Hash Tables

- save items in a key-indexed table (index is a function of a key)
- HASH FUNCTION = method of computing array index from key

Issues

- computing the hash function
- equality test: method for checking whether two keys are equal
- collision resolution: two keys that hash to the same array index

* space-time tradeoff

I) Hash Function

- Idealistic goal = scramble the keys uniformly to produce a table index
  - efficiently computable
  - each table index equally likely for

each key

- practical challenge: need a different approach for each key type
- all Java classes inherit a method hashCode() → returns a 32-bit integer
- requirement: if x.equals(y) => x.hashCode() == y.hashCode()
- desirable: if !x.equals(y) => x.hashCode() != y.hashCode()
- default implementation: memory address

eg.

Class Transaction

- String who
- Date when
- double amount

hash code - pick a non-zero constant

```
int hash = 17;
hash = 31 * hash + who.hashCode();
hash = 31 * hash + when.hashCode();
hash = 31 * hash + ((Double) amount).hc();
```

→ typically a small prime

- standard recipe: use the $31x+y$ rule to combine all the fields
  - * null → return 0

* hash code: int between -2^{31} and $2^{31}-1$
* hash function: int between 0 and M-1 (prime or power of two)
  for use as array index

1) `return Math.abs(k.hashCode()) % M;`
   = 1 in a billion bug
2) `return (key.hashCode() & 0x7fffffff) % M;`
   = correct
   `k.hashCode() & Integer.MAX_VALUE`
   clears the high bit

- uniform hashing assumption

each key is equally likely to hash to an integer between 0 and (M-1)

* bins and balls - throw balls uniformly at random into M bins

=> expect 2 balls in the same bin after $\sim \sqrt{\pi M/2}$ tosses

=> every bin has at least 1 ball after $\sim M \ln M$ tosses

II) Separate chaining

Collision : two different keys hash to the same index

- Idea: use an array of linked lists ($M < N$ lists)

- hash : map key to integer between 0 and $M-1$
- insert : put at front of i^{th} chain (if not already there)
- search : need to search i^{th} chain

- if uniform hashing assumption $\Rightarrow$ probability that the number of keys in a list is within a constant factor of N/M is extremely close to 1

$\Rightarrow$ typical choice $\boxed{M \sim N/5}$ constant time ops

III) Linear Probing

- Open addressing : when a new key collides, find the next empty slot and put it there.

M - bigger than the number of keys

we expect ; just use an array

- hash : map key to integer i between 0 and M-1
- insert : put at table index i if free, if not try (i+1), (i+2)...
- search : search table index i ; if occupied but no match, try (i+1), (i+2)
  - 2 arrays < keys / values
  - arrays resizing — should stay half empty
- knuth's parking problem
  - half full => M/2 cars, mean displacement is $\sim \boxed{3/2}$
  - full => with M cars, mean displacement is $\sim \sqrt{\pi M/8}$

Hash Table Context

- for long keys, skip
- eg. strings => only examine 8-5 evenly spaced characters
  - downside → really bad collision patterns
- Denial-of-service attacks => hash collisions
- one-way hash functions
- two-probe hashing
- double hashing
- cuckoo hashing

Set API

- remove the value field from any symbol table implementation