CSE 326: Data Structures
Hash Tables

James Fogarty

Autumn 2007

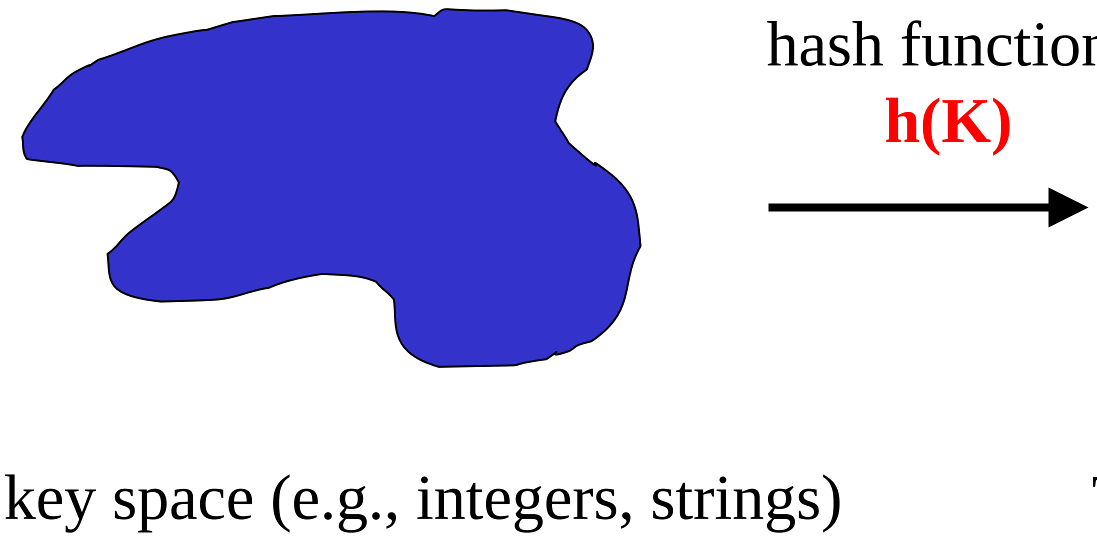
Lecture 14

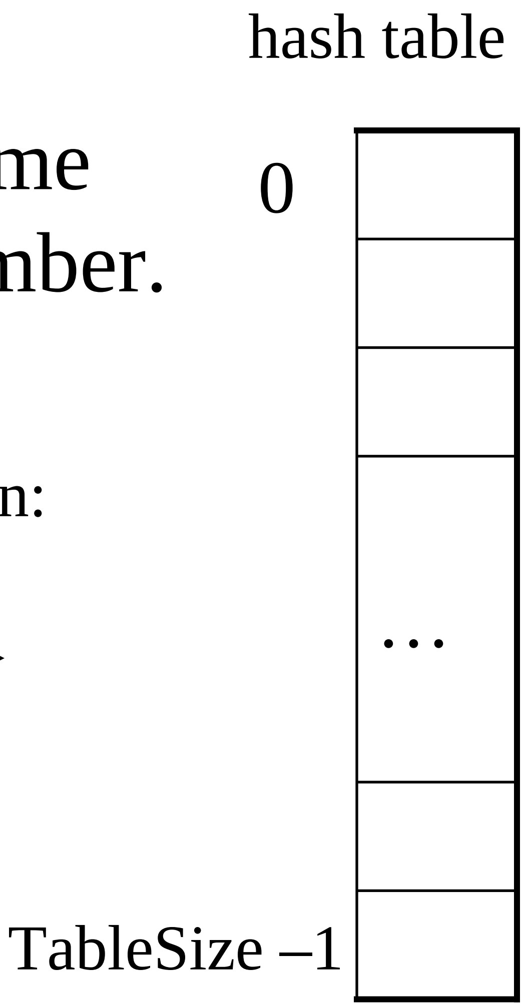
Dictionary Implementations So Far

	Unsorted linked list	Sorted Array	BST	AVL	Splay (amortized)
Insert					
Find					
Delete					

Hash Tables

- Constant time accesses!
- A **hash table** is an array of some fixed size, usually a prime number.
- General idea:



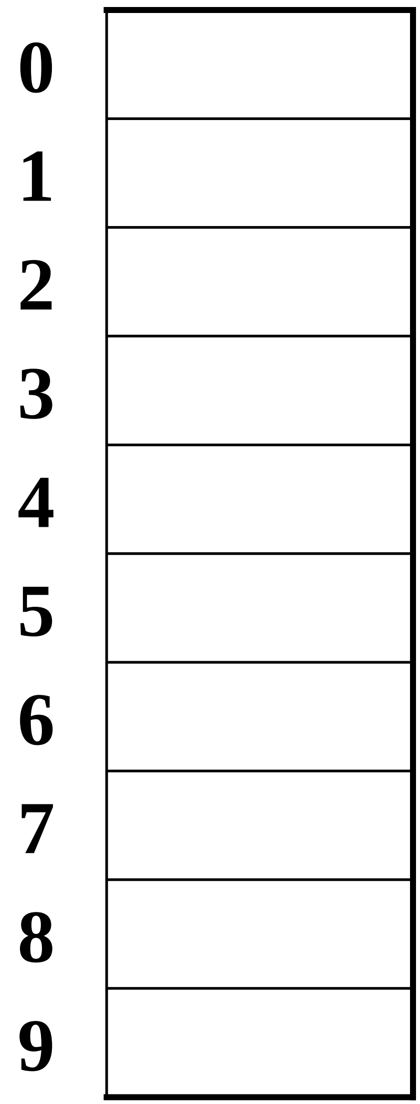


Example

- key space = integers
- TableSize = 10

- **h**(K) = K mod 10

- **Insert**: 7, 18, 41, 94



Another Example

- key space = integers
- TableSize = 6
- **h**(K) = K mod 6
- **Insert**: 7, 18, 41, 34

0	
1	
2	
3	
4	
5	

Hash Functions

1. **simple/fast** to compute,
2. Avoid **collisions**
3. have keys distributed **evenly** among cells.

Perfect Hash function:

Sample Hash Functions:

- key space = strings
- $s = s_0\ s_1\ s_2\ \ldots\ s_{k-1}$

1. $h(s) = s_0 \bmod TableSize$
2. $h(s) = \left(\sum_{i=0}^{k-1} s_i \right) \bmod TableSize$
3. $h(s) = \left(\sum_{i=0}^{k-1} s_i \cdot 37^i \right) \bmod TableSize$

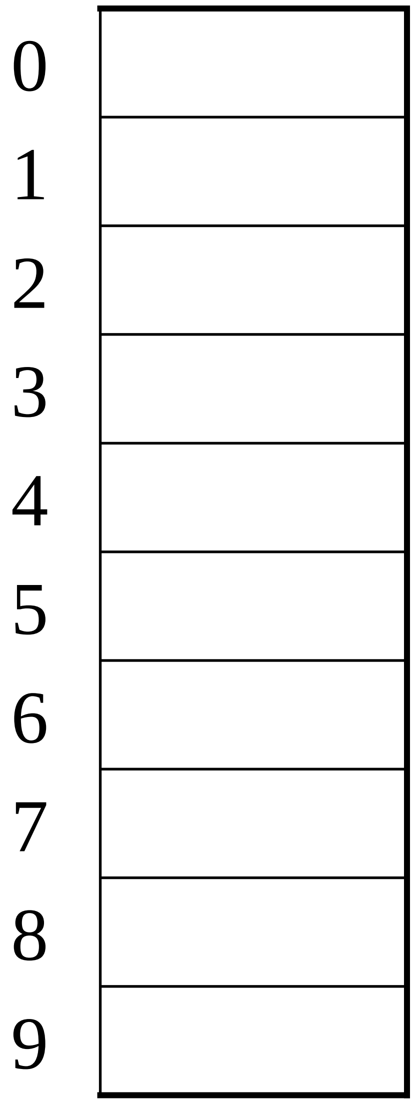
Collision Resolution

Collision: when two keys map to the same location in the hash table.

Two ways to resolve collisions:

1. Separate Chaining
2. Open Addressing (linear probing, quadratic probing, double hashing)

Separate Chaining



Insert:
10
22
107
12
42

- **<u>Separate chaining</u>**: All keys that map to the same hash value are kept in a list (or "bucket").

Analysis of find

- Defn: The load factor, λ, of a hash table is the ratio: $\frac{N}{M}$ ← no. of elements ← table size

For separate chaining, λ = average # of elements in a bucket

- Unsuccessful find:

- Successful find:

How big should the hash table be?

- For Separate Chaining:

tableSize: Why Prime?

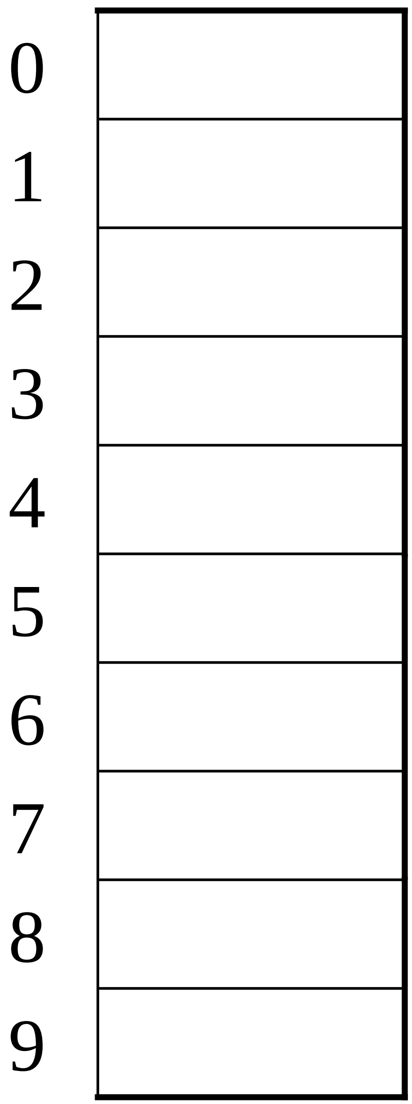
- Suppose
  - data stored in hash table: 7160, 493, 60, 55, 321, 900, 810
  - tableSize = 10

    data hashes to 0, 3, <u>0</u>, 5, 1, <u>0</u>, <u>0</u>
  - tableSize = 11

    data hashes to 10, 9, 5, 0, 2, <u>9</u>, 7

> Real-life data tends to have a pattern
>
> Being a multiple of 11 is usually *not* the pattern ☺

Open Addressing



Insert:
38
19
8
109
10

- **<u>Linear Probing</u>**: after checking spot h(k), try spot h(k)+1, if that is full, try h(k)+2, then h(k)+3, etc.

Terminology Alert!

	“**Open** Hashing”	“Closed Hashing”
	equals	equals
Weiss	“Separate Chaining”	“**Open** Addressing”

Linear Probing

$$f(i) = i$$

- Probe sequence:

  0^{th} probe = h(k) mod TableSize

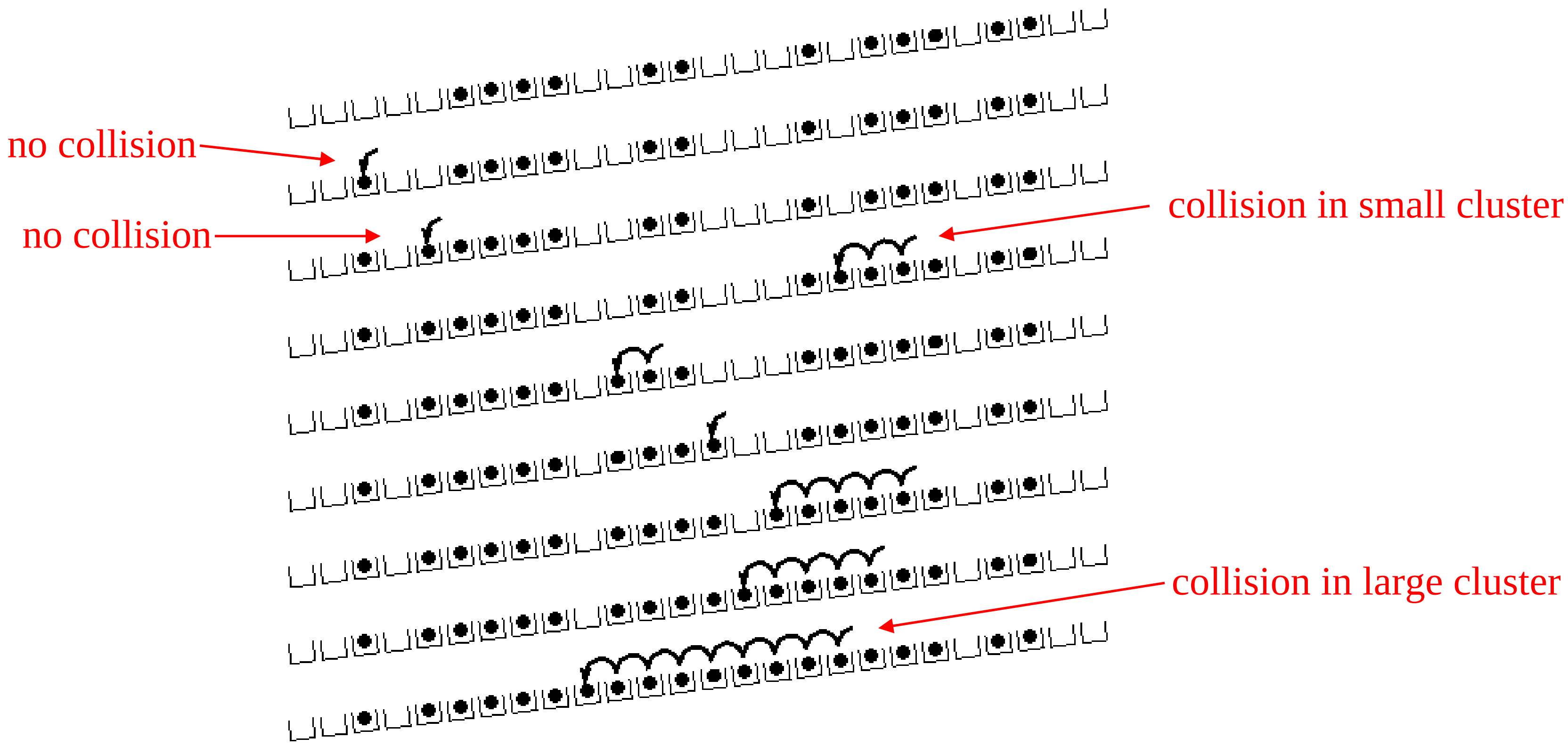
  1^{th} probe = (h(k) + 1) mod TableSize

  2^{th} probe = (h(k) + 2) mod TableSize

  . . .

  i^{th} probe = (h(k) + i) mod TableSize

Linear Probing – Clustering



[R. Sedgewick]

Load Factor in Linear Probing

- For *any* λ < 1, linear probing *will* find an empty slot
- Expected # of probes (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 λ > 1/2

Quadratic Probing

Less likely to encounter Primary Clustering

$$f(i) = i^2$$

- Probe sequence:

  0^{th} probe = h(k) mod TableSize

  1^{th} probe = (h(k) + 1) mod TableSize

  2^{th} probe = (h(k) + 4) mod TableSize

  3^{th} probe = (h(k) + 9) mod TableSize

  . . .

  i^{th} probe = $(h(k) + i^2)$ mod TableSize

Quadratic Probing

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Insert:
89
18
49
58
79

Quadratic Probing Example

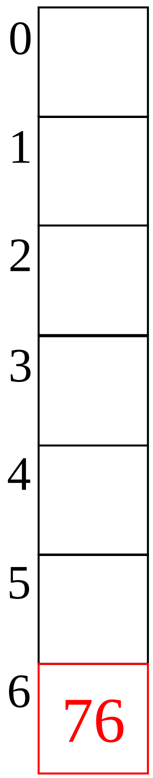
insert(76) 76%7 = 6

insert(40) 40%7 = 5

insert(48) 48%7 = 6

insert(5) 5%7 = 5

insert(55) 55%7 = 6



But… insert(47) 47%7 = 5

Quadratic Probing: Success guarantee for $\lambda < ½$

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

Quadratic Probing: Properties

- For *any* $\lambda < ½$, quadratic probing will find an empty slot; for bigger λ, quadratic probing *may* find a slot

- Quadratic probing does not suffer from *primary* clustering: keys hashing to the same *area* are not bad

- But what about keys that hash to the same *spot*?
  - ***Secondary Clustering!***

Double Hashing

$$f(i) = i * g(k)$$

where g is a second hash function

- Probe sequence:

  0^{th} probe = h(k) mod TableSize

  1^{th} probe = (h(k) + g(k)) mod TableSize

  2^{th} probe = (h(k) + 2*g(k)) mod TableSize

  3^{th} probe = (h(k) + 3*g(k)) mod TableSize

  . . .

  i^{th} probe = (h(<u>k</u>) + i*g(<u>k</u>)) mod TableSize

Double Hashing Example

h(k) = k mod 7 and g(k) = 5 – (k mod 5)

	76	93	40	47	10	55
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

Resolving Collisions with Double Hashing

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Hash Functions:

$H(K) = K \bmod M$

$H_2(K) = 1 + ((K/M) \bmod (M-1))$

$M =$

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

13

28

33

147

43

Rehashing

Idea: When the table gets too full, create a bigger table (usually 2x as large) and hash all the items from the original table into the new table.

- When to rehash?
  - half full ($\lambda = 0.5$)
  - when an insertion fails
  - some other threshold
- Cost of rehashing?

Java hashCode() Method

- Class Object defines a hashCode method
  - Intent: returns a suitable hashcode for the object
  - Result is arbitrary int; must scale to fit a hash table (e.g. obj.hashCode() % nBuckets)
  - Used by collection classes like HashMap
- Classes should override with calculation appropriate for instances of the class
  - Calculation should involve semantically "significant" fields of objects

hashCode() and equals()

- To work right, particularly with collection classes like HashMap, hashCode() and equals() must obey this rule:

  if a.equals(b) then it must be true that

  a.hashCode() == b.hashCode()

  – Why?
- Reverse is not required

Hashing Summary

- Hashing is one of the most important data structures.
- Hashing has many applications where operations are limited to find, insert, and delete.
- Dynamic hash tables have good amortized complexity.