Hashing

CSE 326
Data Structures
Lecture 15

Readings and References

- Reading
  - Chapter 5

Hashing

- Hashing is a family of data structures used to efficiently support insert, delete, find.
- It cannot be used efficently for other operations where the order of data is important. No list-all, range queries, successor, predecessor.

General Idea

- Key space of size M, but we only want to store subset of size N, where N<<M.
  - Keys are identifiers in programs. Compiler keeps track of them in a symbol table.
  - Keys are student names. We want to look up student records quickly by name.
  - Keys are chess configurations in a chess playing program.
  - Keys are URLs in a database of web pages.

Simple Hash Table

T	
0	
1	
2	John Smith
3	
4	Judy Jones
5	
6	Martha Lee
7	Jerry Lee
8	
9	

Hash function:

$h : U \rightarrow \{ 0,1,\ldots,Hsize\ \text{-}1 \}$

U is the universe of keys

h("name") is the hash value of "name"

h(Judy Jones) = 4
h(Jerry Lee) = 7

Find("name") = T[h("name")]

Hashing Properties

- $\text{Load Factor} = \lambda = \dfrac{N}{HSize}$

  - Hash tables may have unused entries $\lambda < 1$
- Good quality hash function distribute data as evenly as possible over the keys.
- Collisions: h(inserted key) = h(existing key).
  - Open hashing - linked lists
  - Closed hashing - find a new place to put inserted key

Good Hash Functions

- Integers: Division method
  - Choose Hsize to be a prime
    $h(n) = n \bmod Hsize$
  - Example. Hsize = 23, h(50) = 4, h(1257) = 15
- Character Strings
  - $x = a_0 a_1 a_2 \ldots a_m$ is a character string. Define int(x) = $a_0 + a_1 128 + a_2 128^2 + \ldots + a_m 128^{m-1}$
    $h(x) = int(x) \bmod Hsize$
  - Compute h(x) using Horner's Rule
    h :=0
    for i = m to 0 by -1 do h := $(a_i + 128h) \bmod Hsize$
    return h

A Bad Hash Function

- Keys able1, able2, able3, able4
  - Hsize = 128
    int(ablex) mod 128 = int(a) = 97
    Thus, h(ablex) =h(abley) for all x and y
- Why use primes for hash table sizes?
  - Primes have no nontrivial divisors
  - Numbers relatively prime to 128 will also work for character strings

Multiplication Method

- Hash function defined by HSize and a floating point number A.
  - Integer case
  - $h(k) = \lfloor HSize * (k*A \bmod 1) \rfloor$
  - Example: HSize = 10, A = .485
    $h(50) = \lfloor 10 * (50*.485 \bmod 1) \rfloor$
    $= \lfloor 10*(24.25 \bmod 1) \rfloor$
    $= \lfloor 10*.25 \rfloor$
    $= 2$
    + HSize need not be prime
    - More computation than division method
- Another alternative – Universal Hashing

What about Collisions?

- Open Hashing - Collisions overflow into linked lists.
  - Load factors > 1 are possible
- Closed Hashing - if a collision occurs find another place in the hash table for the entry.
  - Load factor must be $\leq$ 1

Open Hashing (Chaining)
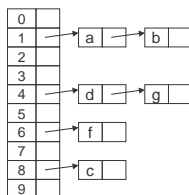
- h(a) = h(b) and h(d) = h(g)
- Chains may be ordered or unordered. Little advantage to ordering.

Open Hashing Properties

- Load factor = λ
  - Unsuccessful searches cost λ comparisons on average
  - Successful searches cost $1 + \lambda/2$ comparisons on average
- Comparisons can be expensive so choosing λ between 1/2 and 1 is wise.

Closed Hashing (Open Addressing)

- No chaining, every key fits in the hash table.
- Probe sequence
  - $h(k)$
  - $(h(k) + f(1))$ mod HSize
  - $(h(k) + f(2))$ mod HSize , …
- Insertion: Find the first probe with an empty slot.
- Find: Find the first probe that equals the query or is empty. Stop at HSize probe, in any case.
- Deletion: lazy deletion is needed. That is, mark locations as deleted, if a deleted key resides there.

Linear Probing

- $f(i) = i$
- Probe sequence
  $h(k)$
  $(h(k) + 1)$ mod HSize
  $(h(k) + 2)$ mod HSize …
- Insertion (assuming $\lambda < 1$)
```
h := h(k)
while T(h) not empty do
  h := (h + 1) mod HSize;
insert k in T(h)
```

Linear Probing Example

76		93		40		47		10		55	

							47		47		47
0		0		0		0	47	0	47	0	47
1		1		1		1		1		1	55
2		2	93	2	93	2	93	2	93	2	93
3		3		3		3		3	10	3	10
4		4		4		4		4		4	
5		5		5	40	5	40	5	40	5	40
6	76	6	76	6	76	6	76	6	76	6	76

Probes   1   1   1   3   1   3

Performance of Linear Probing

- If there is an available slot linear probing will find it.
- For large hash tables the expected number of probes on insertion is:

$$\frac{1}{2}\left(1+\frac{1}{(1-\lambda)^2}\right)$$

- The expected number of probes on successful searches is:

$$\frac{1}{2}\left(1+\frac{1}{1-\lambda}\right)$$

- Linear probing suffers from primary clustering.
- Not a good idea to use linear probing with $\lambda > \frac{1}{2}$.
- Lazy deletion needed.

Linear Probing – Clustering

no collision

no collision   collision in small cluster

collision in large cluster
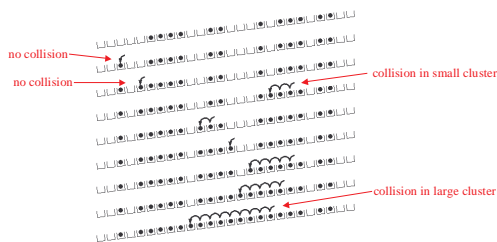
[R. Sedgewick]

Quadratic Probing

- $f(i) = i^2$
- Probe sequence
  $h(k)$
  $(h(k) + 1)$ mod HSize
  $(h(k) + 4)$ mod HSize
  $(h(k) + 9)$ mod HSize, …
- Insertion (assuming $\lambda < 1/2$)
```
h := h(k);
i := 0;
while T(h) not empty do {
  h := (h + 2*i + 1) mod HSize;
  i := i + 1 }
insert k in T(h)
```

Note: $(i+1)^2 - i^2 = 2i + 1$

Quadratic Probing Works for $\lambda < 1/2$

- If HSize is prime then $(h(x) + i^2)$ mod HSize $\neq (h(x) + j^2)$ mod HSize for $i \neq j$ and $0 \leq i,j < $ HSize/2.
- Proof

  $(h(x) + i^2)$ mod HSize $= (h(x) + j^2)$ mod HSize

  $(h(x) + i^2) - (h(x) + j^2)$ mod HSize $= 0$

  $(i^2 - j^2)$ mod HSize $= 0$

  $(i-j)(i+j)$ mod HSize $= 0$

  $\Rightarrow\Leftarrow$ HSize does not divide $(i-j)$ or $(i+j)$

Quadratic Probing may Fail if $\lambda \geq 1/2$

0	
1	
2	16
3	45
4	59
5	
6	76

51

51 mod 7 = 2 ; i = 0

(2 + 1) mod 7 = 3; i = 1

(3 + 3) mod 7 = 6; i = 2

(6 + 5) mod 7 = 4; i = 3

(4 + 7) mod 7 = 4; i = 4

(4 + 9) mod 7 = 6; i = 5

(6 + 11) mod 7 = 3; i = 6

(3 + 13) mod 7 = 2, i = 7

…

Performance of Quadratic Probing

- Although quadratic probing can fail for $\lambda \geq \frac{1}{2}$, it is not likely to do so. We can use load factors greater than ½, but load factors close to 1 should be avoided.
- Quadratic hashing does not suffer from primary clustering, but has only minor secondary clustering.
- With load factors near ½ the expected number of probes per successful search is about 1.5.
- Lazy deletion must be used.

Double Hashing

- $f(i) = i \ g(k)$ where g is a second hash function
- Probe sequence

  h(k)

  (h(k) + g(k)) mod HSize

  (h(k) + 2g(k)) mod HSize

  (h(k) + 3g(k)) mod HSize, …
- In choosing g care must be taken so that it never evaluates to 0.
- A good choice for g is to choose a prime R < HSize and let $g(k) = R - (k$ mod $R)$.

Double Hashing Example

h(k) = k mod 7 and g(k) = 5 – (k mod 5)

	76		93		40		47		10		55
0		0		0		0		0		0	
1		1		1		1	47	1	47	1	47
2		2	93	2	93	2	93	2	93	2	93
3		3		3		3		3	10	3	10
4		4		4		4		4		4	55
5		5		5	40	5	40	5	40	5	40
6	76	6	76	6	76	6	76	6	76	6	76

Probes 1   1   1   2   1   2

Double Hashing is Safe for $\lambda < 1$

- Let $h(k) = k$ mod p and $g(k) = q - (k$ mod $q)$ where $2 < q < p$ and p and q are primes. The probe sequence $h(k) + ig(k)$ mod p probes every entry of the hash table.

  Let $0 \leq m < p$, $h = h(k)$, and $g = g(k)$. We show that h+ig mod p = m for some i. 0 < g < p, so g and p are relatively prime. By extended Euclid's algorithm that are s and t such that

  sg + tp = 1. Choose i = (m-h)s mod p

  (h + ig) mod p =

  (h + (m-h)sg) mod p =

  (h + (m-h)sg + (m-h)tp) mod p =

  (h + (m-h)(sg + tp) mod p =

  (h + (m-h)) mod p = m mod p = m

Deletion in Hashing

- Open hashing (chaining) – no problem
- Closed hashing – must do lazy deletion. Deleted keys are marked as deleted.
  - Find: done normally
  - Insert: treat marked slot as an empty slot and fill it.

$h(k) = k \bmod 7$
Linear probing

Find 59

				Insert 30
0		0		
1		1		
2	16	2	16	
3	23	3	30	
4	59	4	59	
5		5		
6	76	6	76	

Rehashing

- Build a bigger hash table of approximately twice the size when λ exceeds a particular value
  - Go through old hash table, ignoring items marked deleted
  - Recompute hash value for each non-deleted key and put the item in new position in new table
  - Cannot just copy data from old table because the bigger table has a new hash function
- Running time is O(N) but happens very infrequently
  - Not good for real-time safety critical applications

Rehashing Example

- Open hashing – $h_1(x) = x \bmod 5$ rehashes to $h_2(x) = x \bmod 11$.

$\lambda = 1$

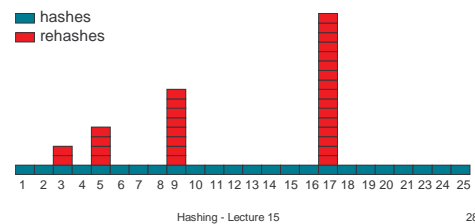| 0	1	2	3	4
25		37	83	
		52	98	

$\lambda = 5/11$

0	1	2	3	4	5	6	7	8	9	10
		25	37			83		52		98

Rehashing Picture

- Starting with table of size 2, double when load factor > 1.

Amortized Analysis of Rehashing

- Cost of inserting n keys is < 3n
- $2^k + 1 \leq n \leq 2^{k+1}$
  - Hashes = n
  - Rehashes = $2 + 2^2 + \ldots + 2^k = 2^{k+1} - 2$
  - Total = $n + 2^{k+1} - 2 < 3n$
- Example
 - n = 33, Total = 33 + 64 −2 = 95 < 99

Case Study

- Spelling Dictionary - 30,000 words
- Goals
  - Fast spell checking
  - Minimal storage
- Possible solutions
  - Sorted array and binary search
  - Open hashing (chaining)
  - Closed hashing with linear probing
- Notes
  - Almost all searches are successful
  - 30,000 word average 8 bytes per word, 240,000 bytes
  - Pointers are 4 bytes

Storage

- Assume word are stored as strings and entries in the arrays are pointers to the strings.

Binary search   Open hashing   Closed hashing

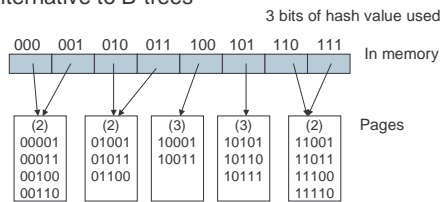N pointers

N/λ + 2N pointers
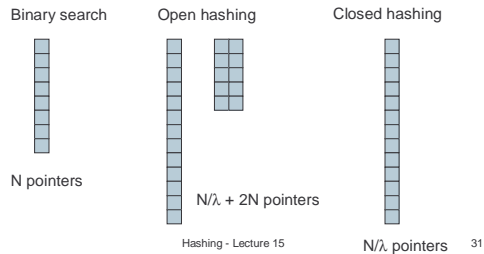
N/λ pointers

Analysis

- Binary Search
  - Storage = N pointers + words = 360,000 bytes
  - Time = $\log_2 N \leq 15$ probes in worst case
- Open hashing
  - Storage = 2N + N/ λ pointers + words
    λ = 1 implies 600,000 bytes
  - Time = 1 + λ/2 probes per access
    λ = 1 implies 1.5 probes per access
- Closed hashing
  - Storage = N/ λ pointers + words
    λ = 1/2 implies 480,000 bytes
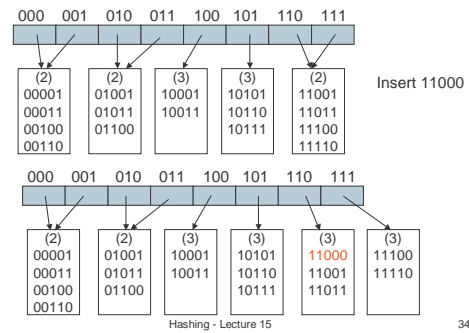 - Time = (1/2)(1+1/(1-λ)) probes
    λ = 1/2 implies 1.5 probes per access

Extendible Hashing

- Extendible hashing is a technique for storing large data sets that do not fit in memory.
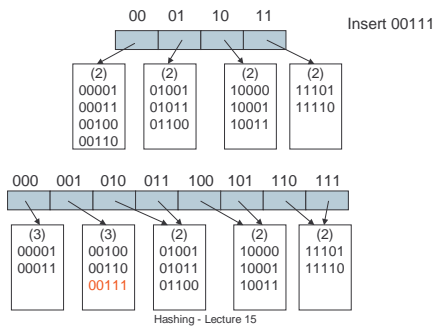- An alternative to B-trees

3 bits of hash value used

000 001 010 011 100 101 110 111   In memory

(2)   (2)   (3)   (3)   (2)   Pages
00001 01001 10001 10101 11001
00011 01011 10011 10110 11011
00100 01100   10111 11100
00110   11110

Splitting

000 001 010 011 100 101 110 111

(2)   (2)   (3)   (3)   (2)   Insert 11000
00001 01001 10001 10101 11001
00011 01011 10011 10110 11011
00100 01100   10111 11100
00110   11110

000 001 010 011 100 101 110 111

(2)   (2)   (3)   (3)   (3)   (3)
00001 01001 10001 10101 11000 11100
00011 01011 10011 10110 11001 11110
00100 01100   10111 11011
00110

Rehashing

00   01   10   11   Insert 00111

(2)   (2)   (2)   (2)
00001 01001 10000 11101
00011 01011 10001 11110
00100 01100 10011
00110

000 001 010 011 100 101 110 111

(3)   (3)   (2)   (2)   (2)
00001 00100 01001 10000 11101
00011 00110 01011 10001 11110
       00111 01100 10011

Analysis of Extendible Hashing

- On deletion neighbors can be merged.
- If table uses k bits but all pages use k-1 bits then rehashing to a smaller table can be done. Not normally an issue with large databases.
- Rehashing does not touch pages.
- Splitting and merging touch only two pages.

Fingerprints

- Given a string x we want a fingerprint x' with the properties.
  - x' is short, say 128 bits
  - Given $x \neq y$ the probability that x' = y' is infintesimal (almost zero)
  - Computing x' is very fast
- MD5 - Message Digest Algorithm 5 is a recognized standard
- Applications in databases and cryptography

Fingerprint Math

Given 128 bits and N strings what is the probability that the fingerprints of two strings coincide?

$$1 - \frac{2^{128}(2^{128} - 1) \cdots (2^{128} - N + 1)}{(2^{128})^N}$$

This is essentially zero for $N < 2^{40}$.

Hashing Summary

- Hashing is one of the most important data structures.
- Hashing has many applications where operations are limited to find, insert, and delete.
- Dynamic hash tables have good amortized complexity.
- Extendible hashing is useful in databases.
- Fingerprints good for databases and crypto.