CSE 326: Data Structures
Hash Tables

Brian Curless

Spring 2008

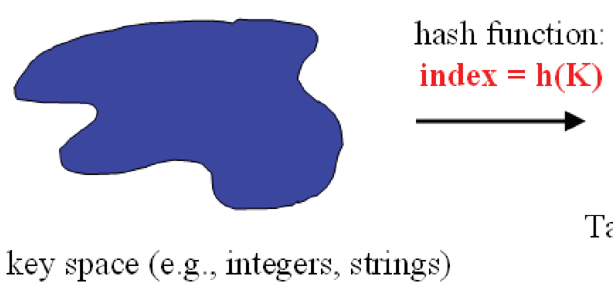
Announcements (5/7/08)

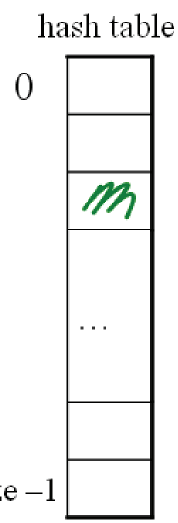
- Project 2B due today.
- Homework #4 due on Friday, beginning of class
- Project #3 assigned on Friday
  - Partner signups by 3pm Friday
- Section: project warm-up, midterms returned, …

- Reading for this lecture: Chapter 5.

Hash Tables

- Find, insert, delete: constant time on average!
- A **hash table** is an array of some fixed size.
- General idea:





Hash Tables

Key space of size M, but we only want to store subset of size N, where N<<M.

- Keys are identifiers in programs. Compiler keeps track of them in a symbol table.
- Keys are student names. We want to look up student records quickly by name.
- Keys are chess configurations in a chess playing program.
- Keys are URLs in a database of web pages.

Simple Integer Hash Functions

- key space = integers
- TableSize = 10
- h(K) = K % 10
- **Insert**: 7, 18, 41, 94

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

Simple Integer Hash Functions

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

0	18
1	7
2	
3	
4	34
5	41

Aside: Properties of Mod

To keep hashed values within the size of the table, we will generally do:

h(K) = function(K) % TableSize

(In the previous examples, function(K) = K.)

It's worth noting a couple properties of the mod function:

- (a + b) % c = [(a % c) + (b % c)] % c
- a % c = b % c → (a – b) % c = 0
- (a b) % c = [(a % c) (b % c)] % c

24 → 4
+57 → 7
81 11
↓
1

24 4
57 7
8 28
↓
8
8

Some String Hash Functions

$s_i \in [0 .. 127]$

key space = strings

K = $s_0\ s_1\ s_2 \ldots s_{m-1}$ (where s_i are characters)

1. $h(K) = s_0$ % TableSize
2. $h(K) = \left(\sum_{i=0}^{m-1} s_i\right)$ % TableSize

   letter aliasing
   anagrams
   spot, post, stop
3. $h(K) = \left(\sum_{i=0}^{m-1} s_i \cdot 128^i\right)$ % TableSize

   $= s_0 + s_1 \cdot 128 + s_2 \cdot 128^2 + \cdots$

Hash Function Desiderata

What are some desirable properties for a hash function?

Evenly distributes values throughout the table.

=> Minimizes collisions

Fast to compute

Designing Hash Functions

We've seen a few possibilities. The simplest is **modular hashing**:

$$h(K) = K \% P$$

where P is usually just the TableSize.

P is often chosen to be prime:

- Reduces likelihood of collisions due to patterns in data
- Is useful for guarantees on certain hashing strategies (as we'll see)

But what would be a more convenient value of P? 2^ℓ

A Fancier Hash Function

Some experimental results indicate that modular hash functions with prime tables sizes are not ideal.

Instead, we can work on designing a really good hash function:

```
jenkinsOneAtATimeHash(String key, int keyLength) {
   hash = 0;
   for (i = 0; i < key_len; i++) {
        hash += key[i];
        hash += (hash << 10);
        hash ^= (hash >> 6);
   }
   hash += (hash << 3);
   hash ^= (hash >> 11);
   hash += (hash << 15);

   return hash % TableSize;
}
```


Collision Resolution

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

How can we cope with collisions?

open addressing { Look at next spot in table, if empty → use; if full, → go to next

separate chaining { List at each table entry

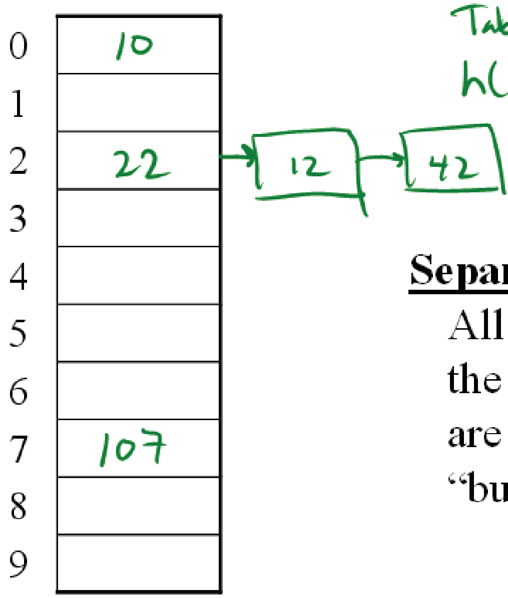
Overflow table?

Separate Chaining

TableSize = 10

h(K) = K % 10

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 Separate Chaining

The load factor, λ, of a hash table is

$$\lambda = \frac{N}{\text{TableSize}} \quad \leftarrow \text{no. of elements}$$

Separate chaining: λ = average # of elems per bucket

Average cost of:

- Unsuccessful find? λ
- Successful find? $1 + \frac{\lambda}{2}$
- Insert? 1

Alternative: Use Empty Space in the Table

h(k) = K % 10

Insert:
38
19
8
109
10

0	8
1	109
2	10
3	
4	
5	
6	
7	
8	38
9	19

Try h(K).
If full, try h(K)+1.
If full, try h(K)+2.
If full, try h(K)+3.
Etc…

Open Addressing

The approach on the previous slide is an example of **open addressing**:

After a collision, try "next" spot. If there's another collision, try another, etc.

Finding the next available spot is called **probing**:

0^{th} probe = h(k) % TableSize
1^{th} probe = (h(k) + f(1)) % TableSize
2^{th} probe = (h(k) + f(2)) % TableSize
. . .
i^{th} probe = (h(k) + f(i)) % TableSize

f(i) is the probing function. We'll look at a few…

Terminology Alert!

- **Separate chaining** is sometimes called **open hashing**.
- **Open addressing** is sometimes called **closed hashing**.

Open Addressing Example, Revisited

0	8
1	109
2	10
3	
4	
5	
6	
7	
8	38
9	19

Insert:
38
19
8
109
10

Try h(K)
If full, try h(K)+1.
If full, try h(K)+2.
If full, try h(K)+3.
Etc…

What is f(i)? $f(i) = i$

Linear Probing

$f(i) = i$

- Probe sequence:

  0^{th} probe = h(K) % TableSize

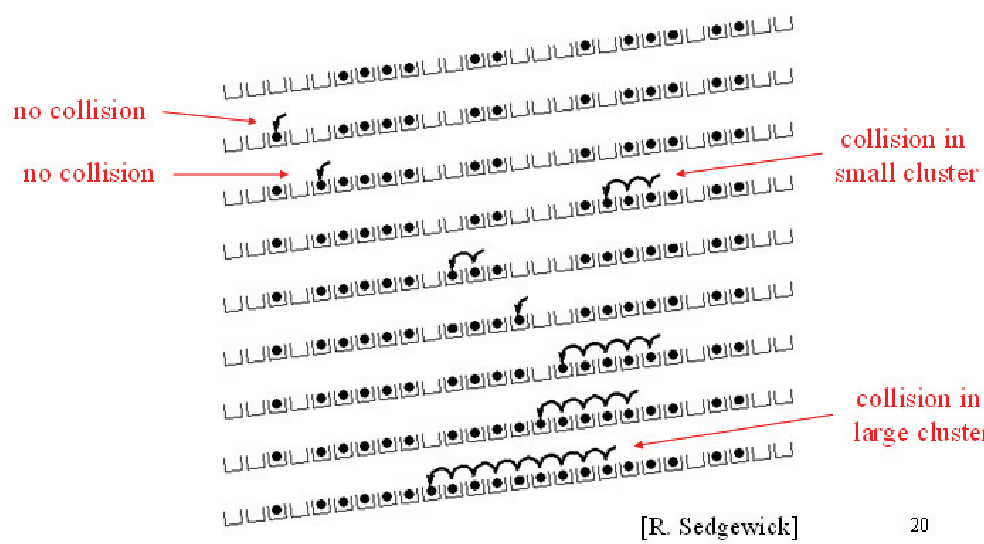
  1^{th} probe = (h(K) + 1) % TableSize

  2^{th} probe = (h(K) + 2) % TableSize

  . . .

  i^{th} probe = (h(K) + i) % TableSize

Linear Probing – Clustering



[R. Sedgewick]

Analysis of Linear Probing

- For *any* $\lambda < 1$, linear probing *will* find an empty slot
- Expected # of probes (for large table sizes)
  - unsuccessful search: $\frac{1}{2}\left(1 + \frac{1}{(1-\lambda)^2}\right)$
  - successful search: $\frac{1}{2}\left(1 + \frac{1}{(1-\lambda)}\right)$

2.5
50.5

$\lambda = 1/2 \Downarrow$ 1.5

$\lambda = 0.9 \Downarrow$ 5.5

- 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) % TableSize
  1^{th} probe = (h(K) + 1) % TableSize
  2^{th} probe = (h(K) + 4) % TableSize
  3^{th} probe = (h(K) + 9) % TableSize
  . . .
  i^{th} probe = (h(K) + i^2) % TableSize

Quadratic Probing Example

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

Insert:
89
18
49
58
79

Another Quadratic Probing Example

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

TableSize = 7
h(K) = K % 7

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
insert(47) 47 % 7 = 5

$(47 + i^2) \% 7$

$(47 \% 7 + i^2 \% 7) \% 7$

$(47 \% 7 + (i \% 7)(i \% 7)) \% 7$

5 + 0, 1, 2, 3, 4, 5, 6

0, 1, 4, 9, 16, 25, 36

5 + 0, 1, 4, 2, 2, 4, 1

5, 6, 9, 7

5, 6, 2, 0

Quadratic Probing: Success guarantee for $\lambda < \frac{1}{2}$

Assertion #1: If T = TableSize is **prime** and $\lambda < \frac{1}{2}$, then quadratic probing will find an empty slot in T/2 probes or fewer.

Assertion #2: If ~~the following holds~~

`(h(K) + i²) % T ≠ (h(K) + j²) % T`

for prime T and all $0 \le i,j \le T/2$ and $i \ne j$,

~~then assertion #1 is true.~~

Assertion #3: if assertion #2 true, then so is assertion #1.

Quadratic Probing: Success guarantee for $\lambda < \frac{1}{2}$

We can prove assertion #2 by contradiction.

Suppose that for some $i \ne j$, $0 \le i,j \le T/2$, prime T:

$$(h(K) + i^2) \% T = (h(K) + j^2) \% T$$

$$(h(K) + i^2 - h(K) - j^2) \% T = 0$$

$$(i^2 - j^2) \% T = 0$$

$$[(i+j)(i-j)] \% T = 0$$

$i+j=0$? N $i+j=T$? N $i-j=0$? N $i-j=T$? N

Quadratic Probing: Properties

- For *any* $\lambda < \frac{1}{2}$, 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

Idea: given two different (good) hash functions h(K) and g(K), it is unlikely for two keys to collide with both of them.

So… let's try probing with a second hash function:

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

- Probe sequence:
  - 0^{th} probe = h(K) % TableSize
  - 1^{th} probe = (h(K) + g(K)) % TableSize
  - 2^{th} probe = (h(K) + 2*g(K)) % TableSize
  - 3^{th} probe = (h(K) + 3*g(K)) % TableSize
  - . . .
  - i^{th} probe = (h(K) + i*g(K)) % TableSize

Double Hashing Example

0	
1	47
2	93
3	10
4	55
5	40
6	76

TableSize = 7
h(K) = K % 7
g(K) = 5 – (K % 5)

Insert(76) 76 % 7 = 6 and 5 - 76 % 5 =
Insert(93) 93 % 7 = 2 and 5 - 93 % 5 =
Insert(40) 40 % 7 = 5 and 5 - 40 % 5 =
Insert(47) 47 % 7 = 5 and 5 - 47 % 5 = 3
Insert(10) 10 % 7 = 3 and 5 - 10 % 5 =
Insert(55) 55 % 7 = 6 and 5 - 55 % 5 = 5

Another Example of Double Hashing

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

Hash Functions:
T = TableSize = 10
h(K) = K % T
g(K) = 1 + (K/T) % (T-1)

Insert these values into the hash table in this order. Resolve any collisions with double hashing:
13
28
33 h=3, g=4
147 h=7, g=6 Σ = 13 → 3 7+2×6 = 19 → 9
43 h=3, g=5

Analysis of Double Hashing

- Double hashing is safe for $\lambda < 1$ for at least one case:
  - h(k) = k % p
  - g(k) = q – (k % q)
  - 2 < q < p, and p, q are primes

Never let g(k)=0

- Expected # of probes (for large table sizes)
  - unsuccessful search: $\dfrac{1}{1-\lambda}$
  - successful search: $\dfrac{1}{\lambda}\log_e\left(\dfrac{1}{1-\lambda}\right)$

Deletion in Separate Chaining

How do we delete an element with separate chaining?

find
remove from list

Deletion in Open Addressing

Can we do something similar for open addressing?

- Delete → mark "occupied"
- Find
- Insert

$h(k) = k \% 7$
Linear probing

Delete(23)
Find(59)
Insert(30)

0	
1	
2	16
3	~~23~~ → occupied → 30
4	59
5	
6	76

Rehashing

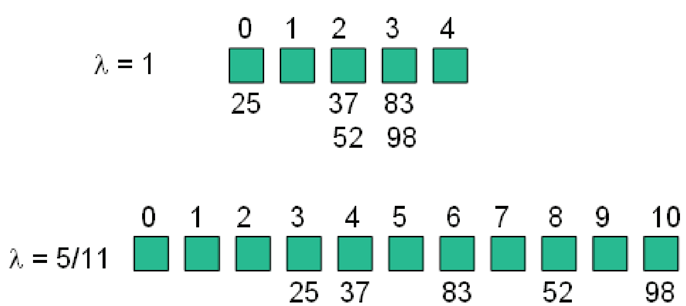
Idea: When the table gets too full, create a bigger table (usually 2x as large) and hash all the items from the original table into the new table.

- When to rehash?
  - Separate chaining: full ($\lambda = 1$)
  - Open addressing: half full ($\lambda = 0.5$)
  - When an insertion fails
  - Some other threshold
- Cost of a single rehashing?

Rehashing Example

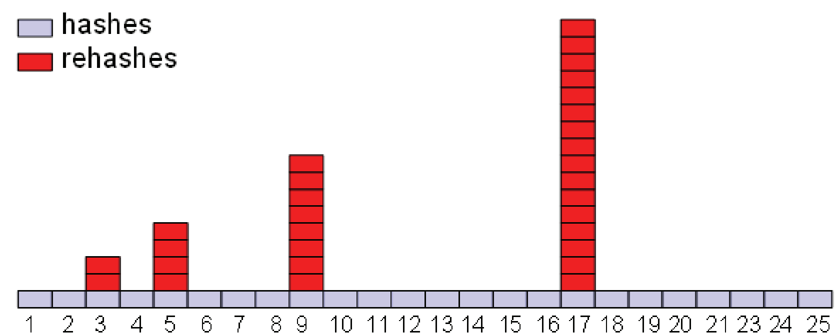
- Separate chaining example:

  $h_1(x) = x \% 5$ rehashes to $h_2(x) = x \% 11$.



Rehashing Picture

- Starting with table of size 2, double when load factor > 1.



Amortized Analysis of Rehashing

- Cost of inserting n keys is $< 3n$
- $2^k + 1 \leq n \leq 2^{k+1}$
  - Hashes = n
  - Rehashes = $2 + 2^2 + \ldots + 2^k = 2^{k+1} - 2$
  - Total = $n + 2^{k+1} - 2 < 3n$
- Example
  - $n = 33$, Total = $33 + 64 - 2 = 95 < 99$

Hashing Summary

- Hashing is one of the most important data structures.
- Hashing has many applications where operations are limited to find, insert, and delete.
  - But what is the cost of doing, e.g., findMin?
- Can use:
  - Separate chaining (easiest)
  - Open hashing (memory conservation, no linked list management)
  - Java uses separate chaining
- Rehashing has good amortized complexity.
- Also has a big data version to minimize disk accesses: extendible hashing. (See textbook.)