

CSE 202 - Algorithms

Hashing

Universal Hash Functions

Extendible Hashing

4/24/2003

CSE 202 - Hashing



Dictionaries

Dynamic Set: A set that can grow & shrink over time.

Example - priority queue. (Has "Insert" and "Extract-Max".)

Dictionary:

Elements have a key field (and often other satellite data).

Supports the following operations (S is the dictionary, p points to an element, k is a value that can be a key.)

- $\text{Insert}(S, p)$ - Adds element pointed to by p to S .
 $p.\text{key}$ must have already been initialized.
- $\text{Search}(S, k)$ - Returns a pointer to some element with key field k , or NIL if there are none.
- $\text{Delete}(S, p)$ - Removes element pointed to by p from S .
(Note: Insert & Delete don't change p or what it points to.)

Is a priority queue a dictionary? Or Vice versa??

Details we won't bother with...

- Can two different elements have the same key?
- What happens if you insert an element that is already in the dictionary?

Any choice is OK - but it affects the implementation and unimportant details of the analysis.

Hash table implementation

U - set of possible keys

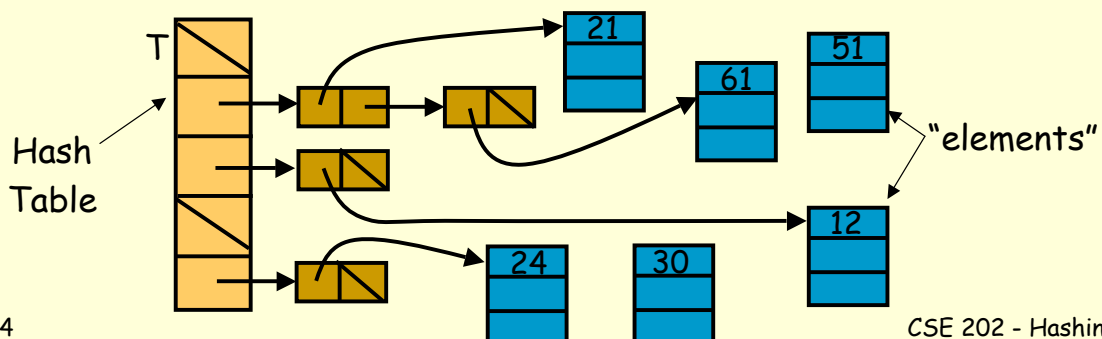
I - indices into an array T (I usually is much smaller than U)

A hash function is any function h from U to I .

Hash table with chaining (i.e. linked list collision resolution):

Each element of T points to a linked list (initially empty).

List $T(i)$ holds pointers to all elements x s.t. $\text{hash}(x.\text{key}) = i$.



Synonyms

Two elements are synonyms if their keys hash to the same value.

Synonyms in a hash table are said to collide.

Hash tables use a collision resolution scheme to handle this.

Some collision resolution methods:

- Chaining (what we just saw)
- $T(i)$ could point to a binary tree.
- Open addressing: $T(i)$ holds only one element.
 - must search $T(i)$, $T(i+1)$, ... until you hit an empty cell.
 - DELETE is difficult to implement well.

We'll stick to chaining.

Speed of Hashing

Given sequence of n requests on empty dictionary

Each request is an Insert, Search, or Delete.

Let k_i be key involved in request i , and $b_i = h(k_i)$.

Time of Request $i < c (1 + \text{number of synonyms of } i \text{ in table}).$

$< c (\text{number of requests } j \text{ s.t. } b_i = b_j).$

- This overcounts when $j > i$.
- and when j isn't an "insert".
- and when j is already in table when i is inserted.
- and when element is deleted before request i .
- Define $X_{ij} = 1$ if $b_i = b_j$, and $X_{ij} = 0$ otherwise.
- Then $\text{Time}(\text{request } i) < c \sum_j X_{ij}$

Complexity Analysis

How should we choose the size of T ?

If $|T| \ll n$, there'll be lots of collisions.

If $|T| \gg n$, it wastes space.

Detail: need to know approximate size of n before you start.

So let's make hash table of size n .

Recall: $\text{Time}(\text{request } i) < c \sum_j X_{ij}$.

so $\text{Time}(\text{all } n \text{ requests}) < c \sum_i \sum_j X_{ij}$.

Thus, expected time $< E(c \sum_i \sum_j X_{ij}) = c \sum_i \sum_j E(X_{ij})$.

If we knew $E(X_{ij}) = 1/n$, we'd know that the average case complexity of processing n requests is $O(n)$.

When is $E(X_{ij}) = 1/n$??

- Assume keys are uniformly distributed:
 - If we make sure that h maps the same number of keys to each index, then the indices will also be uniformly distributed.
 - Easy. For instance, $h(x) = x \bmod |T|$
 - Is "uniformly distributed keys" reasonable?

When is $E(X_{ij}) = 1/n$??

- Assume *indices* are uniformly distributed:
 - In other words, assume the hash function acts like a random number generator.
 - This is a "blame it on someone else" assumption.
 - A standard hash function is: for some well-chosen magic real number a s.t. $0 < a < 1$,
 - Don Knuth says, use $a = .6180339887$
- given an integer x , we compute $h(x)$ by:
- multiply x by a .
 - take result modulo 1 (i.e., keep only the fractional part).
 - multiply this result by $|T|$.

Can WE control the randomization?

- We'd like a "probabilistic" result like the previous "average case" one.
- We can choose a hash function randomly.
 - Sample space = set of hash functions to choose from.
- To ensure $E(X_{ij}) \leq 1/n$, we want:
 - A set of hash functions H from U to $\{0, \dots, n-1\}$,
 - ... such that for all x, y in U (with $x \neq y$),
 - ... the fraction of $h \in H$ s.t. $h(x)=h(y)$ is $\leq 1/n$.
 - actually, to get probabilistic time $O(n)$ for n requests, we only need that this fraction be c/n for some c .

Universal hashing

Def: A set of hash functions H from U to $\{0, \dots, n-1\}$ is universal (or **e-universal**) if,

- for all x, y in A (with $x \neq y$),
 - the fraction of $h \in H$ s.t. $h(x)=h(y)$ is $\leq 1/n$ (or $\leq e$)
- So the definition of universal is exactly what's needed to get probabilistic time $O(n)$.
 - Note that H only needs to do a good job on *pairs* of keys.
 - The book describes one universal set of hash functions (based on $h_{ab}(x) = ax+b \pmod p$.)

This is similar to Knuth's function with a randomly-chosen multiplier, but slightly different.

Polyhash

The world's best hash function (or so I claim):

Allows you to hash long keys efficiently.

Performance guaranty degrades gently as keys get longer.

Choose a finite field, e.g. integers modulo a prime.

Note $p = 2^{31} - 1$ is prime, and $\text{mod } p$ is easy to compute.

For each x in the field, we'll define $h_x(\text{key})$.

Write key as blocks (e.g. halfwords): $\text{key} = a_0 a_1 \dots a_{s-1}$

$$h_x(\text{key}) = a_0 + a_1 x + a_2 x^2 \dots + a_{s-1} x^{s-1}.$$

Polyhash is (s/p) -universal.

Polyhash is (s/p)-universal

s = length of key

p = number of indices

Given $a \neq b$ in U , let $a = a_0 a_1 \dots a_{s-1}$ & $b = b_0 b_1 \dots b_{s-1}$.

For any x in the field, $h_x(a) = h_x(b)$ if and only if

$$a_0 + a_1 x + \dots + a_{s-1} x^{s-1} = b_0 + b_1 x + \dots + b_{s-1} x^{s-1},$$

$$\text{i.e., } (a_0 - b_0) + (a_1 - b_1)x + \dots + (a_{s-1} - b_{s-1}) x^{s-1} = 0.$$

This is a degree $s-1$ (or less) polynomial.

It is not the "all zero" polynomial.

Therefore, it has at most $s-1$ solutions.

(Proof is the same as for real or complex numbers.)

Thus, a & b collide for $< s$ of the p functions $h_0, h_1, \dots, h_{p-1}$.

QED

13

CSE 202 - Hashing

Implementation details (assumes 64-bit integer arithmetic)

- Computing $k \bmod 2^{31}-1$:
 - Write $k = 2^{31}q + r$. (Can be done with shifting.)
 - Note that $2^{31} \equiv 1 \pmod{2^{31}-1}$.
 - Thus $k \equiv q + r \pmod{2^{31}-1}$.
 - This may still be bigger than $2^{31}-1$.
 - It may not matter, or you can repeat.
- Computing polyhash via Horner's rule:
$$a_0 + a_1 x + \dots + a_{s-1} x^{s-1} = a_0 + x (a_1 + x (a_2 + \dots + x(a_{s-1}) \dots))$$

So for each chunk of a , you multiply - add - mod.

14

CSE 202 - Hashing

More polyhash details

- Polyhash can be used on variable-length strings.
 - Stop Horner's rule computation at end of string (rather than going out all the way to s).
 - Beware! There's a SUBTLE BUG.
 - What is it? How do you fix it??
- Polyhash when $|T| < 2^{31} - 1$.
 - Use polyhash to reduce length- s string to 31 bits.
 - Reduce result to $|T|$ using a universal function.
 - Result: $E(X_{ij}) < s/2^{31} + 1/|T|$.
 - For typical parameters (e.g. $|T| = 2^{20}$ and strings are no longer than 2048 bytes long) this gives $E(X_{ij}) < 2/|T|$.

Summary

Hash tables have average time $O(n)$, worst-case $O(n^2)$ time to process any sequence of n dictionary requests (starting with empty set).

Universal hashing says (in theory, at least):

Each time you run the algorithm, after the problem instance has been chosen, choose a random function from a universal set.

Then the expected run time will be $O(n)$.

There are no "bad inputs".

In practice, hash function is usually chosen first.

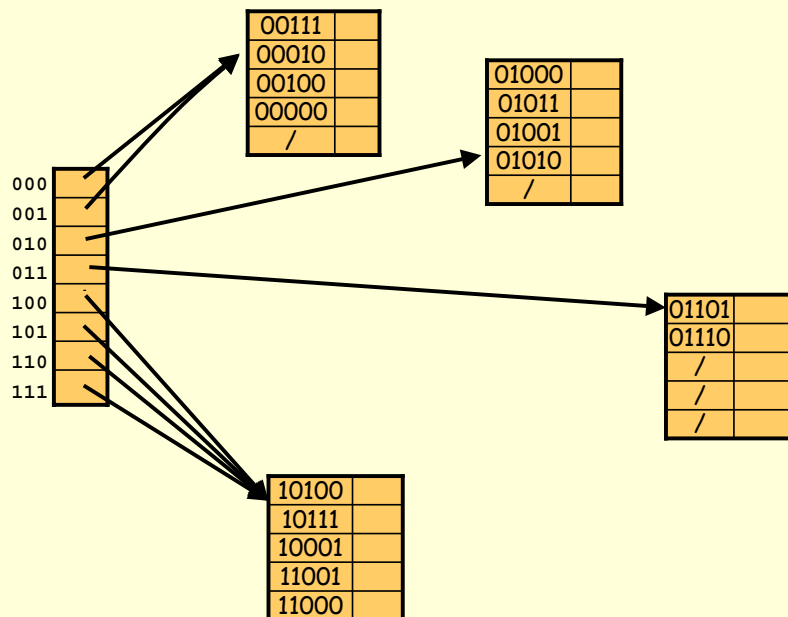
Extendible Hashing

- Hashing is $O(1)$ per request (expected), *provided* the hash table is about the same size as the number of elements.
- Extendible Hashing allows the table size to adjust with the dictionary size.
 - A directory (indexed by first k bits of hash value) points to buckets.
 - k changes dynamically (but infrequently).
 - Each bucket can hold a fixed sized array of elements.
 - Use your favorite method to search within a bucket.
 - When it exceeds the max, the bucket is split in two.
 - The directory is updated as needed.
 - Occasionally, directory needs to double in size (and $k \leftarrow k+1$).

17

CSE 202 - Hashing

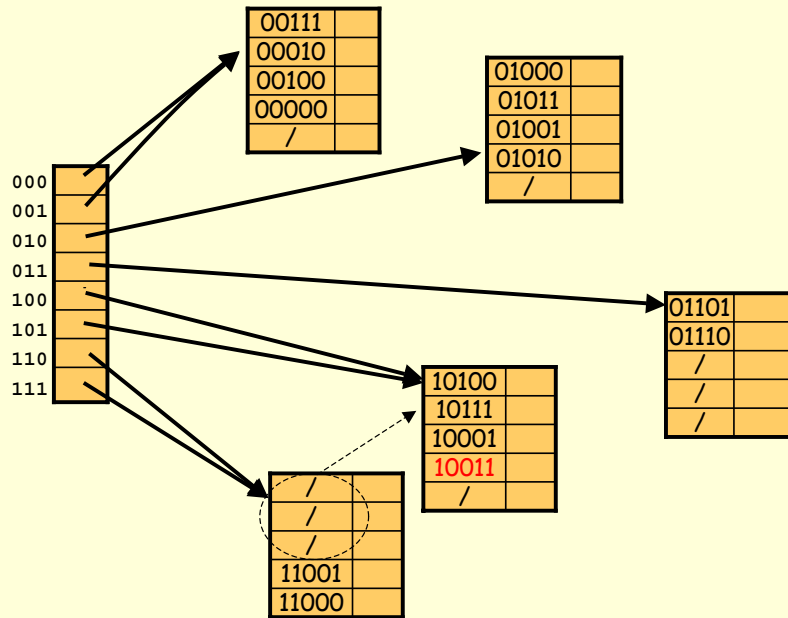
Directory and Buckets



18

CSE 202 - Hashing

Splitting a bucket

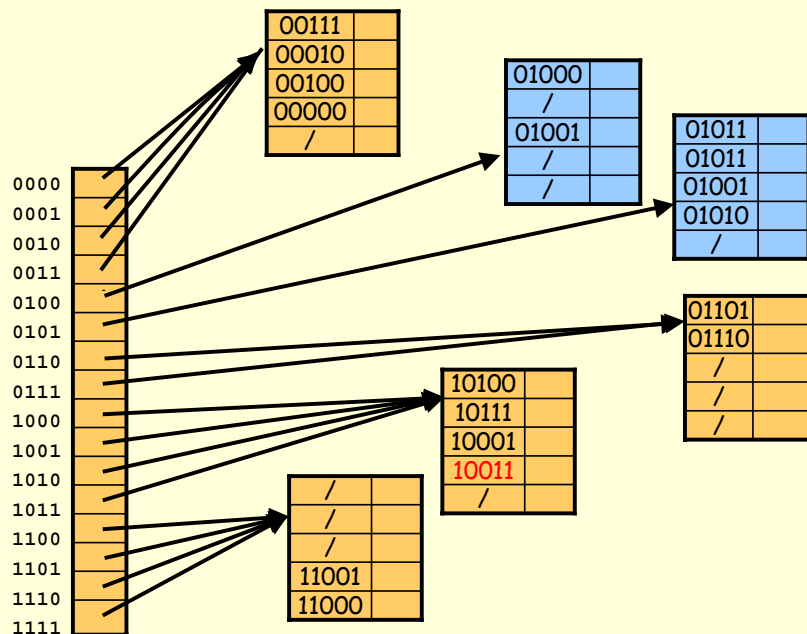


Insert element hashing to **10011** causes bucket split

19

CSE 202 - Hashing

Doubling the directory



Inserting two elements in 010 bucket causes split & directory doubling

20

CSE 202 - Hashing

Extendible hashing analysis

(handwaving version)

- Assumptions:
 - takes 1 time unit to find something in a bucket.
 - takes b time units to split a bucket (b = bucket size).
 - split buckets are each about half full.
 - empty buckets are deleted & directory adjusted.
- Any sequence of n Inserts and Deletes takes time at most $3n$.
 - Each request comes with three 1-unit "coupons".
 - One is used to "pay" for finding the item.
 - Remaining two are "deposited" in the bucket.
 - Half-full bucket on creation will have b coupons when split.
 - This is enough to "pay" for splitting.

21

CSE 202 - Hashing

Extendible handwaving analysis

We could (but won't) improve this analysis:

"Each bucket gets half" assumption could be " $1/4 - 3/4$ ".

- Analysis would use 5 coupons per request.

Split is only rarely worse than 25-75.

- This requires some randomness assumptions.
- Buckets can buy "insurance" against bad breaks.

We could account for shrinking too.

- Paid for by coupons collected on Delete's.

And we can impose a "tax" to pay for resizing the directory.

- Which happens only rarely.

This is an example of amortized analysis.
using "accounting method" (Chapter 17).

22

CSE 202 - Hashing

Extendible hashing in practice

- Databases (e.g. 10^9 elements) stored on disk:
 - Bucket should take one page (e.g. 8KB).
 - Bucket might hold satellite info too.
 - Even with 10^9 elements, directory stays in memory
 - Assuming accesses are frequent.
 - (and if they aren't, who cares?)
 - So there's only one page miss per request.
 - Cost of searching in page is insignificant.
- DRAM-sized hash tables (e.g. 10^6 elements):
 - Bucket can be size of cache line (e.g. 128 Bytes).
 - Directory likely to be in cache.