CSE 326
Lecture 15

Spring Quarter 1998

Quadratic Probing

- $f(i) = i^2$ is quadratic probing
- Probe sequence is:
  $h_0 = h(x)$,
  $h_1 = (h(x) + 1) \bmod HSize$,
  $h_2 = (h(x) + 4) \bmod HSize$,
  $h_3 = (h(x) + 9) \bmod HSize$,
  ..
- Note that the $h_{i+1} = h_i + 2i + 1$
- Quadratic probing insertion is guaranteed to find an available slot provided the load factor is < 1/2 and HSize is prime.

Quadratic Probing Search Algorithm

- Assume assuming T has load factor < 1/2.
- Search for x

```
h := h(x);
i := 0;
while T(h) is not empty do
   if T(h) = x then return(h)
   else  {
        h := (h + 2i + 1) mod HSize;
        i := i +1 }}
endwhile
```

- Example

```
h := h(50) = 50 mod 11  = 6
h := (h + 1) mod 11 = 7
h := (h + 3) mod 11 = 10
h := (h + 5) mod 11 = 4
h := (h + 7) mod 11 = 0
```


Quadratic Probing Finds a Slot if λ <1/2

- If HSize is prime then one of the slots $(h(x) + i^2) \bmod HSize$ is available for $0 \leq i < HSize/2$.

  It is sufficient to argue that $(h(x) + i^2) \bmod HSize$ is different than $(h(x) + j^2) \bmod HSize$ for all i different from j with $0 \leq i,j < HSize/2$.

  Suppose not. Then

  $h(x) + i^2 = h(x) + j^2 \pmod{HSize}$

  $i^2 - j^2 = 0 \pmod{HSize}$

  $(i - j)(i + j) = 0 \pmod{HSize}$

  But HSize does not divide either i - j or i + j.

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

- 51 cannot be fit into this hash table with hash function h(x) = x mod 7.

0	
1	
2	16
3	45
4	59
5	
6	76

h(51) = 51 mod 7 = 2
(2 + 1) mod 7 = 3
(2 + 4) mod 7 = 6
(2 + 9) mod 7 = 4
(2 + 16) mod 7 = 4
(2 + 25) mod 7 = 6
(2 + 36) mod 7 = 3
(2 + 49) mod 7 = 2
.
.
.

Performance of Quadratic Probing

- Although quadratic probing can fail if $\lambda > 1/2$ it is not likely to do so. We can use load factors greater than 1/2, but, as usual, avoid load factors near 1.
- Quadratic probing does not suffer from primary clustering, but only has relatively minor secondary clustering.
- With load factors near 1/2 the expected number of probes per successful search is about 1.5.
- Again lazy deletion must be used.

Double Hashing

- f(i) = i h_2(x) where h_2 is a second hash function.
- Probe sequence is:
  h_1(x),
  (h_1(x) + h_2(x)) mod HSize,
  (h_1(x) + 2 h_2(x)) mod HSize,
  (h_1(x) + 3 h_2(x)) mod HSize,
- In choosing h_2 care must be taken that it never evaluates to 0, other wise there is no real probe sequence. One bad choice is h_2(x) = x mod p.
- A good choice for h_2 is to choose a prime R < HSize and let h_2(x) = R - (x mod R).

Double Hashing Example

- h_1(x) = x mod 7 and h_2(x) = 5 - (x mod 5)

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

Double Hashing is Safe for $\lambda < 1$

- Let $h_1(x) = x \bmod p$ and $h_2(x) = q - (x \bmod q)$ where $2 < q < p$ and q and p are primes. Then the probe sequence $(h_1(x) + i\, h_2(x)) \bmod p$ probes every entry of the hash table.
- Proof:

  Let $0 \leq k < p$. Let $h_1 = h_1(x)$ and $h_2 = h_2(x)$.
  We must show that $(h_1 + i\, h_2) \bmod p = k$ for some i.
  $0 < h_2 < p$ so that h_2 and p are relatively prime.
  By extended Euclid algorithm there exist s and t such that $s\, h_2 + t\, p = 1$.
  Choose $i = (k - h_1)\, s \bmod p$
  $(h_1 + i\, h_2) \bmod p =$
  $(h_1 + (k - h_1)\, s\, h_2) \bmod p =$
  $(h_1 + (k - h_1)\, s\, h_2 + (k - h_1)\, t\, p) \bmod p =$
  $(h_1 + (k - h_1)\, (s\, h_2 + t\, p)) \bmod p =$
  $(h_1 + k - h_1) \bmod p =$
  k

Deletion in Hash Tables

- Open Hashing (chaining) - Deletion is no problem
- Closed Hashing - Must do lazy deletion, that is, deleted items are marked as deleted.

  - Search is done normally in the various probing schemes.
  - Insertion - treat a deleted item as empty slot and fill it.

h(x) = x mod 7

linear probling

marked as deleted → (slot 3)

			insert 51
0		0	
1		1	
2	16	2	16
3	45	3	51
4	59	4	59
5		5	
6	76	6	76

Rehashing

- Rehashing means

  1. initializing a new hash table
  2. Scanning the old hash table and reinserting into the new hash table.
- When to rehash

  - an insertion fails in closed hashing
  - too many probes are occuring in closed hashing
  - load factor reaches some λ too close to 1 in closed hashing.
  - load factor reaches 1 in open hashing.
  - too many items marked deleted.
- Although rehashing is costly at the time it is done, it is not expensive in the long run. That is the amortized cost of the operations is still a constant per access on average.

Rehashing Example

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

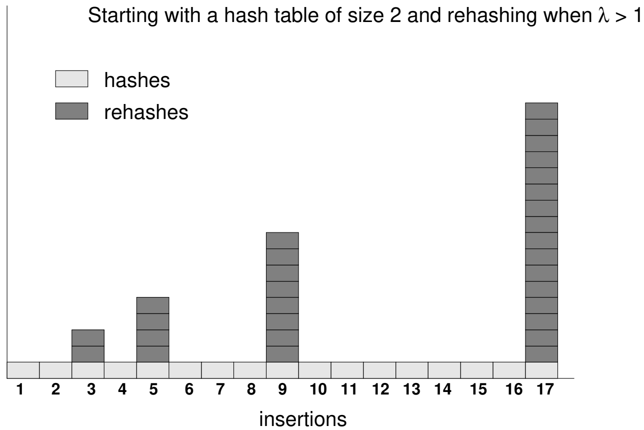
$\lambda=1$

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



Amortized Analysis

- Let us count hash function calculations for doing n insertions where the intital table is size 2. When $\lambda > 1$ hash table size is doubled.

Insertion	Per Oper	Cum
1-2	1	2
3	3	5
4	1	6
5	5	11
6-8	1	14
9	9	23
10-16	1	30
17	17	47
18-32	1	62
33	33	95
34-64	1	126

Analysis

- The cost of inserting n keys is < 3n.
- $2^k+1 \leq n \leq 2^{k+1}$

  Hashes = n

  Rehashes = $2^k + 2^{k-1} + ... + 2 = 2^{k+1} - 2$

  Total = $n + 2^{k+1} - 2 < 3n$
- Example:

  n = 33 Total = 33 + 64 - 2 = 95 < 99

  n = 60 Total = 60 + 64 - 2 = 122 < 180

Case Study

- Spelling dictionary - 30,000 words
- Goals

  - fast spell checking
  - minimal storage
- Possible solutions

  1. sorted array and binary search

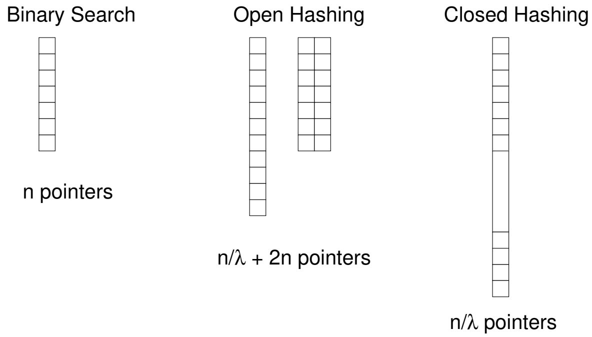
  2. open hashing

  3. closed hashing with linear probing
- Notes

  - Almost all searches are successful
  - 30,000 words at an average of 8 bytes per word is about 1/4 Mbyte
  - pointers are 4 bytes

Storage

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

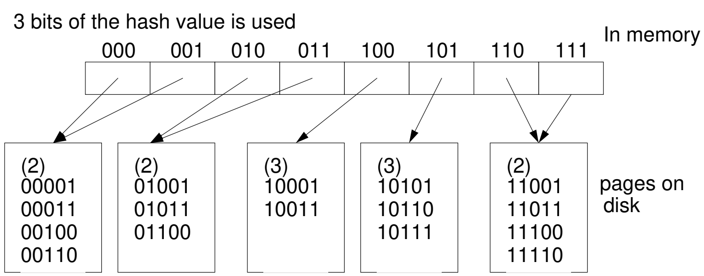


Analysis

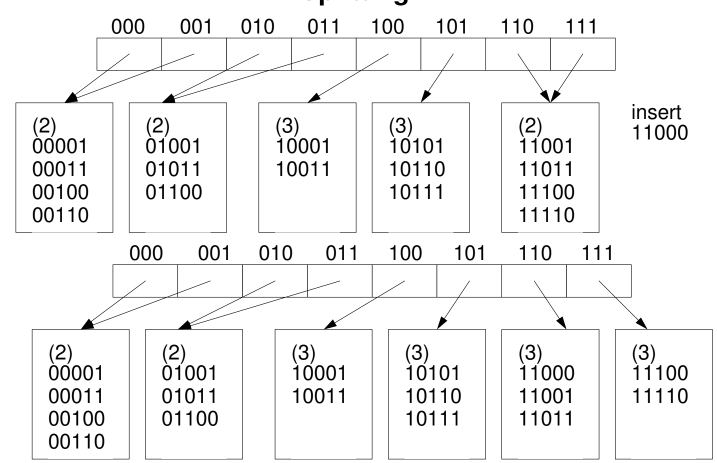
- Binary search
  - storage = n pointers + storage for words
    (360,000 bytes)
  - time = $\log_2 n \leq 15$ probes per access worst case
- Open hashing
  - storage = 2n + n/λ pointers + storage for words
    (λ = 1 implies 600,000 bytes)
  - time = 1 + λ/2 probes per access on average
    (λ = 1 implies 1.5 probes per access)
- Closed hashing using linear probing(λ < 1)
  - storage = n/λ pointers + storage for words
    (λ = .5 implies 480,000 bytes)
  - time = (1/2)(1 + 1/(1 - λ)) probes per access on average
    (λ = .5 implies 1.5 probes per access)

Extendible Hashing

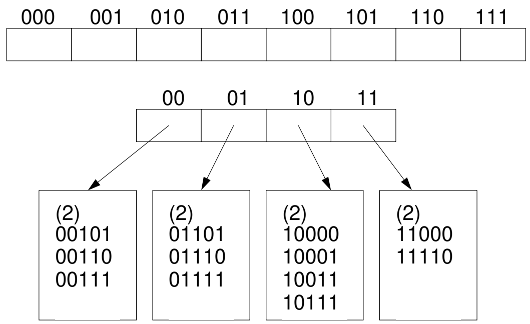
- Extendible hashing is a technique for storing large data sets that do not fit in memory.
- An alternative to B-trees when the order is not important.



Splitting



Rehashing



Rehash of Hashing

- Hashing is an exceptional data structure for storing unordered data that supports insert, delete, and find.
- Both open and closed hashing are useful

  - Open hashing is flexible

  - Closed hashing uses less storage, but performs badly with load factors approaching 1.

  - Entendible hashing for very large data sets in external memory
- Hashing pros and cons

  \+ Very fast
  \+ Simple to implement even with rehashing
  \+ Supports insert, delete, and find
  \- Lazy deletion necessary in closed hashing
  \- Uses extra storage
  \- Does not support range queries or other operations dependent on order
  \- Does not have good locality properties