Hashing

Mahdi Ebrahimi

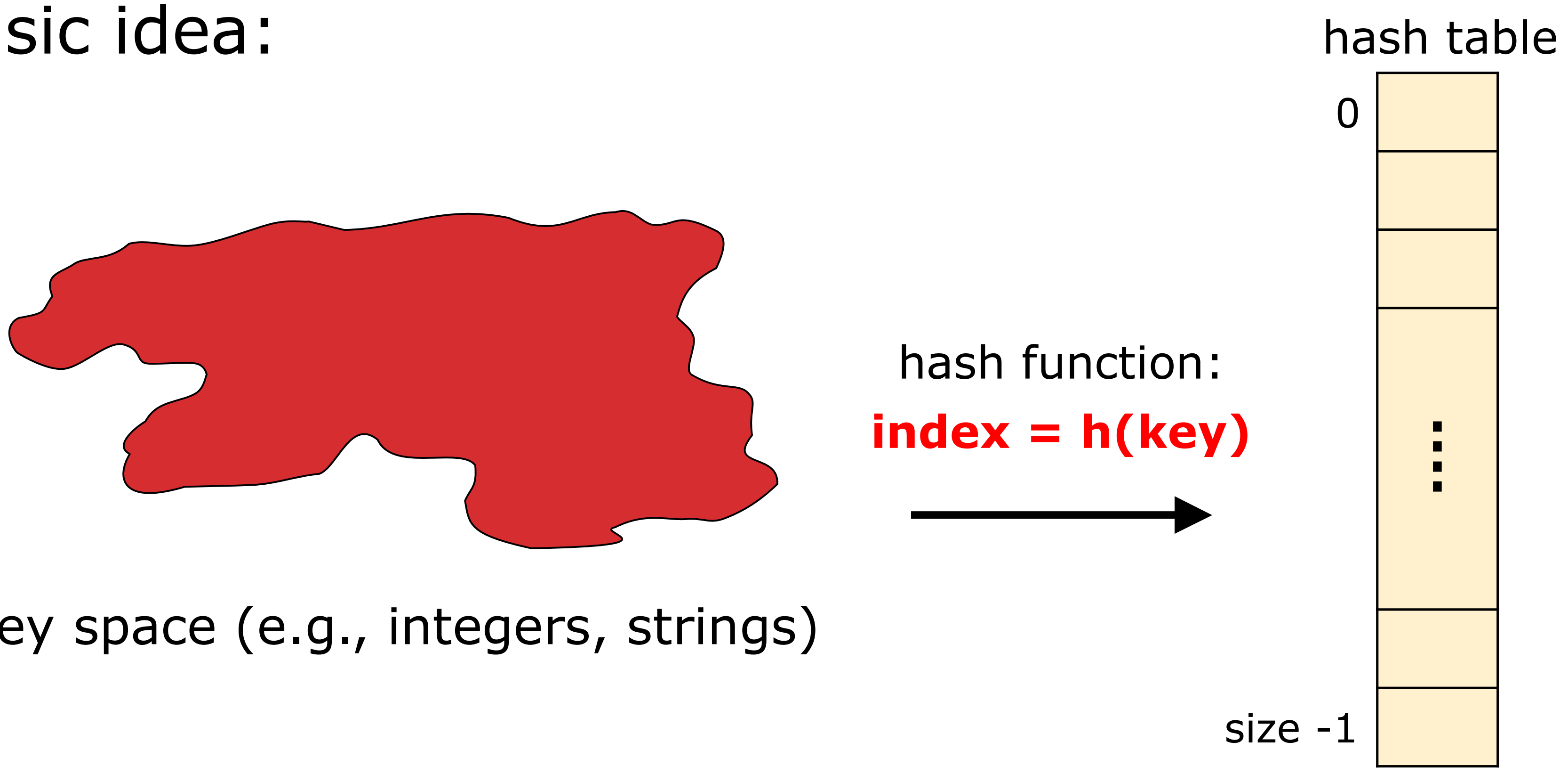
Summer 2020

HASH TABLES

Hash Tables

A hash table is an array of some fixed size

Basic idea:

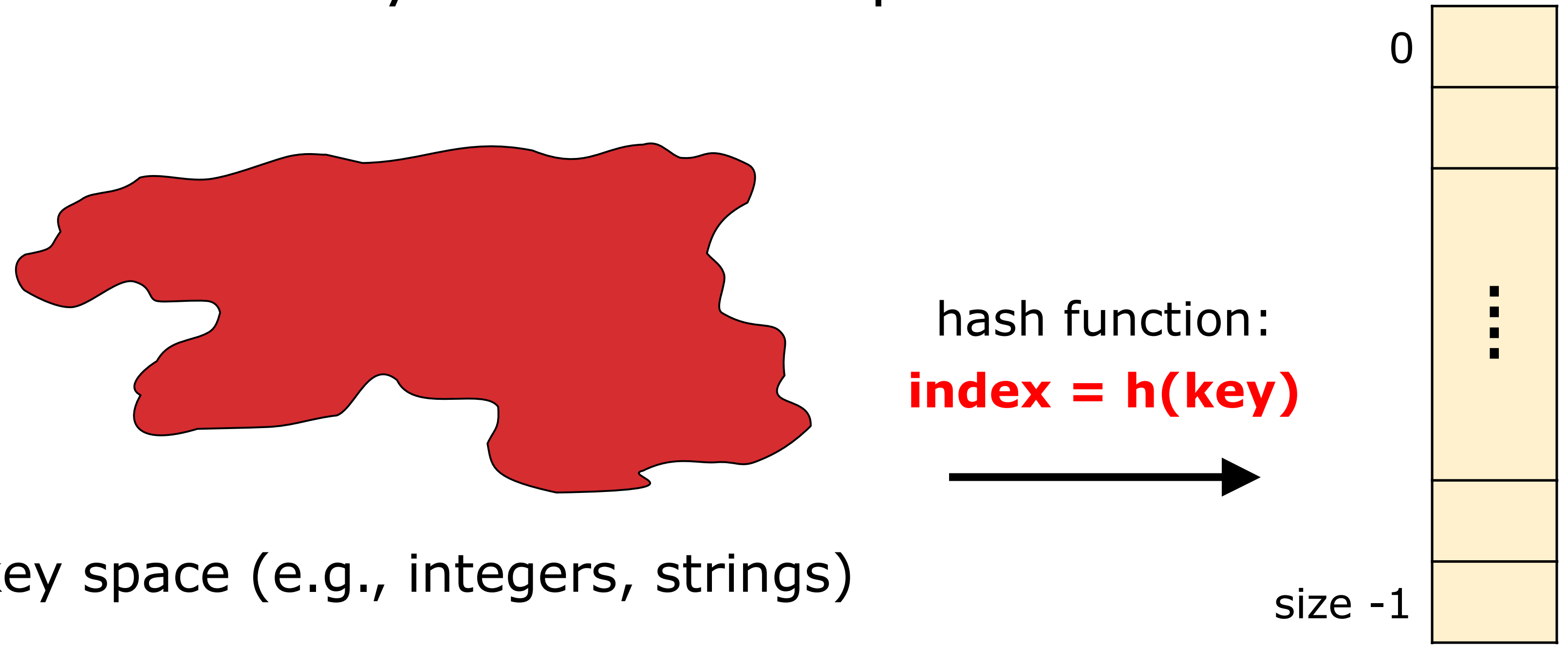


The goal:

Aim for constant-time find, insert, and delete "on average" under reasonable assumptions

An Ideal Hash Functions

- Is fast to compute
- Rarely hashes two keys to the same index
  - Known as *collisions*
  - Zero collisions often impossible in theory but reasonably achievable in practice



What to Hash?

We will focus on two most common things to hash: ints and strings

If you have objects with several fields, it is usually best to hash most of the "identifying fields" to avoid collisions:

```
class Person {
    String firstName, middleName, lastName;
    Date birthDate;
    …
}
```

use these four values

An inherent trade-off:

hashing-time vs. collision-avoidance

Hashing Integers

key space = integers

Simple hash function:

h(key) = key % TableSize

- Client: f(x) = x
- Library: g(x) = f(x) % TableSize
- Fairly fast and natural

Example:

- TableSize = 10
- Insert keys 7, 18, 41, 34, 10

0	10
1	41
2	
3	
4	34
5	
6	
7	7
8	18
9	

Hashing non-integer keys

If keys are not ints, the client must provide a means to convert the key to an int

Programming Trade-off:

- Calculation speed
- Avoiding distinct keys hashing to same ints

Hashing Strings

Key space K = $s_0s_1s_2 \ldots s_{k-1}$
where s_i are chars: $s_i \in [0, 256]$

Some choices: Which ones best avoid collisions?

$$\mathrm{h(K)} = (\mathrm{s}_0)\ \%\ \mathrm{TableSize}$$

$$\mathrm{h(K)} = \left(\sum_{\mathrm{i=0}}^{\mathrm{k-1}} \mathrm{s_i}\right)\ \%\ \mathrm{TableSize}$$

$$\mathrm{h(K)} = \left(\sum_{\mathrm{i=0}}^{\mathrm{k-1}} \mathrm{s_i} \cdot 37^{i}\right)\ \%\ \mathrm{TableSize}$$

COLLISION RESOLUTION

Collision Avoidance

With (x%TableSize), number of collisions depends on

- the ints inserted
- TableSize

Larger table-size tends to help, but not always

- Example: 70, 24, 56, 43, 10
  with TableSize = 10 and TableSize = 60

Collision Resolution

Collision:

When two keys map to the same location in the hash table

We try to avoid it, but the number of keys always exceeds the table size

Hash tables generally must support some form of collision resolution

Flavors of Collision Resolution

Separate Chaining

Open Addressing

- Linear Probing
- Quadratic Probing
- Double Hashing

Terminology Warning

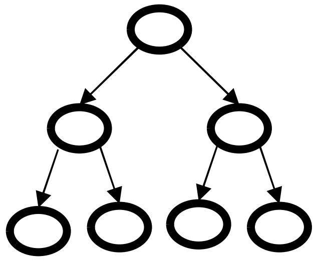
We and the book use the terms

- "chaining" or "separate chaining"
- "open addressing"

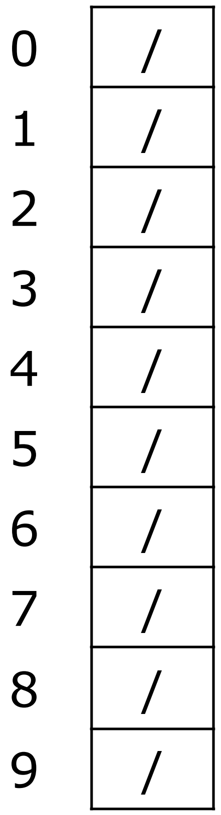
Very confusingly, others use the terms

- "open hashing" for "chaining"
- "closed hashing" for "open addressing"

We also do trees upside-down



Separate Chaining



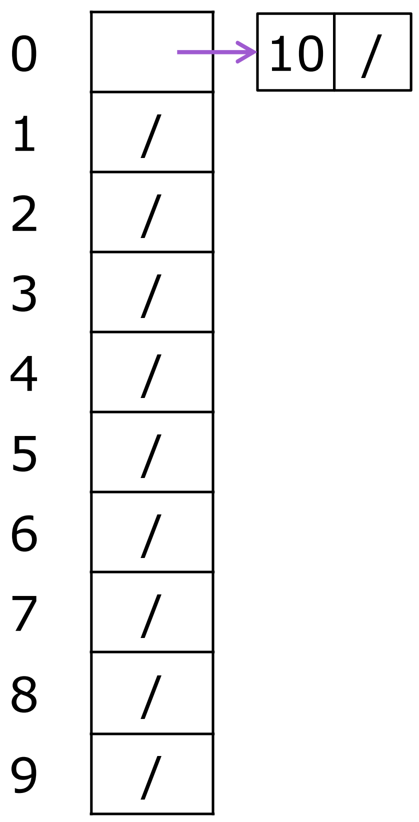
All keys that map to the same table location are kept in a linked list (a.k.a. a "chain" or "bucket")

As easy as it sounds

Example:

    insert 10, 22, 86, 12, 42
    with h(x) = x % 10

Separate Chaining



All keys that map to the same table location are kept in a linked list (a.k.a. a "chain" or "bucket")

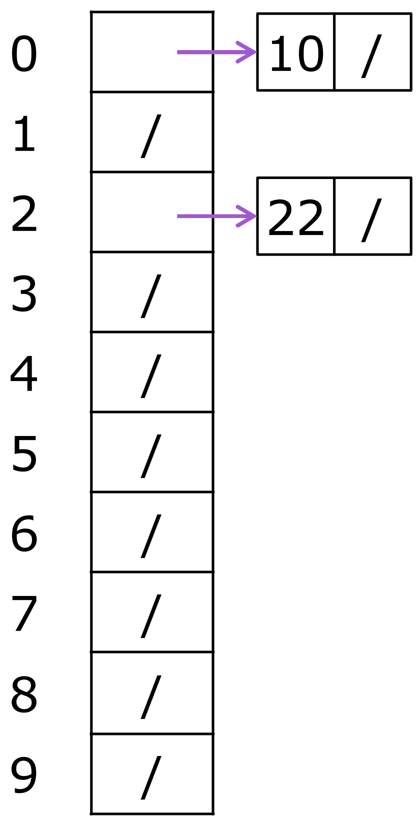
As easy as it sounds

Example:

insert 10, 22, 86, 12, 42

with h(x) = x % 10

Separate Chaining

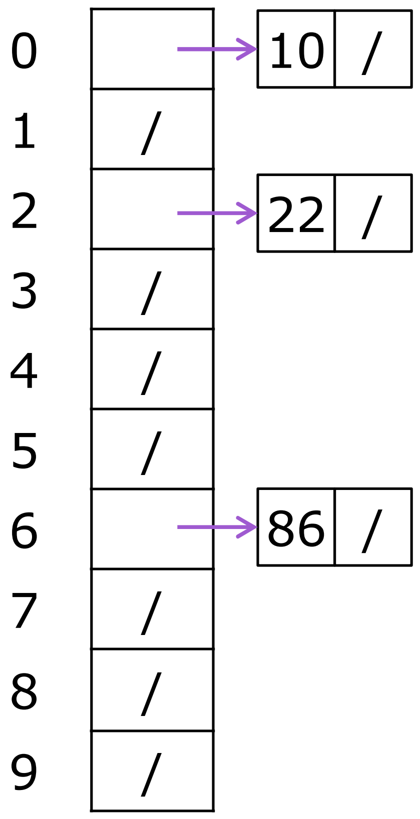


All keys that map to the same table location are kept in a linked list (a.k.a. a "chain" or "bucket")

As easy as it sounds

Example:
insert 10, 22, 86, 12, 42
with h(x) = x % 10

Separate Chaining



All keys that map to the same table location are kept in a linked list (a.k.a. a "chain" or "bucket")

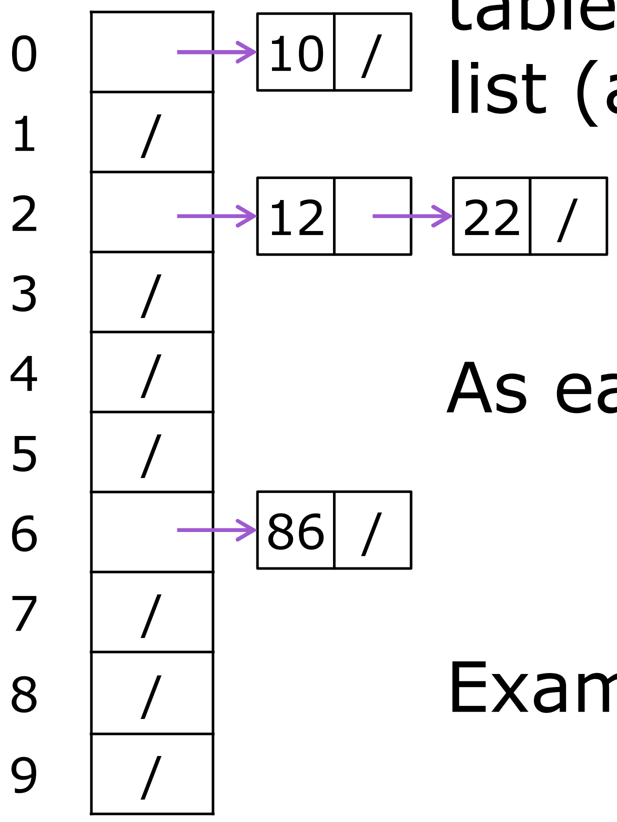
As easy as it sounds

Example:

insert 10, 22, 86, 12, 42

with h(x) = x % 10

Separate Chaining



All keys that map to the same table location are kept in a linked list (a.k.a. a "chain" or "bucket")

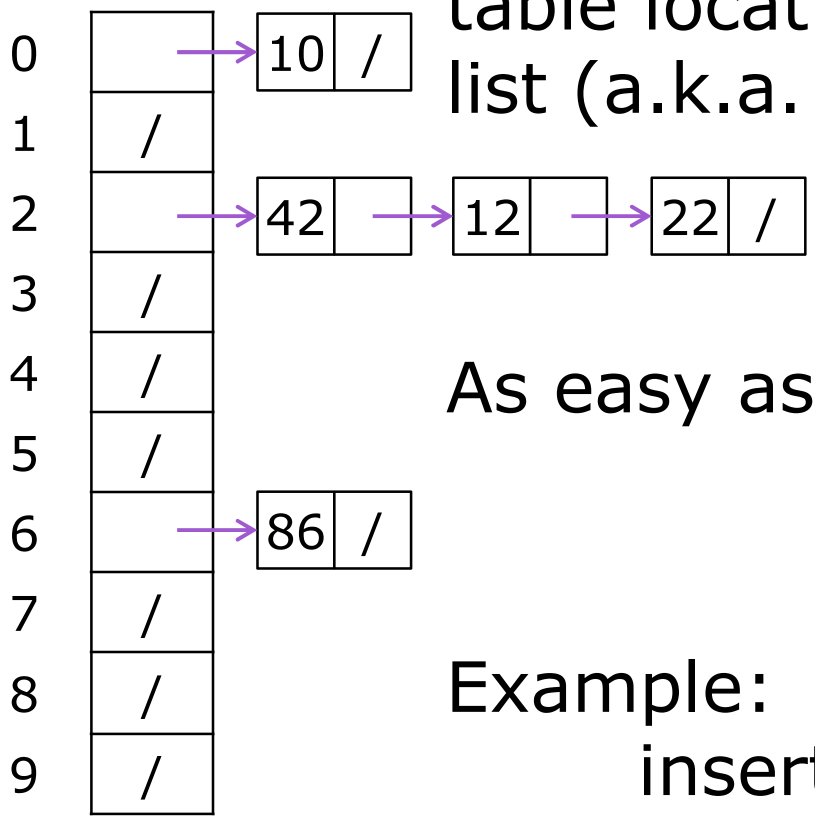
As easy as it sounds

Example:

insert 10, 22, 86, 12, 42

with h(x) = x % 10

Separate Chaining



All keys that map to the same table location are kept in a linked list (a.k.a. a "chain" or "bucket")

As easy as it sounds

Example:

insert 10, 22, 86, 12, 42

with h(x) = x % 10

Thoughts on Separate Chaining

Worst-case time for find?

- Linear
- But only with really bad luck or bad hash function

Beyond asymptotic complexity, some "data-structure engineering" can improve constant factors

- Linked list, array, or a hybrid
- Insert at end or beginning of list
- Sorting the lists gains and loses performance
- Splay-like: Always move item to front of list

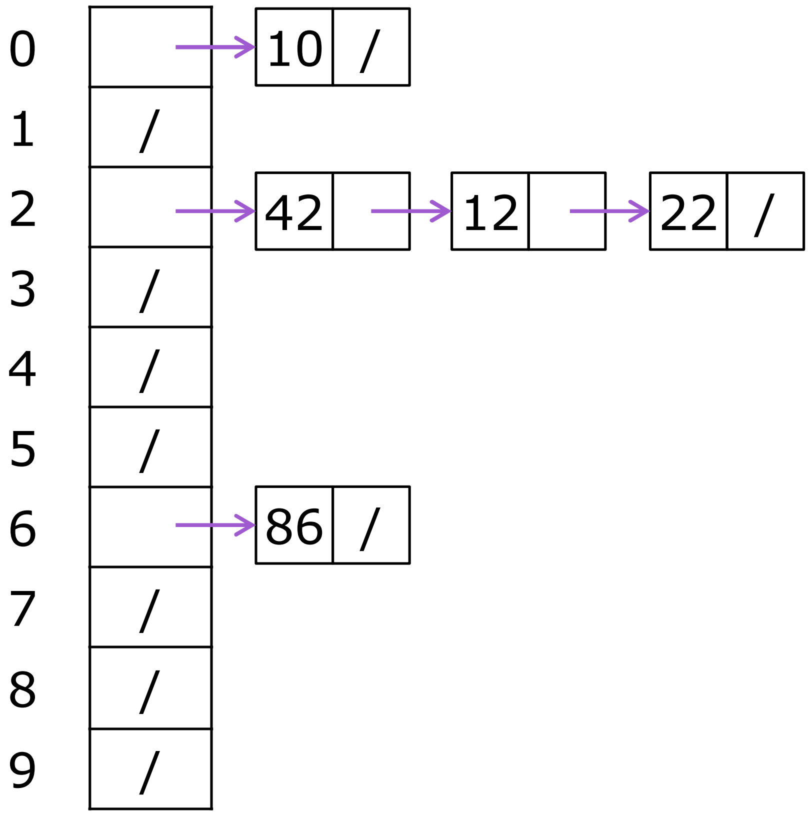
Rigorous Separate Chaining Analysis

The load factor, λ, of a hash table is calculated as

$$\lambda = \frac{n}{TableSize}$$

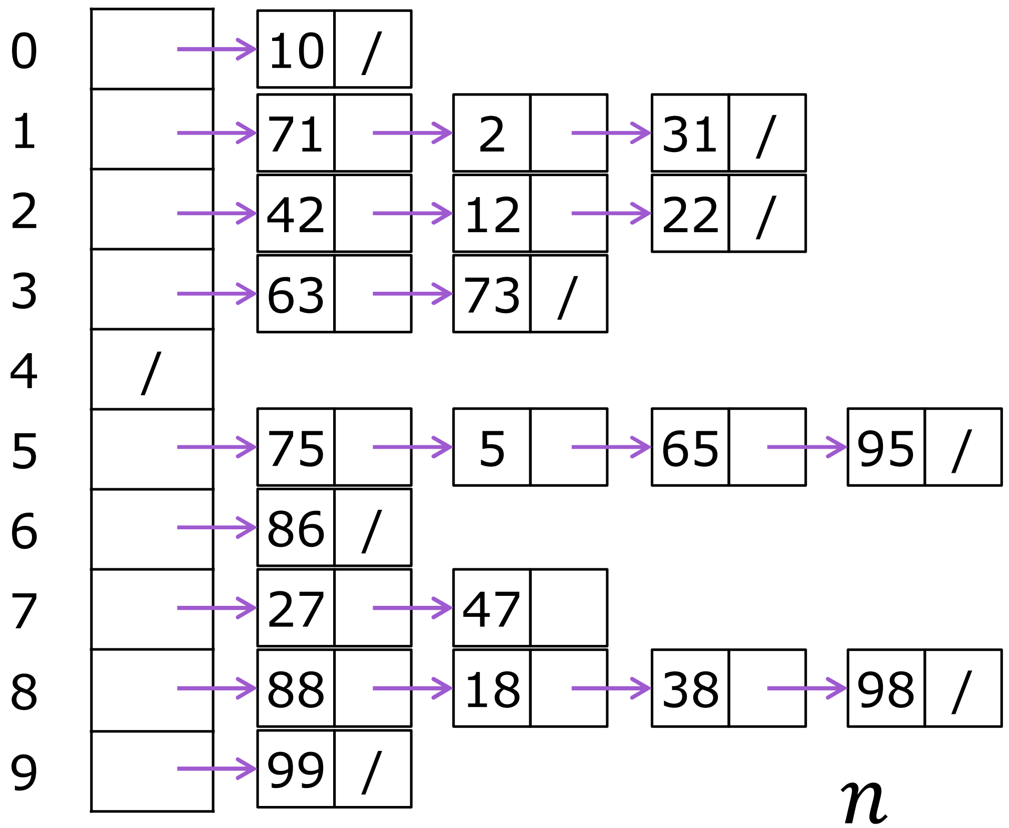
where *n* is the number of items currently in the table

Load Factor?



$$\lambda = \frac{n}{TableSize} = \frac{5}{10} = 0.5$$

Load Factor?



$$\lambda = \frac{n}{TableSize} = \frac{21}{10} = 2.1$$

Rigorous Separate Chaining Analysis

The load factor, λ, of a hash table is calculated as

$$\lambda = \frac{n}{TableSize}$$

where *n* is the number of items currently in the table

Under chaining, the average number of elements per bucket is ___

So if some inserts are followed by random finds, then on average:

- Each unsuccessful find compares against ___ items
- Each successful find compares against ___ items

How big should TableSize be??

Rigorous Separate Chaining Analysis

The load factor, λ, of a hash table is calculated as

$$\lambda = \frac{n}{TableSize}$$

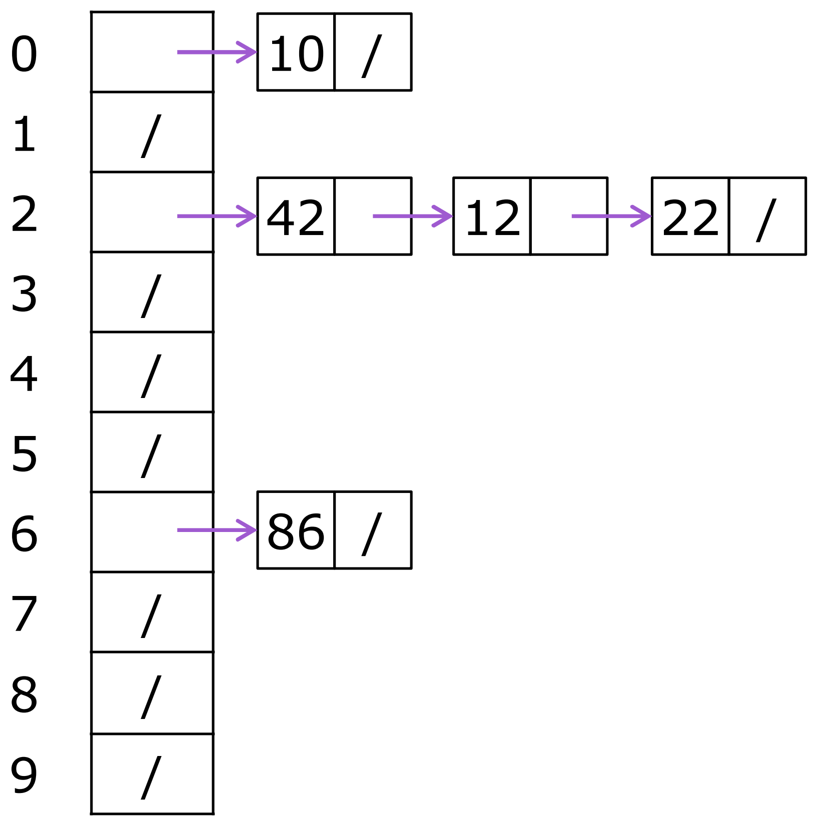
where *n* is the number of items currently in the table

Under chaining, the average number of elements per bucket is λ

So if some inserts are followed by random finds, then on average:

- Each unsuccessful find compares against λ items
- Each successful find compares against λ items
- If λ is low, find and insert likely to be O(1)
- We like to keep λ around 1 for separate chaining

Separate Chaining Deletion

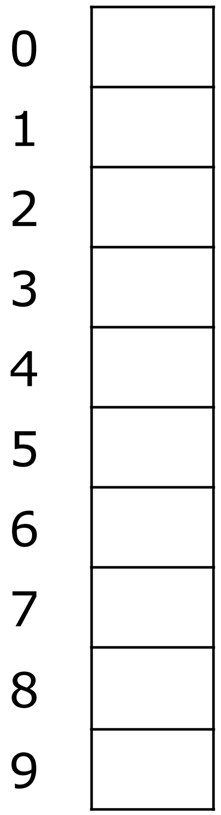


Not too bad and quite easy

- Find in table
- Delete from bucket

Similar run-time as insert

Open Addressing: Linear Probing



Separate chaining does not use all the space in the table. Why not use it?

- Store directly in the array cell
- No linked lists or buckets

How to deal with collisions?

If **`h(key)`** is already full,

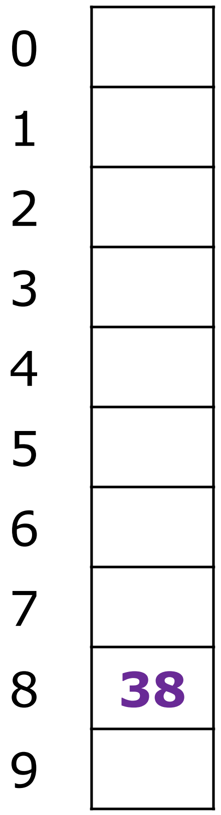
try **`(h(key) + 1) % TableSize`**. If full,

try **`(h(key) + 2) % TableSize`**. If full,

try **`(h(key) + 3) % TableSize`**. If full…

Example: insert 38, 19, 8, 79, 10

Open Addressing: Linear Probing



Separate chaining does not use all the space in the table. Why not use it?

- Store directly in the array cell (no linked list or buckets)

How to deal with collisions?

If **`h(key)`** is already full,

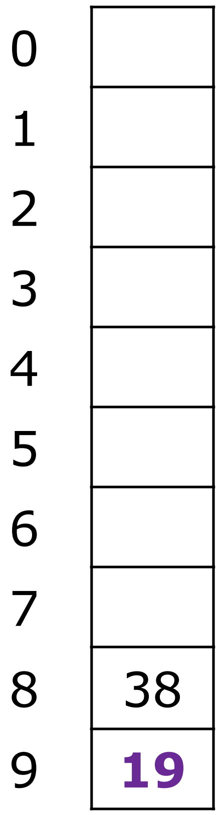
try **`(h(key) + 1) % TableSize`**. If full,

try **`(h(key) + 2) % TableSize`**. If full,

try **`(h(key) + 3) % TableSize`**. If full…

Example: insert 38, 19, 8, 79, 10

Open Addressing: Linear Probing



Separate chaining does not use all the space in the table. Why not use it?

- Store directly in the array cell (no linked list or buckets)

How to deal with collisions?

If **`h(key)`** is already full,

try **`(h(key) + 1) % TableSize`**. If full,

try **`(h(key) + 2) % TableSize`**. If full,

try **`(h(key) + 3) % TableSize`**. If full…

Example: insert 38, 19, 8, 79, 10

Open Addressing: Linear Probing

0	**8**
1	
2	
3	
4	
5	
6	
7	
8	38
9	19

Separate chaining does not use all the space in the table. Why not use it?

- Store directly in the array cell (no linked list or buckets)

How to deal with collisions?

If **`h(key)`** is already full,

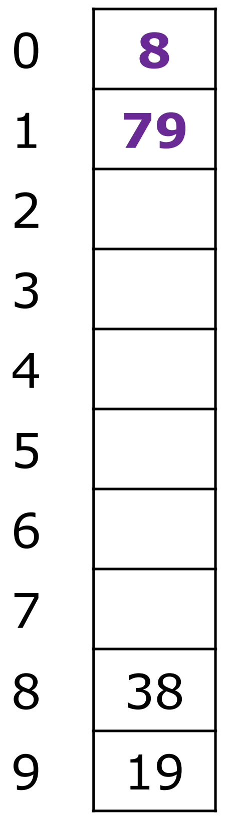
try **`(h(key) + 1) % TableSize`**. If full,

try **`(h(key) + 2) % TableSize`**. If full,

try **`(h(key) + 3) % TableSize`**. If full…

Example: insert 38, 19, 8, 79, 10

Open Addressing: Linear Probing



Separate chaining does not use all the space in the table. Why not use it?

- Store directly in the array cell (no linked list or buckets)

How to deal with collisions?

If **`h(key)`** is already full,

try **`(h(key) + 1) % TableSize`**. If full,

try **`(h(key) + 2) % TableSize`**. If full,

try **`(h(key) + 3) % TableSize`**. If full…

Example: insert 38, 19, 8, 79, 10

Open Addressing: Linear Probing

0	8
1	79
2	**10**
3	
4	
5	
6	
7	
8	38
9	19

Separate chaining does not use all the space in the table. Why not use it?

- Store directly in the array cell (no linked list or buckets)

How to deal with collisions?

If **`h(key)`** is already full,

try **`(h(key) + 1) % TableSize`**. If full,

try **`(h(key) + 2) % TableSize`**. If full,

try **`(h(key) + 3) % TableSize`**. If full…

Example: insert 38, 19, 8, 79, 10

Load Factor?

0	8
1	79
2	**10**
3	
4	
5	
6	
7	
8	38
9	19

Can the load factor when using linear probing ever exceed 1.0?

Nope!!

$$\lambda = \frac{n}{TableSize} = \frac{5}{10} = 0.5$$

Open Addressing in General

This is one example of open addressing

Open addressing means resolving collisions by trying a sequence of other positions in the table

Trying the next spot is called probing

- We just did linear probing
  h(key) + i) % TableSize
- In general have some probe function f and use
  h(key) + f(i) % TableSize

Open addressing does poorly with high load factor λ

- So we want larger tables
- Too many probes means we lose our O(1)

Open Addressing: Other Operations

insert finds an open table position using a probe function

What about find?

- Must use same probe function to "retrace the trail" for the data
- Unsuccessful search when reach empty position

What about delete?

- Must use "lazy" deletion. Why?

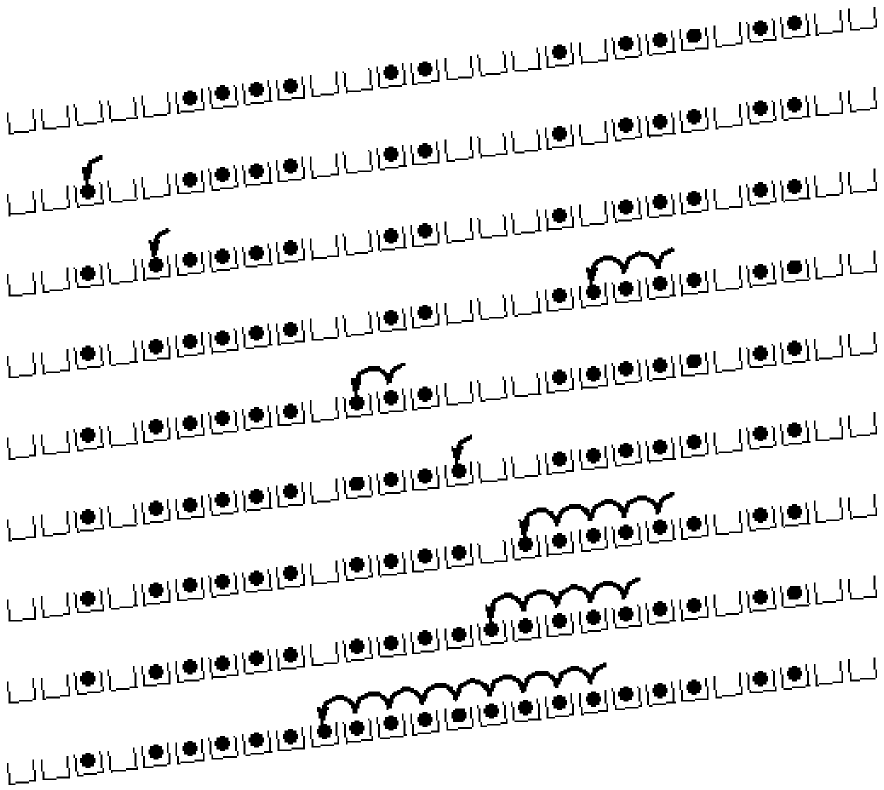
10	✗	/	**23**	/	/	**16**	✗	**26**

- Marker indicates "data was here, keep on probing"

Primary Clustering

It turns out linear probing is a bad idea, even though the probe function is quick to compute (which is a good thing)

- This tends to produce clusters, which lead to long probe sequences
- This is called *primary clustering*
- We saw the start of a cluster in our linear probing example



[R. Sedgewick]

Open Addressing: Quadratic Probing

We can avoid primary clustering by changing the probe function from just i to f(i)

    (h(key) + f(i)) % TableSize

For quadratic probing, f(i) = i^2:

0th probe:	(h(key) + 0) % TableSize
1st probe:	(h(key) + 1) % TableSize
2nd probe:	(h(key) + 4) % TableSize
3rd probe:	(h(key) + 9) % TableSize
…	
ith probe:	(h(key) + i^2) % TableSize

Intuition: Probes quickly "leave the neighborhood"

Quadratic Probing Example

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

TableSize = 10

insert(89)

Quadratic Probing Example

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	**89**

TableSize = 10

insert(89)

insert(18)

Quadratic Probing Example

0	
1	
2	
3	
4	
5	
6	
7	
8	**18**
9	89

TableSize = 10

insert(89)

insert(18)

insert(49)

Quadratic Probing Example

0	**49**
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

TableSize = 10

insert(89)

insert(18)

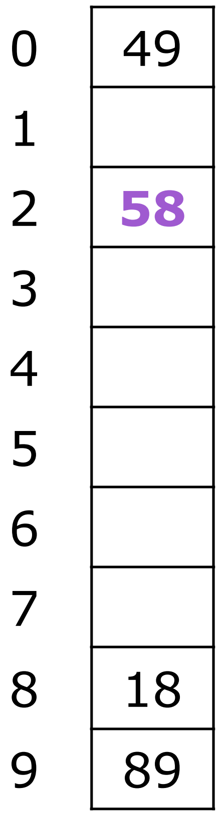
insert(49)

    49 % 10 = 9 collision!

    (49 + 1) % 10 = 0

insert(58)

Quadratic Probing Example



0	49
1	
2	**58**
3	
4	
5	
6	
7	
8	18
9	89

TableSize = 10

insert(89)

insert(18)

insert(49)

insert(58)

  58 % 10 = 8 collision!

  (58 + 1) % 10 = 9 collision!

  (58 + 4) % 10 = 2

insert(79)

Quadratic Probing Example

0	49
1	
2	58
3	79
4	
5	
6	
7	
8	18
9	89

TableSize = 10

insert(89)

insert(18)

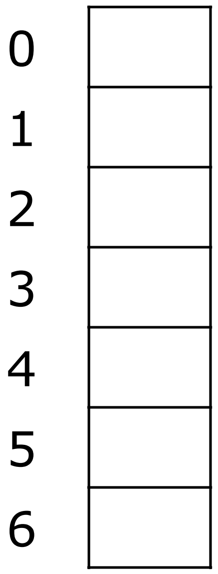
insert(49)

insert(58)

insert(79)

- 79 % 10 = 9 collision!
- (79 + 1) % 10 = 0 collision!
- (79 + 4) % 10 = 3

Another Quadratic Probing Example



TableSize = 7

Insert:

76	(76 % 7 = 6)
40	(40 % 7 = 5)
48	(48 % 7 = 6)
5	(5 % 7 = 5)
55	(55 % 7 = 6)
47	(47 % 7 = 5)

Another Quadratic Probing Example

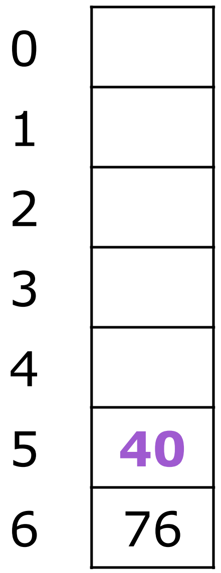
0	
1	
2	
3	
4	
5	
6	**76**

TableSize = 7

Insert:

76	(76 % 7 = 6)
40	(40 % 7 = 5)
48	(48 % 7 = 6)
5	(5 % 7 = 5)
55	(55 % 7 = 6)
47	(47 % 7 = 5)

Another Quadratic Probing Example

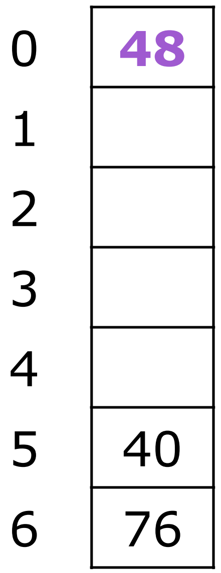


TableSize = 7

Insert:

76	(76 % 7 = 6)
40	(40 % 7 = 5)
48	(48 % 7 = 6)
5	(5 % 7 = 5)
55	(55 % 7 = 6)
47	(47 % 7 = 5)

Another Quadratic Probing Example



TableSize = 7

Insert:

76	(76 % 7 = 6)
40	(40 % 7 = 5)
48	(48 % 7 = 6)
5	(5 % 7 = 5)
55	(55 % 7 = 6)
47	(47 % 7 = 5)

Another Quadratic Probing Example

0	48
1	
2	5
3	
4	
5	40
6	76

TableSize = 7

Insert:

76	(76 % 7 = 6)
40	(40 % 7 = 5)
48	(48 % 7 = 6)
5	(5 % 7 = 5)
55	(55 % 7 = 6)
47	(47 % 7 = 5)

Another Quadratic Probing Example

0	48
1	
2	5
3	**55**
4	
5	40
6	76

TableSize = 7

Insert:

76	(76 % 7 = 6)
40	(40 % 7 = 5)
48	(48 % 7 = 6)
5	(5 % 7 = 5)
55	(55 % 7 = 6)
47	(47 % 7 = 5)

Another Quadratic Probing Example

0	48
1	
2	5
3	55
4	
5	40
6	76

TableSize = 7

Insert:

76	(76 % 7 = 6)
40	(40 % 7 = 5)
48	(48 % 7 = 6)
5	(5 % 7 = 5)
55	(55 % 7 = 6)
47	(47 % 7 = 5)

(47 + 1) % 7 = 6 collision!

(47 + 4) % 7 = 2 collision!

(47 + 9) % 7 = 0 collision!

(47 + 16) % 7 = 0 collision!

(47 + 25) % 7 = 2 collision!

Will we ever get a 1 or 4?!?

Another Quadratic Probing Example

0	48
1	
2	5
3	55
4	
5	40
6	76

insert(47) will always fail here. Why?

For all *n*, $(5 + n^2) \% 7$ is 0, 2, 5, or 6

Proof uses induction and

$$(5 + n^2) \% 7 = (5 + (n - 7)^2) \% 7$$

In fact, for all *c* and *k*,

$$(c + n^2) \% k = (c + (n - k)^2) \% k$$

From Bad News to Good News

After TableSize quadratic probes, we cycle through the same indices

The good news:

- For prime T and $0 \leq i, j \leq T/2$ where $i \neq j$, $(h(key) + i^2) \% T \neq (h(key) + j^2) \% T$
- If TableSize is prime and $\lambda < ½$, quadratic probing will find an empty slot in at most TableSize/2 probes
- If you keep $\lambda < ½$, no need to detect cycles as we just saw

Clustering Reconsidered

Quadratic probing does not suffer from primary clustering as the quadratic nature quickly escapes the neighborhood

But it is no help if keys initially hash the same index

- Any 2 keys that hash to the same value will have the same series of moves after that
- Called *secondary clustering*

We can avoid secondary clustering with a probe function that depends on the key: *double hashing*

Open Addressing: Double Hashing

Idea:

Given two good hash functions h and g, it is very unlikely that for some key, h(key) == g(key)

Why not probe using g(key)?

For double hashing, f(i) = i · g(key):

0th probe:	(h(key) + 0 · g(key)) % TableSize
1st probe:	(h(key) + 1 · g(key)) % TableSize
2nd probe:	(h(key) + 2 · g(key)) % TableSize
…	
ith probe:	(h(key) + i · g(key)) % TableSize

Crucial Detail:

We must make sure that g(key) cannot be 0

Double Hashing

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

T = 10 (TableSize)

Hash Functions:

$h(key) = key \bmod T$

$g(key) = 1 + ((key/T) \bmod (T-1))$

Insert these values into the hash table in this order. Resolve any collisions with double hashing:

13

28

33

147

43

Double Hashing

0	
1	
2	
3	**13**
4	
5	
6	
7	
8	
9	

T = 10 (TableSize)

Hash Functions:

  h(key) = key mod T

  g(key) = 1 + ((key/T) mod (T-1))

Insert these values into the hash table in this order. Resolve any collisions with double hashing:

13

28

33

147

43

Double Hashing

0	
1	
2	
3	13
4	
5	
6	
7	
8	**28**
9	

T = 10 (TableSize)

Hash Functions:

h(key) = key mod T

g(key) = 1 + ((key/T) mod (T-1))

Insert these values into the hash table in this order. Resolve any collisions with double hashing:

13

28

33

147

43

Double Hashing

0	
1	
2	
3	13
4	
5	
6	
7	**33**
8	28
9	

T = 10 (TableSize)

Hash Functions:

h(key) = key mod T

g(key) = 1 + ((key/T) mod (T-1))

Insert these values into the hash table in this order. Resolve any collisions with double hashing:

13

28

33 → g(33) = 1 + 3 mod 9 = 4

147

43

Double Hashing

0	
1	
2	
3	13
4	
5	
6	
7	33
8	28
9	**147**

T = 10 (TableSize)

Hash Functions:

h(key) = key mod T

g(key) = 1 + ((key/T) mod (T-1))

Insert these values into the hash table in this order. Resolve any collisions with double hashing:

13

28

33

147 $\rightarrow$ g(147) = 1 + 14 mod 9 = 6

43

Double Hashing

0	
1	
2	
3	13
4	
5	
6	
7	33
8	28
9	147

T = 10 (TableSize)

Hash Functions:

h(key) = key mod T

g(key) = 1 + ((key/T) mod (T-1))

Insert these values into the hash table in this order. Resolve any collisions with double hashing:

13

28

33

147 → g(147) = 1 + 14 mod 9 = 6

43 → g(43) = 1 + 4 mod 9 = 5

We have a problem:

3 + 0 = 3 3 + 5 = 8 3 + 10 = 13

3 + 15 = 18 3 + 20 = 23

Double Hashing Analysis

Because each probe is "jumping" by g(key) each time, we should ideally "leave the neighborhood" and "go different places from the same initial collision"

But, as in quadratic probing, we could still have a problem where we are not "safe" due to an infinite loop despite room in table

This cannot happen in at least one case:

For primes p and q such that $2 < q < p$

h(key) = key % p

g(key) = q – (key % q)

Summarizing Collision Resolution

Separate Chaining is easy

- find, delete proportional to load factor on average
- insert can be constant if just push on front of list

Open addressing uses probing, has clustering issues as it gets full but still has reasons for its use:

- Easier data representation
- Less memory allocation
- Run-time overhead for list nodes (but an array implementation could be faster)

REHASHING

Rehashing

As with array-based stacks/queues/lists

- If table gets too full, create a bigger table and copy everything
- Less helpful to shrink a table that is underfull

With chaining, we get to decide what "too full" means

- Keep load factor reasonable (e.g., < 1)?
- Consider average or max size of non-empty chains

For open addressing, half-full is a good rule of thumb

Rehashing

What size should we choose?

- Twice-as-big?
- Except that won't be prime!

We go twice-as-big but guarantee prime

- Implement by hard coding a list of prime numbers
- You probably will not grow more than 20-30 times and can then calculate after that if necessary

Rehashing

Can we copy all data to the same indices in the new table?

- Will not work; we calculated the index based on TableSize

Rehash Algorithm:

Go through old table

Do standard insert for each item into new table

Resize is an O(n) operation,

- Iterate over old table: O(n)
- n inserts / calls to the hash function: $n \cdot O(1) = O(n)$

Is there some way to avoid all those hash function calls?

- Space/time tradeoff: Could store h(key) with each data item
- Growing the table is still O(n); only helps by a constant factor

Final Word on Hashing

The hash table is one of the most important data structures

- Efficient find, insert, and delete
- Operations based on sorted order are not so efficient
- Useful in many, many real-world applications
- Popular topic for job interview questions

Important to use a good hash function

  - Good distribution of key hashs
  - Not overly expensive to calculate (bit shifts good!)

Important to keep hash table at a good size

- Keep TableSize a prime number
- Set a preferable λ depending on type of hashtable