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

Hannah Tang and Brian Tjaden
Summer Quarter 2002

Reminder: Dictionary ADT

- Dictionary operations
  - insert
  - find
  - delete

insert
• Darth
- wouldn't want to meet him in a dark alley

find(Tjaden)
• Tjaden
- who ever heard of a name spelled with "tj"

- Tjaden
  - who ever heard of a name spelled with "tj"
- Hannah
  - does she really have an "h" at the end of her name
- Albert
  - what's his relation to Albert S. Wong again

- Stores *values* associated with user-specified *keys*
  - values may be any (homogenous) type
  - keys may be any (homogenous) comparable type

Implementations So Far

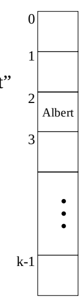
	insert	find	delete
• Unsorted list	O(1)	O(n)	O(n)
• Sorted list	O(n)	O(log n)?	O(n)
• Trees	O(log n)	O(log n)	O(log n)

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

Hash Table Goal

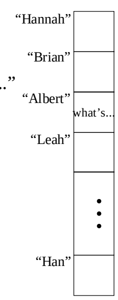
We can do:

a[2] = "Albert"

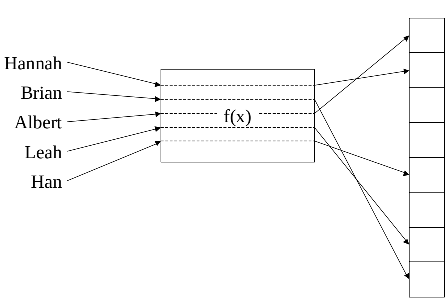


We want to do:

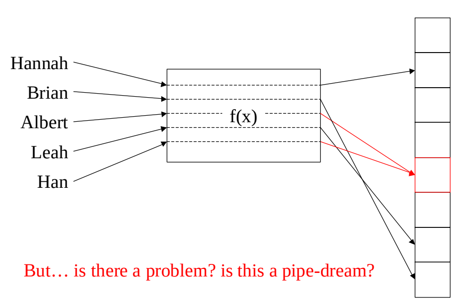
a["Albert"] = "what's..."



Hash Table Approach



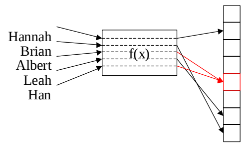
Hash Table Approach



But… is there a problem? is this a pipe-dream?

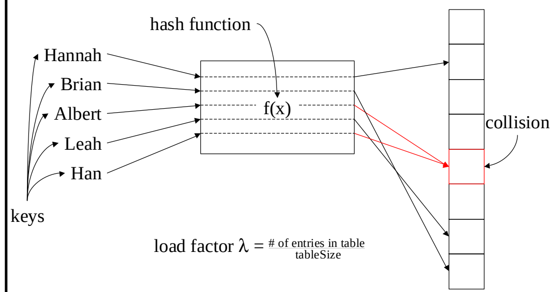
Hash Table
Dictionary Data Structure

- Hash function: maps keys to integers
  - result: can quickly find the right spot for a given entry



What if we have a sparse unordered table?
Can we efficiently list all entries?

Hash Table Terminology



Hash Table Code
First Pass

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

What should the hash function be?

How should we resolve collisions?

What should the table size be?

A Good Hash Function…

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

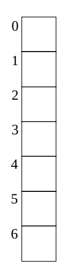
…distributes the data evenly (hash(a) % size ≠ hash(b) % size).

…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)



Good Hash Function for Strings?

- Let $s = s_1 s_2 s_3 s_4 \ldots s_n$:
  - hash(s) = ASCII(s_1) + ASCII(s_2) + ... + ASCII(s_n)

Problems?

Making the String Hash Easy to Compute

- Use Horner's Rule

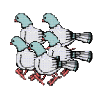
```
int hash(String s) {
  h = 0;
  for (i = s.length() - 1; i >= 0; i--) {
    h = (s_i + 128*h) % tableSize;
  }
  return h;
}
```

How to Design a Hash Function

- Know what your keys are
- Study how your keys are distributed
- Try to include all important information in a key in the construction of its hash
- Try to make "neighboring" keys hash to very different places
- Prune the features used to create the hash until it runs "fast enough" (very application dependent)

Collisions

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



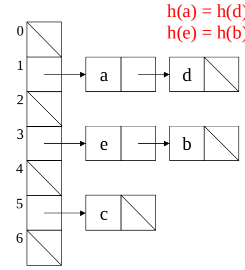
- What do we do when two keys hash to the same entry?
  - open hashing: put little dictionaries in each entry

    shove extra pigeons in one hole!
  - closed hashing: pick a next entry to try

Open Hashing or Hashing with Chaining

- Put a little dictionary at each entry
  - choose type as appropriate
  - common case is unordered linked list (chain)
- Properties
  - λ can be greater than 1
  - performance degrades with length of chains

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



Open Hashing Code

```
Dictionary findBucket(Key k) {
  return table[hash(k)%table.size];
}

Value find(Key k)
{
  return findBucket(k).find(k);
}

void delete(Key k)
{
  findBucket(k).delete(k);
}

void insert(Key k, Value v)
{
  findBucket(k).insert(k,v);
}
```

Load Factor in Open Hashing

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

Closed Hashing *or* 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 *go in another spot*

- Properties
  - λ ≤ 1
  - performance degrades with difficulty of finding right spot

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

0	
1	a
2	d
3	e
4	b
5	c
6	

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
  - if i reaches size - 1, the probe has failed
  - depending on f(), the probe may fail sooner
  - long sequences of probes are costly!

Linear Probing

f(i) = i

- Probe sequence is
  - h(k) mod size
  - h(k) + 1 mod size
  - h(k) + 2 mod size
  - …
- findEntry using linear probing:

```
bool findEntry(Key k, Entry entry) {
  int probePoint = hash(k);
  do {
    entry = table[probePoint];
    probePoint = (probePoint + 1) % size;
  } while (!entry.isEmpty() && entry.getKey() != k);
  return !entry.isEmpty();
}
```

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

Load Factor in Linear Probing

- For *any* λ < 1, linear probing will find an empty slot
- Search cost (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 λ > 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
  - …
- findEntry using quadratic probing:

```
bool findEntry(Key k, Entry entry) {
  int probePoint = hash(k), numProbes = 0;
  do {
    entry = table[probePoint];
    numProbes++;
    probePoint = (probePoint + 2*numProbes - 1) % size;
  } while (!entry.isEmpty() && entry.getKey() != key);
  return !entry.isEmpty();
}
```

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			48	47	47
1					
2				5	5
3					55
4					
5		40	40	40	40
6	76	76	76	76	76
probes:	1	1	2	3	3

Quadratic Probing Example ☹

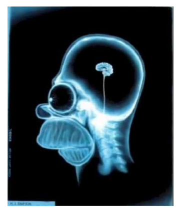
	insert(76) $76\%7 = 6$	insert(93) $93\%7 = 2$	insert(40) $40\%7 = 5$	insert(35) $35\%7 = 0$	insert(47) $47\%7 = 5$
0				35	35
1					
2		93	93	93	93
3					
4					
5			40	40	40
6	76	76	76	76	76
probes:	1	1	1	1	∞

Quadratic Probing Succeeds (for $\lambda \le ½$)

- If size is prime and $\lambda \le ½$, 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 $i \ne j$
    $(h(x) + i^2) \bmod size \ne (h(x) + j^2) \bmod size$
  - by contradiction: suppose that for some $i \ne j$:
    $(h(x) + i^2) \bmod size = (h(x) + j^2) \bmod size$
    $i^2 \bmod size = j^2 \bmod size$
    $(i^2 - j^2) \bmod size = 0$
    $[(i + j)(i - j)] \bmod size = 0$
  - but how can $i + j = 0$ or $i + j = size$ when $i \ne j$ and $i,j \le size/2$?
  - same for $i - j \bmod size = 0$

Quadratic Probing May Fail (for $\lambda > ½$)

- For any *i* larger than size/2, there is some *j* smaller than *i* that adds with *i* to equal size (or a multiple of size). D'oh!



Load Factor in Quadratic Probing

- For *any* $\lambda \le ½$, quadratic probing will find an empty slot; for greater λ, quadratic probing *may* find a slot
- Quadratic probing does not suffer from *primary* clustering
- Quadratic probing possibly suffers from *secondary* clustering

Double Hashing

$f(i) = i \cdot hash_2(x)$

- Probe sequence is
  - $h_1(k) \bmod size$
  - $(h_1(k) + 1 \cdot h_2(x)) \bmod size$
  - $(h_1(k) + 2 \cdot h_2(x)) \bmod size$
  - …
- Code for finding the next linear probe:

```
bool findEntry(Key k, Entry entry) {
  int probePoint = hash1(k), hashIncr = hash2(k);
  do {
    entry = table[probePoint];
    probePoint = (probePoint + hashIncr) % size;
  } while (!entry.isEmpty() && entry.getKey() != k);
  return !entry.isEmpty();
}
```

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:
$hash_2(x) = R - (x \bmod R)$

Double Hashing Example

	insert(76) 76%7 = 6	insert(93) 93%7 = 2	insert(40) 40%7 = 5	insert(47) 47%7 = 5, 5 - (47%5) = 3	insert(10) 10%7 = 3	insert(55) 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* $\lambda < 1$, double hashing will find an empty slot (given appropriate table size and $hash_2$)
- Search cost appears to approach optimal (random hash):
  - successful search: $\frac{1}{\lambda}\ln\frac{1}{1-\lambda}$
  - unsuccessful search: $\frac{1}{1-\lambda}$
- No primary clustering and no secondary clustering
- One extra hash calculation

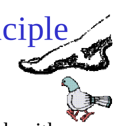
Deletion in Closed Hashing

	insert(7)	delete(2)	find(7)
0	0	0	0
1	1	1	1
2	2	2	
3		7	7
4			
5			
6			

Where is it?!

- Must use lazy deletion!
- On insertion, treat a deleted item as an empty slot

The Squished Pigeon Principle



- An insert using closed hashing *cannot* work with a load factor of 1 or more.
- An insert using closed hashing with quadratic probing may not work with a load factor of ½ or more.
- Whether you use open or closed hashing, 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 λ), rehash all the elements into a new, larger table:
  - takes O(n), but amortized O(1) as long as we (just about) double table size on the resize
  - spreads keys back out, may drastically improve performance
  - gives us a chance to retune parameterized hash functions
  - avoids failure for closed hashing techniques
  - allows arbitrarily large tables starting from a small table
  - clears out lazily deleted items

It’s all about tradeoffs!