CSE 326: Data Structures
More Hashing Techniques

Hannah Tang and Brian Tjaden
Summer Quarter 2002

Remember This List?

- How should we resolve collisions?
- What should the table size be?
- What should the hash function be?
- How well does hashing work in the real world?
  - We'll see a case study today!

Hashing Dilemma

Suppose your **WorstEnemy** 1) knows your hash function; 2) gets to decide which keys to send you?

Faced with this enticing possibility, WorstEnemy decides to:

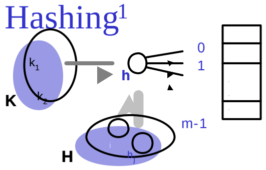
a) Send you keys which maximize collisions for your hash function.
b) Take a nap.

Moral: No *single* hash function can protect you!

Faced with this dilemma, you:

a) Give up and use a linked list for your Dictionary.
b) Drop out of software, and choose a career in fast foods.
c) Run and hide.
d) Proceed to the next slide, in hope of a better alternative.

Universal Hashing[1]



Suppose we have a set **K** of possible keys, and a *finite* set **H** of hash functions that map keys to entries in a hashtable of size m.

Definition:
H is a **universal** collection of hash functions if and only if …

For any two keys k_1, k_2 in K, there are at most |H|/m functions in H for which $h(k_1) = h(k_2)$.

- *So* … if we randomly choose a hash function from H, our chances of collision are no more than if we get to choose hash table entries at random!

[1] Motivation: see previous slide (or visit http://www.burgerking.com/jobs)

Random Hashing – *Not!*

How can we "randomly choose a hash function"?

- Certainly we cannot randomly choose hash functions at runtime, interspersed amongst the inserts, finds, deletes! *Why not?*

- We can, however, randomly choose a hash function each time we *initialize* a new hashtable.

Conclusions

- WorstEnemy never knows which hash function we will choose – neither do we!
- No *single* input (set of keys) can always evoke worst-case behavior

---

Good Hashing:
Universal Hash Function A (UHF_a)

Parameterized by prime table size and vector of *r* integers:

$a = <a_1 \dots a_r>$ where $0 <= a_i < size$

Represent each key as a vector *k* of *r* integers, where $k_i <$ size

- size = 11, key = 39752 ==> <3,9,7,5,2>
- size = 29, key = "hello world" ==> <8,5,12,12,15,23,15,18,12,4>

$$h_a(k) = \left(\sum_{i=0}^{r} a_i k_i\right) \bmod size$$

UHF_a: Example

- Context: hash strings of length 3 in a table of size 131

let a = <35, 100, 21>
h_a(“xyz”) = (35*120 + 100*121 + 21*122) % 131
= 129

Let b = <25, 90, 83>
h_b(“xyz”) = (25*120 + 90*121 + 83*122) % 131
= 43

Thinking about UHF_a

Strengths:
- Works on any type as long as you can map keys to vectors
- If we’re building a static table, we can try many values of the hash vector **`<a>`**
- Random **`<a>`** has guaranteed good properties no matter what we’re hashing

Weaknesses:
- Must choose prime table size larger than any k_i

Good Hashing:
Universal Hash Function B (UHF_b)

Parameterized by *j*, *a*, and *b*:
- *j* * *size* should fit into an `int`
- *a* and *b* must be less than *size*

$$\mathbf{h_{j,a,b}(k) = ((ak + b) \bmod (j*size))/j}$$

UHF_b : Example

Context: hash integers in a table of size 160

Let $j = 32$, $a = 13$, $b = 142$
$h_{j,a,b}(1000)$ = ((13*1000 + 142) % (32*160)) / 32
= (13142 % 5120) / 32
= 2902 / 32
= 90

Let $j = 31$, $a = 82$, $b = 112$
$h_{j,a,b}(1000)$ = ((82*1000 + 112) % (31*160)) / 31
= (82112 % 4960) / 31
= 2752 / 31
= 89

Thinking about UHF_b

Strengths
- If we’re building a static table, we can try many parameter values
- Random *a*,*b* has guaranteed good properties no matter what we’re hashing
- Can choose any size table
- Very efficient if *j* and *size* are powers of 2 - why?

Weaknesses
- Need to turn non-integer keys into integers

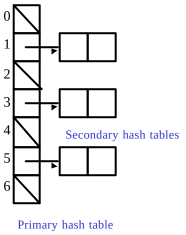
Perfect Hashing

When we know the entire key set in advance …
- Examples: programming language keywords, CD-ROM file list, spelling dictionary, etc.

… then **perfect hashing** lets us achieve:
- Worst-case O(1) time complexity!
- Worst-case O(n) space complexity!

Perfect Hashing Technique

- Static set of *n* known keys
- Separate chaining, two-level hash
- Primary hash table size= *n*
- j^th secondary hash table size= n_j^2
  (where n_j keys hash to slot *j* in primary hash table)
- Universal hash functions in all hash tables
- Conduct (a few!) random trials, until we get collision-free hash functions



Perfect Hashing Theorems[2]

Theorem: If we store n keys in a hash table of size n^2 using a randomly chosen universal hash function, then the probability of any collision is < ½.

Theorem: If we store n keys in a hash table of size m=n using a randomly chosen universal hash function, then

$$E\left[\sum_{j=0}^{m-1} n_j^2\right] < 2n$$

where n_j is the number of keys hashing to slot j.

Corollary: If we store n keys in a hash table of size m=n using a randomly chosen universal hash function and we set the size of each secondary hash table to $m_j=n_j^2$, then:

a) The probability that the total storage used for all secondary hash tables exceeds 4n is less than ½.
b) The expected amount of storage required for all secondary hash tables is less than 2n.

[2] *Intro to Algorithms*, 2nd ed. Cormen, Leiserson, Rivest, Stein

---

Perfect Hashing Conclusions

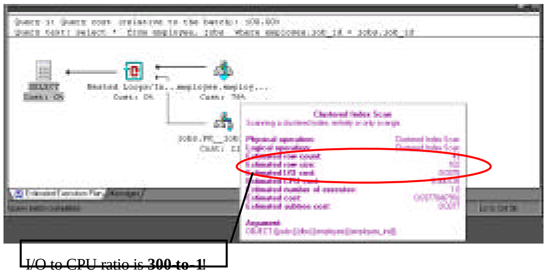
Perfect hashing theorems set tight expected bounds on sizes and collision behavior of all the hash tables (primary and all secondaries).

→ Conduct a few random trials of universal hash functions, by simply varying UHF parameters, until we get a set of UHFs and associated table sizes which deliver …

- Worst-case O(1) time complexity!
- Worst-case O(n) space complexity!

Extendible Hashing:

Cost of a Database Query



I/O to CPU ratio is **300-to-1**!

Extendible Hashing

Hashing technique for huge data sets

- Optimizes to reduce disk accesses
- Each hash bucket fits on one disk block
- Better than B-Trees if order is not important – *why?*

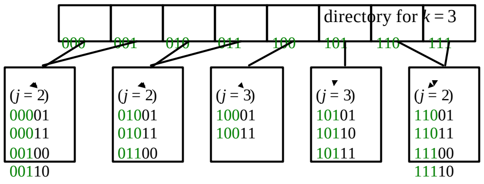
Table contains:

- Buckets, each fitting in one disk block, with the data
- A directory that fits in one disk block is used to hash to the correct bucket

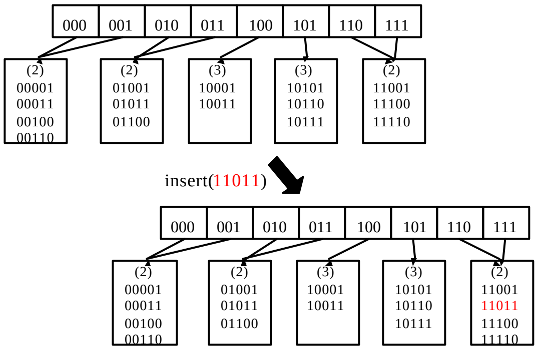
---

Extendible Hash Table

- Directory entry: *key prefix* (first *k* bits) and a pointer to the bucket with all keys starting with its prefix
- Each bucket contains keys matching on first *j* ≤ *k* bits, plus the value associated with each key



Inserting (easy case)


000 001 010 011 100 101 110 111

(2) 00001 00011 00100 00110

(2) 01001 01011 01100

(3) 10001 10011

(3) 10101 10110 10111

(2) 11001 11100 11110

insert(11011)

000 001 010 011 100 101 110 111

(2) 00001 00011 00100 00110

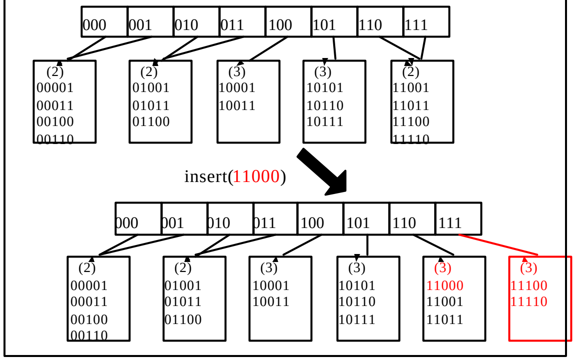
(2) 01001 01011 01100

(3) 10001 10011

(3) 10101 10110 10111

(2) 11001 11011 11100 11110

Splitting a Leaf


000 001 010 011 100 101 110 111

(2) 00001 00011 00100 00110

(2) 01001 01011 01100

(3) 10001 10011

(3) 10101 10110 10111

(2) 11001 11011 11100 11110

insert(11000)

000 001 010 011 100 101 110 111

(2) 00001 00011 00100 00110

(2) 01001 01011 01100

(3) 10001 10011

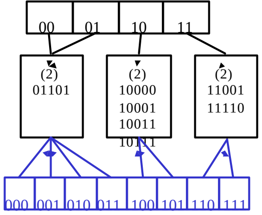
(3) 10101 10110 10111

(3) 11000 11001 11011

(3) 11100 11110

Splitting the Directory

1. insert(10010)
   But, no room to insert and *no adoption!*
2. Solution: **Expand directory**
3. Then, it's just a normal split.


00 01 10 11

(2) 01101

(2) 10000 10001 10011 10111

(2) 11001 11110

000 001 010 011 100 101 110 111

If Extendible Hashing Doesn't Cut It

Store only pointers/references to the items: (key, value) pairs are in disk
- + (Potentially) much smaller M
- + Fewer items in the directory
- – One extra disk access!

Rehash
- + Potentially better distribution over the buckets
- + Fewer unnecessary items in the directory
- – Can't solve the problem if there's simply too much data

What if these don't work?
- – Use a B-Tree to store the directory!

Hash Wrap-up

Hash function: maps keys to integers; table size should be prime

Collision resolution
- Separate Chaining
  - Expand beyond hashtable via secondary Dictionaries
  - Allows λ > 1
- Open Addressing
  - Expand within hashtable
  - Secondary probing: {linear, quadratic, double hash}
  - λ ≤ 1 (by definition!)
  - λ ≤ ½ (by preference!)

Choosing a Hash Function
- Universal hashing
  - Guarantees no (always) bad input
- Perfect hashing
  - Requires known, fixed keyset
  - Achieves O(1) time, O(n) space - guaranteed!

- Rehashing
  - Tunes up hashtable when λ crosses the line

Hash Wrap-up (part 2)

- *Also:* Extendible hashing
  - For disk-based data
  - Combine with B-tree directory if needed

Dictionary ADT Wrapup: Case Study

- Your company, Procrastinators Inc., will release its highly hyped word-processing program, *WordMaster2000* (yeah, they're a little behind the times), next month.
- Your highly successful alpha-test was marred by user requests for a spell-checker.
- Your mission: write and test a spell-checker module before *WordMaster2000* is released.
- For now, you only need to worry about the English language, although *WordMaster2000* is successful, you may need to port your spell-checker to other languages/character sets.

Case Study: Assumptions

You will be given a spelling dictionary of English words

- 30,000 words
- Static (ie, does not support adding user-supplied words *yet*)
- Arbitrary(ish) preprocessing time

Practical notes

- Almost all searches are successful – *Why?*
- Words average about 8 characters in length
- 30,000 words at 8 bytes/word ~ .25 MB
- There are *many* regularities in the structure of English words

Case Study:
Design Considerations

Issues:

- Which data structure should we use?
- What are our design goals?

Possible Solutions?