CSE 326: Data Structures
It's an open-and-closed hash!

Luke McDowell
Summer Quarter 2003

Reminder: Dictionary ADT

- Dictionary ADT:
  - Maps *values* to user-specified *keys*
  - Or: a set of (key, value) pairs
    - Keys may be any (homogeneous) type
    - Values may be any (homogeneous) type
- Operations:
  - Insert (key, value)
  - Find (key)
  - Remove (key)

insert
• steve0@u
- aka stevaroo…

find(erbad)
• erbad
- Aiman Erbad, the …

- lucasm
  - Lucas Kelsey McDowell, your favorite 5th year grad student who needs help with Hollywood trivia
- stevaroo
  - Steven Loyd Martin, a recently graduated CSE ugrad who's destined fpr great things at Berkeley
- erbad
  - Aiman Mahmood Erbad, the man with the mysterious accent and real name that is even longer

The Dictionary ADT is sometimes called the "Map ADT"

Implementations So Far

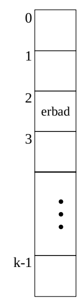
	insert	find	delete
• Unsorted list	O(1)	O(n)	O(n)
• Sorted list	O(n)	O(log n)?	O(n)
• Trees	O(log n)	O(log n)	O(log n)

How about O(1) insert/find/delete?

Hash Table Goal

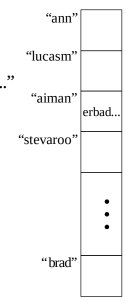
We can do:

a[2] = "erbad"

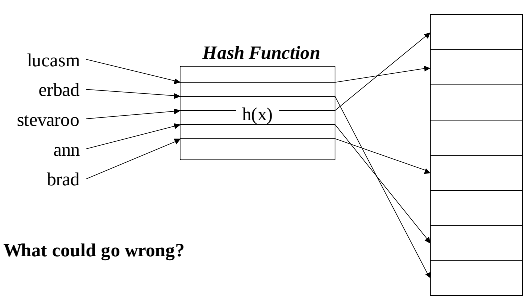


We want to do:

a["erbad"] = "Aiman..."



Hash Table Approach



What could go wrong?

Hash Table Code
First Pass

```
Value find(Key k) {
  int index = hash(k) % tableSize;
  return Table[index];
}
```

Key Questions:

1. What should the hash function be?
2. How should we resolve collisions?
3. What should the table size be?

A Good Hash Function…

…is easy (fast) to compute (O(1) *and* practically fast).

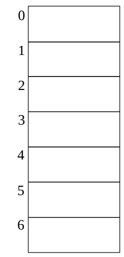
…distributes the data evenly (hash(a) % size ? hash(b) % size).

…uses the whole hash table (for all 0 ? k < size, there's an i such that hash(i) % size = k).

Good Hash Function for Integers

- Choose
  - tableSize is prime
  - hash(n) = n
- Example:
  - tableSize = 7

insert(4)
insert(17)
find(12)
insert(9)
delete(17)



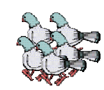
Easy/boring stuff we're going to skip

- Why does the table size have to be prime?
- Picking a good hash function for strings

Read Section 5.2 of the text!

Collisions

- Pigeonhole principle says we can't avoid all collisions
  - try to hash without collision *m* keys into *n* slots with *m* > *n*
  - e.g., try to put 7 pigeons into 5 holes



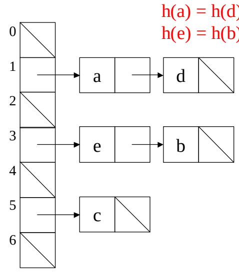
- What do we do when two keys hash to the same entry?
  - Separate chaining: put little dictionaries in each entry
    shove extra pigeons in one hole!
  - Open addressing: pick a next entry to try
- Frequency depends on load factor

load factor ? = $\frac{\text{\# of entries in table}}{\text{tableSize}}$

Separate Chaining

- Put a mini-Dictionary at each entry
  - Usually a linked list
  - Why not a search tree?
- Properties
  - ? can be greater than 1
  - performance degrades with length of chains

h(a) = h(d)
h(e) = h(b)



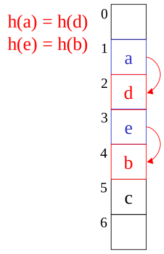
Load Factor in Separate Chaining

- Search cost ?
  - unsuccessful search:
  - successful search:
- Desired load factor:

Open Addressing

What if we only allow one Key at each entry?
- two objects that hash to the same spot can't both go there
- first one there gets the spot
- next one must **probe** for another spot

- Properties
  - ? ? 1
  - performance degrades with difficulty of finding right spot



Salary-Boosting Obfuscation

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

Probing

- Probing how to:
  - First probe - given a key k, hash to h(k)
  - Second probe - if h(k) is occupied, try h(k) + f(1)
  - Third probe - if h(k) + f(1) is occupied, try h(k) + f(2)
  - And so forth
- Probing properties
  - we force f(0) = 0
  - the i^{th} probe is to (h(k) + f(i)) mod size
- **When does the probe fail?**
- **Does that mean the table is full?**

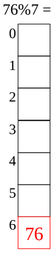
Linear Probing

f(i) = i

- Probe sequence is
  - h(k) mod size
  - h(k) + 1 mod size
  - h(k) + 2 mod size
  - …

Linear Probing Example

insert(76)	insert(93)	insert(40)	insert(47)	insert(10)	insert(55)
76%7 = 6	93%7 = 2	40%7 = 5	47%7 = 5	10%7 = 3	55%7 = 6



Problem?

Load Factor in Linear Probing

- For *any* ? < 1, linear probing will find an empty slot
- Search cost (for large table sizes)
  - successful search: $\frac{1}{2}\left(1+\frac{1}{(1-?)}\right)$
  - unsuccessful search: $\frac{1}{2}\left(1+\frac{1}{(1-?)^2}\right)$
- Linear probing suffers from *primary clustering*
- Performance quickly degrades for ? > 1/2

Quadratic Probing

$f(i) = i^2$

- Probe sequence is
  - h(k) mod size
  - (h(k) + 1) mod size
  - (h(k) + 4) mod size
  - (h(k) + 9) mod size
  - …

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 Succeeds (for ? < ½)

- If size is prime and ? < 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 how can `i + j = 0` or `i + j = size` when `i ? j` and `i,j ? size/2`?
  - same for `i - j mod size = 0`

Load Factor in Quadratic Probing

- For *any* ? < ½, quadratic probing will find an empty slot; for greater ?, quadratic probing *may* find a slot
- Quadratic probing does not suffer from *primary* clustering
- But what about keys that hash to the same spot?

Double Hashing

$f(i) = i \,?\, hash_2(x)$

- Probe sequence is
  - $h_1(k)$ mod size
  - $(h_1(k) + 1 \,?\, h_2(x))$ mod size
  - $(h_1(k) + 2 \,?\, h_2(x))$ mod size
  - …
- Goal?

A Good Double Hash Function…

…is quick to evaluate.

…differs from the original hash function.

…never evaluates to 0 (mod size).

One good choice is to choose a prime R < size and:

$hash_2(x)$ = R - (x mod R)

Double Hashing Example

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

Load Factor in Double Hashing

- For *any* ? < 1, double hashing will find an empty slot (given appropriate table size and $hash_2$)
- Search cost appears to approach optimal (random hash):
  - successful search: $\frac{1}{?}\ln\frac{1}{1??}$
  - unsuccessful search: $\frac{1}{1??}$
- No primary clustering and no secondary clustering
- Cost?

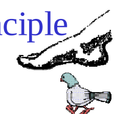
Deletion with Open Addressing

insert(7) delete(2) find(7)

0	0
1	1
2	2
3	
4	
5	
6	

- Solution?

The Squished Pigeon Principle



- An insert using open addressing *cannot* work with a load factor of 1 or more.
- An insert using open addressing with quadratic probing may not work with a load factor of ½ or more.
- Whether you use separate chaining or open addressing, large load factors lead to poor performance!
- How can we relieve the pressure on the pigeons?

Rehashing

- When the load factor gets "too large" (over a constant threshold on ?), rehash all the elements into a new, larger table:
  - spreads keys back out, may drastically improve performance
  - avoids failure for closed hashing techniques
  - allows arbitrarily large tables starting from a small table
  - clears out lazily deleted items
- Cost?
- Can we just copy over into a bigger array?

Rehashing Example

0	20
1	96
2	82
3	
4	89

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Extendible Hashing

Hashing technique for huge data sets

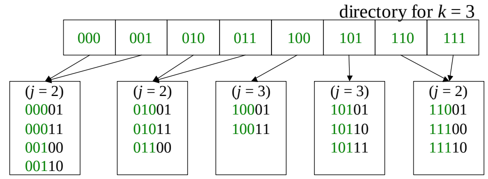
- Optimizes to reduce disk accesses
- Each hash bucket fits on one disk block
- Better than B-Trees if order is not important – *why?*

Table contains:

- Buckets, each fitting in one disk block, with the data
- A directory is used to hash to the correct bucket

Extendible Hash Table

- Directory entry: *key prefix* (first *k* bits) and a pointer to the bucket with all keys starting with its prefix
- Each bucket contains keys matching on first *j* ? *k* bits, plus the value associated with each key

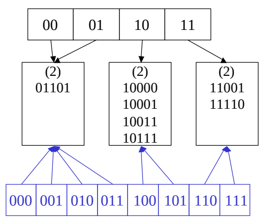


insert(11011)?

insert(11011)?

Splitting the Directory

1. insert(10010)
   But, no room to insert and *no adoption!*
2. Solution: **Expand directory**
3. Then, it's just a normal split.



How to ensure this uncommon?

If Extendible Hashing Doesn't Cut It

Option 1: Store only pointers/references to the items: (key, value) pairs are in disk

Option 2: Improve Hash + Rehash

The One-Slide Hash

Hash function: maps keys to integers

Collision resolution

- Separate Chaining
  - Expand beyond hashtable via secondary Dictionaries
  - Allows ? > 1
- Open Addressing
  - Expand within hashtable
  - Secondary probing: {linear, quadratic, double hash}
  - ? ? 1 (by definition!)
  - ? ? ½ (by preference!)

Choosing a Hash Function

- Make sure table size is prime
- Careful choice for strings
- **"Perfect hashing"**
  - If keys known in advance, tune hash function for them!

Rehashing

- Tunes up hashtable when ? crosses the line

Extendible hashing

- For disk-based data
- Combine with B-tree directory if needed

Implementations So Far

	insert	find	delete
• Unsorted list	O(1)	O(n)	O(n)
• Sorted list	O(n)	O(log n)?	O(n)
• Trees	O(log n)	O(log n)	O(log n)
• Hash Table	O(1)	O(1)	O(1)

Is there anything a hash table can't do fast?