CSE 326 Data Structures

Dave Bacon

Hashing

Logistics

- Project 2b due at 11:59pm.
  - → Turn in everything.
  - → max 20 → use zip (no rar)

  office 4-5 David 002
- Homework 4 due Friday
- Midterm in section Thur.
  given back

Implementations So Far

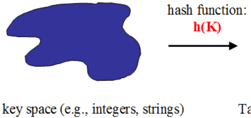
skew

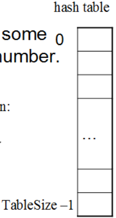
	Unsorted linked list	Sorted Array	BST	AVL	Splay (amortized)
Insert	O(1)	O(N)	O(N)	O(log N)	O(log N)
Find	O(N)	O(log N)	O(N)	O(log N)	O(log N)
Delete	O(N)	O(N)	O(N)	O(log N)	O(log N)

Worst Case

Hash Tables

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





Example

- key space = positive integers
- TableSize = 10

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

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

7 mod 10 = 7
18 mod 10 = 8
41 mod 10 = 1

insert 38?
38 mod 10
= 8

Find (28) → 8

0	
1	41
2	
3	
4	94
5	
6	
7	7
8	18
9	

Another Example

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

0	18
1	7
2	
3	
4	34
5	41

7 mod 6 = 1
18 mod 6 = 0
41 mod 6 = 5
34 mod 6 = 4

what do we want from Hash function

Hash Functions

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

hash function

evenly ≠ uniform

key

Perfect Hash function:

No collisions
No blank locations.

"Ignorant" about data

26 letters 10 digits +1 "space" 37

Sample Hash Functions:

different combinations

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

Table Size $= 10^6$

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

   Spread: 37 0,...,36

   DAVE → 4 collisions
   DUMB

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

   spread 37K

   SPOT, POST, STOP, h → X

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

   Horner's

   $\boxed{s_0}\boxed{s_1}\boxed{s_2}\ldots$

   $s_0 + s_1 \cdot 37 + s_2 37^2 + \ldots$

   spread: 37^K

   anagram problem is gone.

Designing a Hash Function for web URLs

www.hash.com

$s = \underline{s_0\ s_1\ s_2 \ldots s_{k-1}}$

Issues to take into account:

→ huge urls.

→ "www" – "com" – "edu"

→ "." "?"

simple + fast

as much as possible

h(s) =

spread large ~~common~~ keys → same location

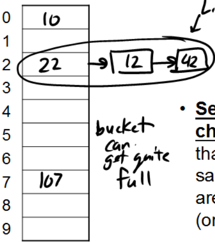
Collision Resolution

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



(BST, AVL)
(etc.)
(hash)

Insert:
10
22
107
12
42

bucket can get quite full

Find 42
h(42)=2

- **Separate chaining**: 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: $\lambda = \frac{N}{M}$ ← no. of elements (N) / ← table size (M)

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

- unsuccessful: λ ← "Avg"

- successful: L.L. not sorted

$1 + \frac{\lambda}{2}$

How big should the hash table be?

- For Separate Chaining:

$\lambda = 1$ ← want

$\lambda = \frac{1}{2}$ wasting space.

tableSize: Why Prime?

=prime number

- 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

collision

Real-life data tends to have a pattern

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

Open Addressing

too many collisions

cluster merge

Insert:
38
19
8
109
10

Find 8

0	8
1	109
2	10
3	
4	
5	
6	
7	
8	38
9	$\boxed{19}$

? delete? Lazy deletion

- **<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" equals "Separate Chaining"

"Closed Hashing" equals "**Open** Addressing"

Weiss

Weiss

$h(k) + f(i)$

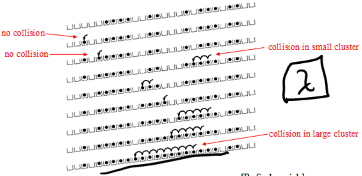
some f

Linear Probing

$f(i) = i$

- Probe sequence:
  - 0^{th} probe = $h(k) \bmod TableSize$
  - 1^{th} probe = $(h(k) + 1) \bmod TableSize$
  - 2^{th} probe = $(h(k) + 2) \bmod TableSize$
  - . . .
  - i^{th} probe = $(h(k) + i) \bmod TableSize$

Linear Probing – Clustering



λ

[R. Sedgewick]

When are we OK?

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 $\lambda > 1/2$

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

0	
1	
2	
3	
4	
5	
6	76

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

    `(h(x) + i`^{2}`) mod size` $\ne$ `(h(x) + j`^{2}`) mod size`
  - by contradiction: suppose that for some i $\ne$ j:

    `(h(x) + i`^{2}`) mod size = (h(x) + j`^{2}`) mod size`

    $\Rightarrow$ `i`^{2} `mod size = j`^{2} `mod size`

    $\Rightarrow$ `(i`^{2} `- j`^{2}`) mod size = 0`

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

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

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?

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.