CSE 326: Data Structures
Hash Tables

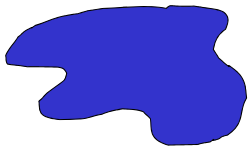
Hal Perkins
Winter 2008
Lecture 16

Dictionary Implementations So Far

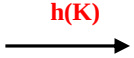
	Unsorted linked list	Sorted Array	BST	AVL	Splay (amortized)
Insert					
Find					
Delete					

Hash Tables

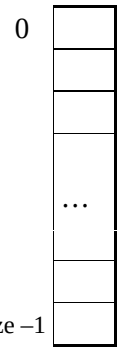
- Constant time accesses!
- A **hash table** is an array of some fixed size, usually a prime number.
- General idea:



hash function: **h(K)**



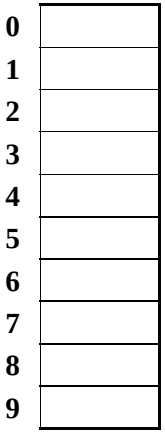
hash table



key space (e.g., integers, strings)

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	

5

Hash Functions

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

Perfect Hash function:

6

Sample Hash Functions:

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

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

7

Collision Resolution

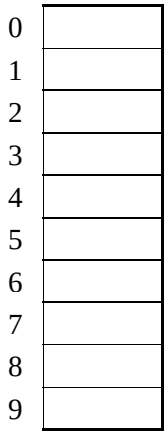
Collision: when two keys map to the same location in the hash table.

Two ways to resolve collisions:

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

8

Separate Chaining



Insert:
10
22
107
12
42

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

9

tableSize: Why Prime?

- Suppose
  - data stored in hash table: 7160, 493, 60, 55, 321, 900, 810
  - tableSize = 10
    data hashes to 0, 3, 0, 5, 1, 0, 0
  - tableSize = 11
    data hashes to 10, 9, 5, 0, 2, 9, 7

Real-life data tends to have a pattern

Being a multiple of 11 is usually *not* the pattern ☺

10

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:

11

How big should the hash table be?

- For Separate Chaining:

12

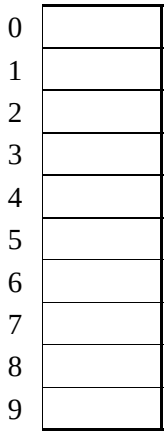
Rehashing

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?

13

Open Addressing



Insert:
38
19
8
109
10

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

14

Terminology Alert!

"**Open** Hashing"	"Closed Hashing"
equals	equals
Weiss "Separate Chaining"	"**Open** Addressing"

15

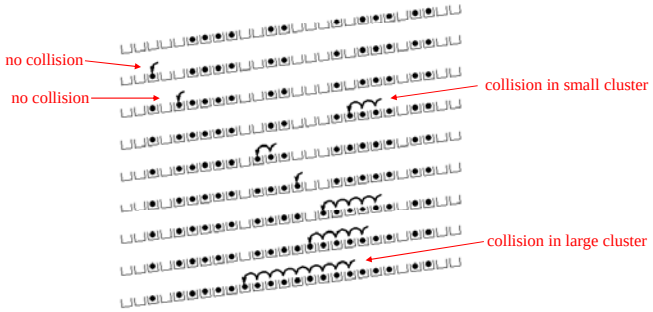
Linear Probing

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

16

Linear Probing – Clustering



[R. Sedgewick]

17

Load Factor in Linear Probing

- For *any* $\lambda < 1$, 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 $\lambda > 1/2$

18

Quadratic Probing

$f(i) = i^2$

Less likely to encounter Primary Clustering

- 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

19

Quadratic Probing

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Insert:
89
18
49
58
79

20

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 λ < ½

- If size is prime and λ < 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`
    `⇒ (i² - j²) mod size = 0`
    `⇒ [(i + j)(i - j)] mod size = 0`

  BUT size does not divide `(i-j)` or `(i+j)`

Quadratic Probing: Properties

- For *any* λ < 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!***

Quadratic Probing Works for λ < 1/2

- If HSize is prime then

(h(x) + i^2) mod HSize ≠ (h(x) + j^2) mod HSize for i ≠ j and 0 ≤ i,j < HSize/2.

- Proof

  (h(x) + i^2) mod HSize = (h(x) + j^2) mod HSize
  (h(x) + i^2) - (h(x) + j^2) mod HSize = 0
  (i^2 - j^2) mod HSize = 0
  (i-j)(i+j) mod HSize = 0
  ⇒⇐ HSize does not divide (i-j) or (i+j)

Double Hashing

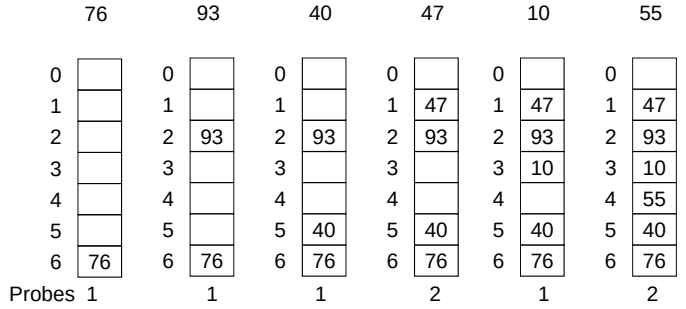
f(i) = i * g(k)
where g is a second hash function

- Probe sequence:
  - 0th probe = h(k) mod TableSize
  - 1th probe = (h(k) + g(k)) mod TableSize
  - 2th probe = (h(k) + 2*g(k)) mod TableSize
  - 3th probe = (h(k) + 3*g(k)) mod TableSize
  - . . .
  - ith probe = (h(k) + i*g(k)) mod TableSize

25

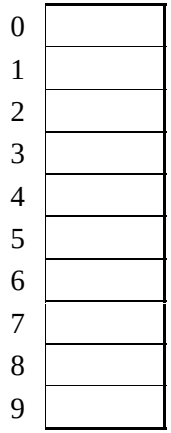
Double Hashing Example

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



26

Resolving Collisions with Double Hashing



Hash Functions:
H(K) = K mod M
$H_2(K)$ = 1 + ((K/M) mod (M-1))
M =

Insert these values into the hash table in this order. Resolve any collisions with double hashing:
13
28
33
147
43

27

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

28

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

29

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.

30