Unit #6: Hash functions and the Pigeonhole principle

CPSC 221: Algorithms and Data Structures

Lars Kotthoff[1]
larsko@cs.ubc.ca

[1]With material from Will Evans, Steve Wolfman, Alan Hu, Ed Knorr, and Kim Voll.

Unit Outline

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

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.
- `Insert`, `delete`, and `find` using various open addressing and chaining schemes.
- Define various forms of the pigeonhole principle; recognize and solve the specific types of counting and hashing problems to which they apply.

Reminder: 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)

Stores values associated with user-specified keys

- values may be any type
- keys must be comparable

Implementations so far

Worst-case runtimes

	insert	delete	find
Unsorted list	$O(1)$	$\Theta(n)$	$\Theta(n)$
Balanced Trees	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$

Implementations so far

Worst-case runtimes

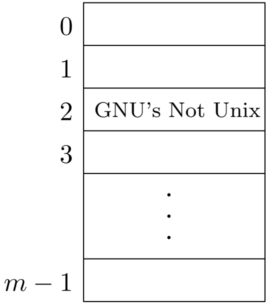
	insert	delete	find
Unsorted list	$O(1)$	$\Theta(n)$	$\Theta(n)$
Balanced Trees	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
Special case: keys in $\{0, 1, \ldots, m-1\}$	$O(1)$	$O(1)$	$O(1)$

Can we get $O(1)$ insert/find/delete for any key type?

Hash Table Goal

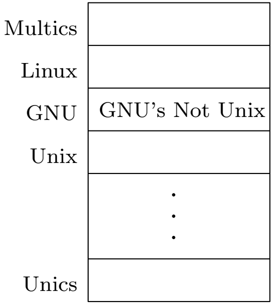
We can do:

a[2]=“GNU’s Not Unix”



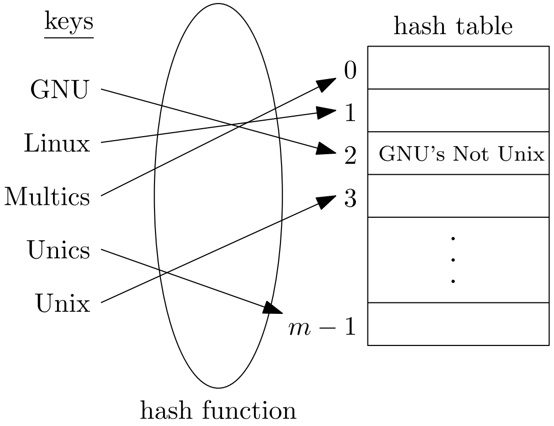
We want to do:

a[“GNU”]=“GNU’s Not Unix”



Hash table approach

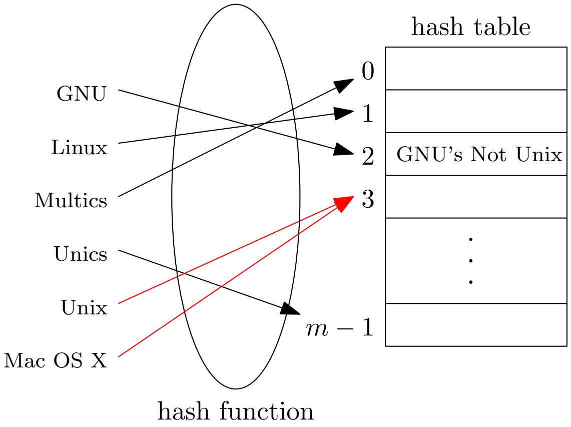
Use a **hash function** to map keys to indices.



hash("GNU") $= 2$

Collisions

A **collision** occurs when two different keys x and y map to the same index, hash$(x) =$ hash(y).



Can we prevent collisions?

Hash table: `find` (first try)

```
Value &find(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?

Good hash function properties

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

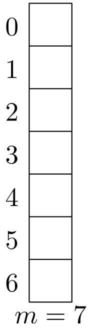
- fast to compute, and
- causes few collisions (we hope).

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

Why a prime number?

Example: hash$(x) = x \bmod 7$

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



Hashing strings

One option

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

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

Hash function treats string an a base 256 number.

Hashing strings

One option

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

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

Hash function treats string an a base 256 number.

Problems

- hash("really, really big") = well...something really, really big
- hash("anything") $\bmod 256$ = hash("anything else") $\bmod 256$

Hashing strings with 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 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

- Fast to compute
- Cause few collisions

Sample hash functions

- For numeric keys x, hash$(x) = x \bmod m$
- hash$(s) =$ string as base 256 number mod m
- Multiplicative hash: hash$(k) = \lfloor m \cdot \mathsf{frac}(ka) \rfloor$ where frac(x) is the fractional part of x and $a = 0.6180339887$ (for example).
- Universal hash: hash$(k) = (a \cdot k + b) \bmod m$ where a and b were chosen at random from $[1, m-1]$ and m prime.
- Cryptographically secure hash (such as SHA-1)

Universal hash functions

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

Example: Suppose $m = 2^b$ and keys are r bits long. Choose a random 0/1 matrix A of size $b \times r$. hash$(x) = A \cdot x$.

$$A \cdot x = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = \text{hash}(x)$$

General form of hash functions

1. Map key to a sequence of bytes.
   - Two equal sequences iff two equal keys.
   - Easy. The key probably is a sequence of bytes already.
2. Map sequence of bytes to an integer x.
   - Changing bytes should cause apparently **random** changes to x.
   - Hard. May be expensive. Cryptographic hash.
3. Map x to a table index using $x \bmod m$.

Collisions

Pigeonhole principle

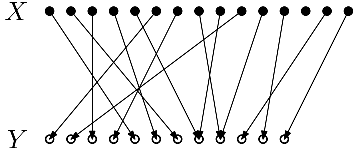
If more than m pigeons fly into m pigeonholes then some pigeonhole contains at least two pigeons.

Corollary

If we hash $n > m$ keys into m slots, two keys will collide (but may already with fewer keys!).

The Pigeonhole Principle

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



The Pigeonhole Principle: Example #0

Image from Wikipedia.

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

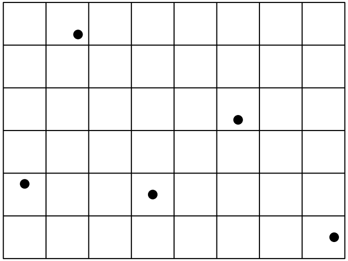
The Pigeonhole Principle: Example #2

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

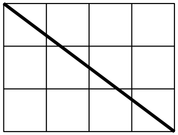
a. 2
b. 4
c. 6
d. 8
e. None of these

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 this diagonal?



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. E.g., $129 = 2^0 \cdot 129$; $60 = 2^2 \cdot 15$.

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 holes, at least one hole contains at least $k = \lceil n/m \rceil$ pigeons.

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

Pigeonhole Principle: Example #5

Show that in a 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).)
Suppose A has ≥ 3 friends (the enemies case is similar) and 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.

Collision Resolution

Birthday Paradox

With probability $>$ _____, two people, in a room of 23, have the same birthday.

General birthday paradox

Even if we *randomly* hash only $\sqrt{2m}$ 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 entry?

- ▷ separate chaining: store multiple items in each entry
- ▷ open addressing: pick a next entry to try

Hashing with Chaining

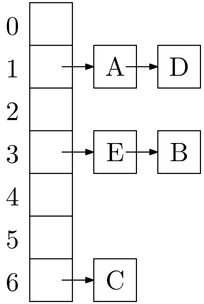
Store multiple items in each entry.

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 on each insertion.



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

Hash: Chaining Code

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

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

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

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

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

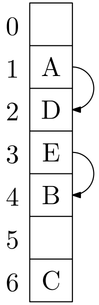
Allow only one item in each slot. The hash function specifies a *sequence* of slots to try.

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 (found) or an empty slot (not found).

Result

- Cannot hash more than m items into a table of size m.
- Hash table memory allocated once.
- Performance depends on number of tries.



Probe Sequence

The sequence of slots 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:

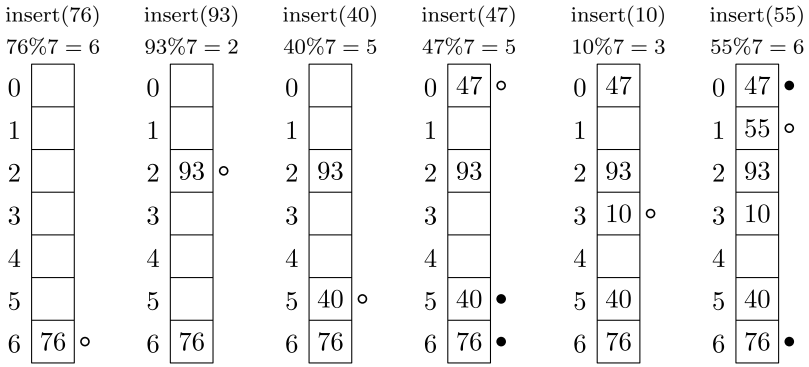
- We first examine slot $h(k, 0)$.
- If it's full, we examine slot $h(k, 1)$.
- If it's full, we examine slot $h(k, 2)$.
- And so on...

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

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

```
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;
}
```

Linear probing example



Access time for linear probing

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

Search cost

On average,

- ▷ an unsuccessful search probes $\approx \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$ slots.
- ▷ a successful search probes $\approx \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$ slots.

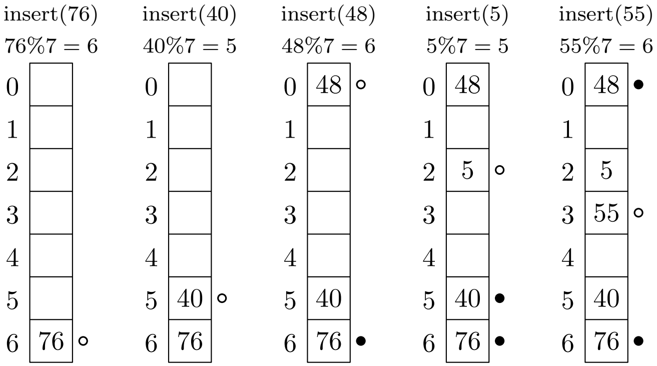
Linear probing suffers from **primary clustering**: creation of long consecutive sequences of filled slots. (They tend to get longer and merge.)

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

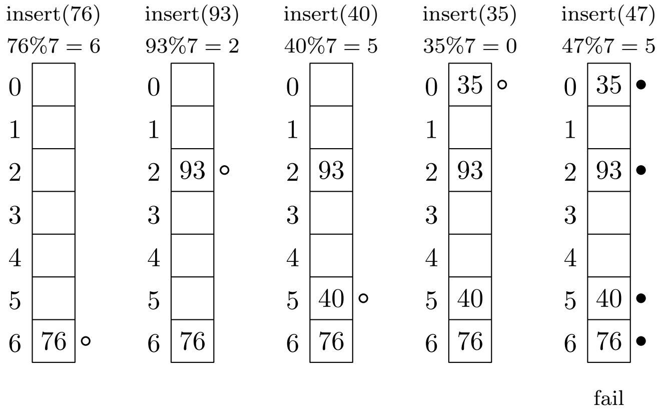
Quadratic probing: $h(k,i) = (\mathsf{hash}(k) + i^2) \bmod m$

```
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;
}
```

Quadratic probing example



Quadratic probing example



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 $0 \le i < j \le \lfloor m/2 \rfloor$,

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

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

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 $i = m - j$.

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

For example: $m = 7$

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

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.

How could we avoid that?

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

```
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;
}
```

Choosing hash$_2(k)$

hash$_2(k)$ should:

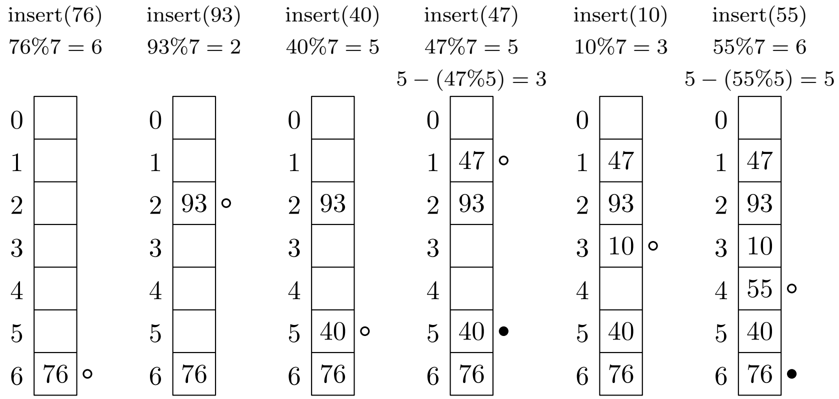
- be quick to evaluate
- differ from hash(k)
- never be 0 (mod m)

We'll use:

$$\text{hash}_2(k) = r - (k \bmod r)$$

for a prime number $r < m$.

Double hashing example



Access time for double hashing

For $\alpha < 1$, double hashing will find an empty slot (assuming m and hash$_2$ are well-chosen).

Search cost

Appears to approach uniform hashing:

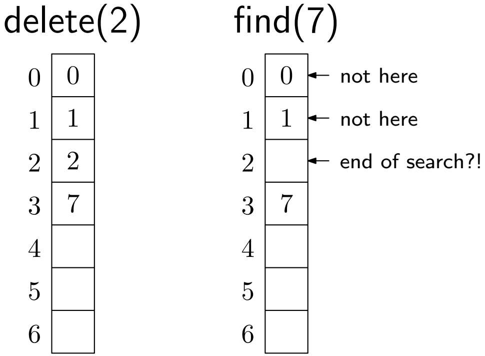
- an unsuccessful search probes $\frac{1}{1-\alpha}$ slots.
- a successful search probes $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$ slots.

No primary or secondary clustering.

One extra hash calculation.

Deletion in Open Addressing

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

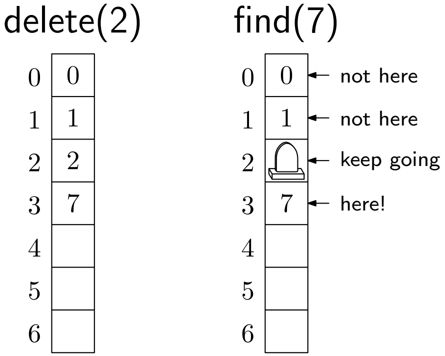


Put a tombstone in the slot.

Find Treat tombstone as an occupied slot.
Insert Treat tombstone as an empty slot.
However, you may need to Find before Insert if you want to avoid duplicate keys (which you do).

Deletion in Open Addressing

Example: hash$(k) = k \bmod 7$.



Put a tombstone in the slot.

Find Treat tombstone as an occupied slot.
Insert Treat tombstone as an empty slot.
However, you may need to Find before Insert if you want to avoid duplicate keys (which you do).

The Squished Pigeon Principle

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

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!

How can we relieve the pressure on the pigeons?

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 $\Theta(n)$ time, but amortized $O(1)$ as long as we double table size on the resize
- 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