CSE 326 Data Structures

Dave Bacon

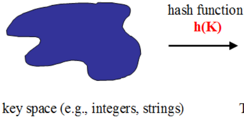
Hashing

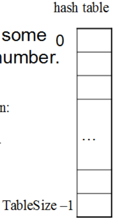
Logistics

- Turn in Homework 4
- Reading: Chapter 5, Start Chapter 8

Hash Tables

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





Sample Hash Functions:

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

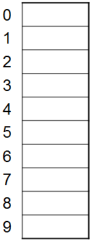
1. $h(s) = s_0 \bmod TableSize$

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

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

tableSize: Why Prime?

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").

Open Addressing

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

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.

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 $\lambda > 1/2$

Primary Clustering

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

0	
1	
2	
3	
4	
5	
6	76

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

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 $0 \le$ `i,j` $\le$ `size/2` and `i` $\neq$ `j`

    $(h(x) + i^2) \bmod size \neq (h(x) + j^2) \bmod size$
  - by contradiction: suppose that for some $i \neq j$:

    $(h(x) + i^2) \bmod size = (h(x) + j^2) \bmod size$

    $\Rightarrow i^2 \bmod size = j^2 \bmod size$

    $\Rightarrow (i^2 - j^2) \bmod size = 0$

    $\Rightarrow [(i + j)(i - j)] \bmod size = 0$

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

Secondary Clustering

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(k) + i*g(k)) mod TableSize

Double Hashing Example

$h(k) = k \bmod 7$ and $g(k) = 5 - (k \bmod 5)$

	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?

Good	Bad	Ugly

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.

Disjoint Sets

Chapter 8

Disjoint Union - Find

- Maintain a set of pairwise disjoint sets.
  - {3,5,7} , {4,2,8}, {9}, {1,6}
- Each set has a unique name, one of its members
  - {3,5,7} , {4,2,8}, {9}, {1,6}

Union

- Union(x,y) – take the union of two sets named x and y
  - {3,5,7} , {4,2,8}, {9}, {1,6}
  - Union(5,1)

    {3,5,7,1,6}, {4,2,8}, {9},

Find

- Find(x) – return the name of the set containing x.
  - {3,5,7,1,6}, {4,2,8}, {9},
  - Find(1) = 5
  - Find(4) = 8