n$ of $n \geq (r-1)(s-1)+1$ distinct numbers contains an increasing subsequence of length r or a decreasing subsequence of length s.

4, 7, 12, 3, 62, 14, 2, 8, 11, 5, 20, 17, 1, 22, 15, 13, 18

Proof by Contradiction: Label x_i with the pair (a_i, b_i) where a_i is the length of the longest increasing subsequence ending with x_i and b_i is the length of the longest decreasing subsequence ending with x_i. No two numbers receive the same label since (for $i < j$) if $x_i < x_j$ then $a_i < a_j$ and if $x_i > x_j$ then $b_i < b_j$. If for all i, $a_i < r$ and $b_i < s$, then there are only $(r-1)(s-1)$ labels, so by the Pigeonhole Principle, two numbers receive the same label. Thus, we reach a contradiction.