ID1020 2018-09-26 #10

Hash tables

Hash function. Method for computing array index from key.

Use an array of M<N linked lists
(*) Hash: map key to integer i between 0 and M-1
Insert: put at front of ith chain (if not already there)
Search: need to search only ith chain

Goal for hash func.: Scramble the keys uniformly to produce a table index.
- Efficiently computable
- Each table index equally likely for each key. ← problematic

Ex. Phone numbers: Last three digits better than first three.

Java: hashCode() returns 32-bit int.

Requirement: If x.equals(y), then (x.hashCode() == y.hashCode())
Highly desirable: If !x.equals(y), then (x.hashCode() != y.hashCode())

Default: Memmory address
Legal (but poor) to always return same.
Custom implementations Integer, Double, String, File, URL, Date,...
User-defined types. Users are on their own

Caching can be used to improve performance

"Standard" recipe
- Combine each significant field using the 31x+y rule.
- If field is a primitive type, use wrapper type hashCode().
- If field is null, return 0.
- If field is a reference type, use hashCode().
- If field is an array, apply to each entry.

Used in Java libraries

Modular hashing.

Hash code: Int between -2^{31} and $2^{31}-1$
Hash function: Int between 0 and M-1 (for use as array index)
↖Typically prime

Uniform hashing assumption: Each key is equally likely to hash to an integer between 0 and M-1

Collisions. Two distinct keys hashing to same index

Seperate-chaining symbol table (* on previous page)

Under uniform hashing assumption, prob. that the number of keys in a list is within a constant factor of N/M is extremely close to 1
- Distribution of list size obeys a binomeal distribution

Consequence: Number of probes for search/insert is proportional to N/M,
- M too large ⇒ too many empty chains
- M too small ⇒ chains too long.
- Typical choice: $M \sim N/4$ ⇒ Constant-time ops.

Resizing in a seperate-chaining hash table

Goal - Average length of list N/M = constant
- Double array size M when $N/M \geq 8$
- Halve size of array M when $N/M \leq 2$
- Need to rehash all keys when resizing

Average case: 3-5 probes

Linear probing

Open addressing - When key collides - find next empty slot

Array size M must be greater than # key-value pairs N

Problem: Clustering, Knuth's parking problem

Under uniform hashing assumption, the average # of probes in a linear probing hash table of size M that contains $N = \alpha M$ keys:

$$\underbrace{\sim \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)}_{\text{hit}} \qquad \underbrace{\sim \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)}_{\text{miss/insert}}$$

M too large: too many empty array entries
M too small: search time blows up
Typical choise: $\alpha = N/M \sim 1/2$

Resizing. Goal: Average length of list $N/M \leq 1/2$.
- Double array size M when $N/M \geq 1/2$
- Halve array size M when $N/M \leq 1/8$
- Need to rehash all keys when resizing

Deleting requires moving entries with same hash after deleted element.

Algorithmic complexity attacks

· One way hash functions
Ex. MD4, MD5, SHA-0, SHA-1, SHA-2, WHIRLPOOL, RIPEMD-160

Applications: Digital fingerprint, message digest, storing passwords.
Too expensive to use in ST implementations

Seperate chaining
- Performance degrades gracefully
- Clustering less sensitive to poorly-designed hash function.

Linear probing
- Less wasted space
- Better cache performance

Two-probe hashing
Double hashing
Cuckoo hashing

Hash tables vs Balanced search trees
- HT simpler to code
- HT faster for simpler keys
- BT has stronger performance guarantee
- BT supports ordered ST operations
- BT easier to implement.

Usage

- Sets
- Dictionary clients
- Indexing clients
- Sparse vectors

Matrix-vector multiplication
- Symbol tables instead of arrays

  - AI
  - Page rank