CSE 332: Data Structures and Parallelism

Fall 2022

Richard Anderson

Lecture 12: Hashing

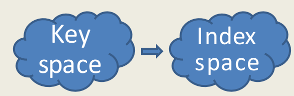
Announcements

- Midterm – Nov 4, in class
  - Coverage: stuff in class, up to the midterm
    - Details of topics will be posted
  - Practice midterms
    - Posted. Different instructors have different styles
  - Review session

Today

- Hashing
  - Arrays for dictionary
  - Key space to array space
  - Dealing with collisions
  - Hash functions
  - Resizing and Load Factors
  - Expected performance

Hashing



- Dictionary ADT
  - Access by key
- Arrays
  - Key is an index
  - O(1) Access
  - What if key space is large
- Hashing
  - Key space mappings

Hashing Implementation

- Separate Chaining

- Open Addressing

- Load factor: λ = N/Tablesize
- Rehashing: double the size of the table

Open Addressing Summary

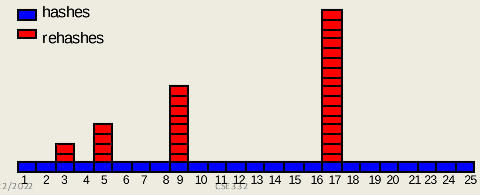
- Does not need extra pointers
- Probe sequence
  - Order of finding open space for a key
    - Linear Probing
    - Quadratic Probing
    - Double Hashing
- Drawbacks
  - Clustering harms run time
  - Deletes are annoying
  - Fails when $\lambda > 1$
  - Can fail when $\lambda > ½$ for quadratic probing

Separate chaining run time

- Average bucket size is λ
- O(1) run time if $\lambda \leq 1$
  - Sort of: worst case is really O(N)
- Controlling load factor
  - If N is known in advance, allocate a hash table of size N
  - If inserts are dynamic, double table size when λ=1

Rehashing

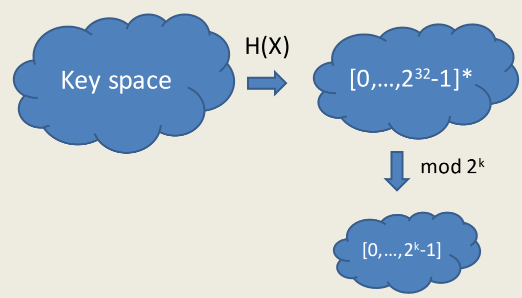
- Cost of rehashing is number of elements in the hash table
- Parameters chosen so rehashing work is about the same as hashing work



Amortized Analysis of Rehashing

- Assume cost of inserting n keys is < 3n
- Suppose $2^k + 1 \leq n \leq 2^{k+1}$
  - Hashes = n
  - Rehashes = $2 + 2^2 + \ldots + 2^k = 2^{k+1} - 2$
  - Total = $n + 2^{k+1} - 2 < 3n$
- Question:
  - Do you need a new hash function every time you rehash?

The correct way to do hashing



* Unsigned arithmetic

Choosing a Hash Function

- Considerations
  - Efficiency
  - Depend on entire input
  - Spread out values
  - Uniform coverage of range
  - Avoid patterns
  - Non-invertable

Efficiency

- For data structure use, H(X) needs to be fast to compute
- Hash tables are competing with balanced trees – need to beat the log N factor
- Bit operations are fasters than arithmetic operations
  - Division is particularly slow

Common choice: aX+b mod p

- Sometimes, mod 2^{32} instead
- Constants can be random, or various recommendations are available
- Fibonacci hashing: a=2654435769
- Generalizations to finite fields
  – Number theory / Algebra

Other approaches – bit hacking

```
unsigned long ElfHash(const unsigned char *s)
{
    unsigned long   h = 0, high;
    while (*s)
    {
        h = (h << 4) + *s++;
        if (high = h & 0xF0000000)
            h ^= high >> 24;
        h &= ~high;
    }
    return h;
}
```


Hashing strings

K = s_0 s_1 s_2 … s_{m-1} (where s_i are chars: $s_i \in [0, 127]$)

1. h(K) = s_0 mod TS
2. h(K) = $\left(\sum_{i=0}^{m-1} s_i\right)$ mod TS
3. h(K) = $\left(\sum_{i=0}^{m-1} s_i \cdot 128^i\right)$ mod TS

Application of a bad hash function

- D = $[0..2^K-1]$
- G : [0..127] -> D (Hashing characters)
- $H(s_1 s_2 \ldots s_j) = \sum_i G(s_i)$

Build an anagram dictionary using H

Multiword hashing

- Hashing $W = w_1 w_2 \ldots w_j$
- Hash each w_i into a result
  – Do in a way that order matters
- D = $[0..2^K-1]$
- G : $[0..\ 2^K-1]$ -> D (Hashing characters)
- $H(w_1 w_2 \ldots w_j) = \sum_i G(w_i + f(i))$

Example Hash Function

```
jenkinsOneAtATimeHash(String key, int keyLength) {
      hash = 0;
      for (i = 0; i < key_len; i++) {
            hash += key[i];
            hash += (hash << 10);
            hash ^= (hash >> 6);
      }
      hash += (hash << 3);
      hash ^= (hash >> 11);
      hash += (hash << 15);
      return hash;
}
```

What would Java do?

- From the source code for Hash Map
- Chained hash table
- Initial size is 64
- Double hash table size when $\lambda = 3/4$
- Hash buckets implemented at Lists – but are converted to balanced trees at size 8
  - Guard against bad data (so O(log n))

Messing with a hash table

- Find a large number of keys that hash to same value
- For a hash function H, find x, such that H(x) = z
- H(x) = (ax + b) mod p

  $z \equiv ax + b \pmod{p} \Rightarrow a^{-1}z - b \equiv x \pmod{p}$

- If we are hashing with to H(x) mod 2^k, we find values where

  $H(x) = 0, 2^k, 2*2^k, 3*2^k, \ldots$

Cryptographic Hash Functions

- Hash functions that are hard to invert, e.g., given z, it is hard to find an x, such that h(x) = z
  - Examples, MD5, SHA-1, SHA-2, SHA-3, . . .
- Cryptographic Hash Functions are expensive to compute, so NOT appropriate for data structures
- Standard use case, store a file of passwords

Expected performance

- Worst case, everything goes in one bucket
- Load factor λ, expected number of items per bucket is λ
- Analysis, hashing N items into a table of size N, assume the hashing is random and independent
- Prob(H(X) = Y) = 1/N
- What is the probability that a particular bucket has j items?

The math: Balls in Bins

- Probability that a bin is empty is $(1 - 1/n)^n$
- Probability that a bin has on element is almost $(1 - 1/n)^n$
- Approximated by a poisson process
- Expected length of the longest chain is O(log n / loglog n)