Lecture 9: Hash tables

CS 2112 Spring 2012
Andrew Myers

Collection implementations

	Unsorted sets & maps	Resizable arrays (array)	Sorted sets Sorted maps (search tree)	Stacks Queues (array)	Priority queues (tree, heap)
add, push		O(1)	O(lg n)	O(1)	O(lg n)
get, contains, put		O(1)	O(lg n)		
remove, pop		O(1)	O(lg n)	O(1)	O(lg n)

Can we get the O(1) performance of arrays on general keys?

Direct Address Table

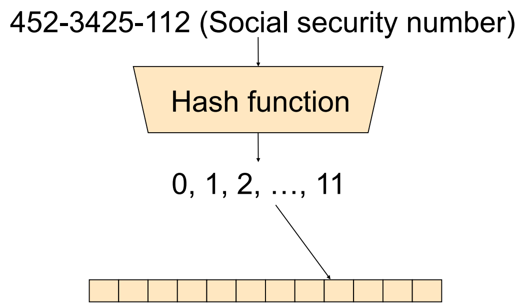
- Want a map from keys to values
- Suppose we can convert keys to different small integers
  - Example: Addresses on my street
    - Start at 1, go to 88
    - A few lots don't have houses
- Make an array as large as the set of keys
- To find an entry, we just index to that entry of the array
  - Use null or special value to indicate absence
- Lookup operations take O(1) time!

Problem

- Want O(1) operations but with general keys
  - E.g., look up employee records by social security #
- Direct address table?
- Problem: too many SS numbers
  - Will have 10,000,000,000 mostly empty entries…

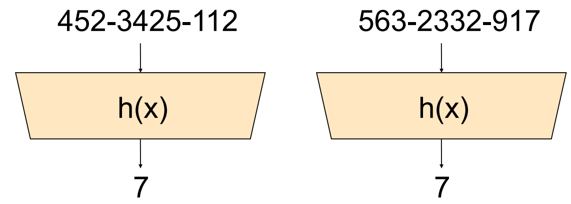
Hash functions

- Idea: define a (cheap) function to map keys onto a small range of array indices ("buckets")
- Given an array of size 12:



Collisions

- Problem: hash function may create collisions between two different keys



1. Cheap but avoids collisions: a function that looks as random as possible
2. Need a way to deal with collisions when they (inevitably) happen

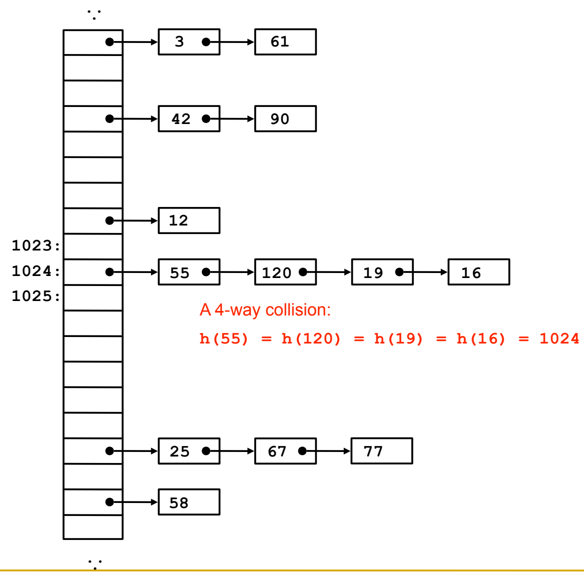
Examples of Hash Functions

`int → {0,1,...,99}`

- Bad: use only part of the key
  - constant functions: hash(x) = 7
  - two most significant digits: hash(379988) = 37
  - two least significant digits: hash(379988) = 88
- Better: Use all the information in the key
  - sum of digit pairs mod 100: hash(379988) = 37+99+88 (mod 100) = 24
  - square number and take middle digits
- Best: Every change to the argument key produces an unpredictable, apparently random change to result
  - MD5 hash function, CRC (cyclic redundancy check) on key data

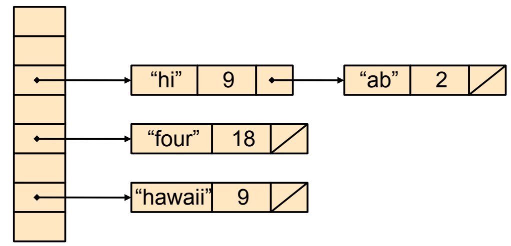
Collision resolution #1

- Chained buckets: array elements are linked lists
- Walk down linked list till you find
- Expected length of linked list is proportional to **load factor**
  - Load factor = # elements / # buckets
  - Good load factor ~ 1-2 for chained buckets



Implementing maps

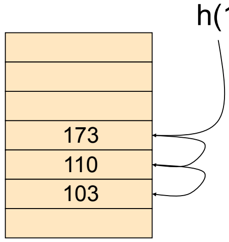
- Map is just a set of key/value pairs
  - A String→int map with chained buckets:



Collision resolution #2: open addressing

- Just use an array of elements.
- If you find the wrong element, search elsewhere in array
- Simple: walk forward till you find it.

h(103) = 103 mod 7



- Better: skip forward by secondary hash function $h_2(x)$
- Also: quadratic hashing…
- Load factor should stay less than 0.75 to avoid long chains

Performance

Affected by many factors:

- Size of array relative to number of data items
  - Consider limit where there is only 1 bucket
  - as bad as simple linked lists!
- Quality of hash function
  - Good hash functions do not lead to clustering of data → low collision rate

Analysis for Hashing with Chaining

- Analyzed in terms of *load factor* λ = n/m = (items in table)/(table size)
- We count the expected number of *probes* (key comparisons)
- Goal: Determine U = number of probes for an *unsuccessful* search

- Claim U is the same as the average number of items per table position = n/m = λ
- Claim S = number of probes for a *successful* search = 1 + λ/2

The hashCode method

- Want to store arbitrary objects, not just integers
- All Java objects have hashCode() method for use by hash tables

```
int hashCode();
```

  - By default: memory address of object

• hashCode needs to capture important information

• hash table can handle information diffusion (randomness)

Pitfalls

- Easy to define a hash function that doesn't seem very random
- E.g., pick the first character of string keys
  - What if all strings have the same first char?
- E.g., use the memory address
  - All addresses = 0 mod 4 or mod 8.
  - Hash table effectively four times as small if modular hashing used with power of two size
  - The Java default…

Some reasonably good hash functions

- Modular hashing: h(k) = k mod m for some m=#buckets
  - But: avoid m = power of 2. Prime m is good
- Multiplicative hashing: h(k) = (ka/2^q) mod m for appropriately chosen values of a, m, and q.
  - Similar to random number generator
  - Multiplier a should be large and "random"
  - q is crucial and typically forgotten
  - Cheaper than modular hashing, works fine with power-of-2 bucket array

Universal Hashing

- Idea: choose randomly from a large collection of hash functions
- Parameterized family of numeric functions
  - e.g., $f_{abc}(x) = ax^2 + bx + c \pmod{100}$
- Pick a,b,c at random!
- Works as well or better than hand-crafted hash functions in most cases!
- Disadvantage: no persistence

Testing a Hash Function

- If bucket i contains x_i elements, then the *clustering* is $(\sum (x_i^2)/n) - n/m$.
- Clustering < 1: hashing is better than random
- Clustering > 1: worse than random
- Clustering = k: roughly k times slower than random
  - E.g., randomly picking every other bucket gives clustering of 2.

Observations

- Hashing is popular in practice because code is easy to write and maintain and performance is typically excellent
- Performance depends on two key factors:
  - load factor λ = number of entries/size of array
  - choice of hash function
  - if λ appropriately small and hash function is chosen well, get expected O(1) complexity for all operations
- Chained hashing is faster, less fragile -- used in Java Collections
  - java.util.HashMap implements java.util.Map
  - java.util.HashSet implements java.util.Set

Table Doubling

- We know each operation takes time $O(\lambda)$ where $\lambda=n/m$
- But isn't $\lambda = \Theta(n)$?
- What's the deal here? It's still linear time!

Table Doubling:

- Set a bound for λ (call it λ_0)
- Whenever λ reaches this bound we
  - Create a new table, twice as big and
  - Re-insert all the data
- Easy to see operations *usually* take time O(1)
  - But sometimes we copy the whole table

Analysis of Table Doubling

- Suppose we reach a state with n items in a table of size m and that we have just completed a table doubling

	Copying Work
Everything has just been copied	n inserts
Half were copied previously	n/2 inserts
Half of those were copied previously	n/4 inserts
…	…
Total work	n + n/2 + n/4 + … = 2n