CSE 326
Lecture 19

Spring Quarter 1995

Hashing

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

  We have a key space of size M and we wish to store N keys where N << M

  Examples:

  1. keys are identifiers in a program and we need to store them in a "symbol table" during compilation.

  2. keys are student names and we wish to store student records using their names to look them up.

Simple Hash Table

T

0	
1	
2	**John Smith**
3	
4	**Judy Jones**
5	
6	**Peter Doe**
7	**Henry Ford**
8	
9	

- Hash function

  h: set of possible keys to {0,1,...,HSize-1}

  h("name") is called the hash value of "name"
- Example:

  h(Judy Jones) = 4
  h(Henry Ford) = 7
- To find the entry for "name" we simply calculate T(h("name")).

Hashing Properties

- Generally, hash tables have some unused entries.

$$LoadFactor = \lambda = \frac{N}{HSize}$$

- A good quality hash function distributes the data as evenly as possible over the hash table.
- Upon insertion we may have a collision, h(inserted key) = h(existing key).

  There are different strategies for dealing with collisions.

Some Good Hash Functions

- Integers:

  Choose HSize to be a prime and h(n) = n mod HSize

  Example: HSize = 23: h(50) = 4, h(1257) = 15, ...
- Character Strings:

  Think of the character string $x=a_0a_1a_2...a_m$ as the integer $int(x) = a_0 + a_1 128 + a_2 128^2 + ... + a_m 128^{m-1}$. We calculate h(x) = int(x) mod HSize.

  This is done using Horner's rule without computing large integers.
- Computing h(x)

  h := 0;
  for i = m to 0 by -1 do h := a_i + 128h mod 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.

Multiplication Method

- Hash function defined by HSize and floating point number A.

  Integer case: h(k) = | HSize (k A mod 1) | where 0 < A < 1.

  Example: HSize = 10, A = 1/3

  h(50) = | 10 (50 x 1/3 mod 1) | = | 10 (16.66... mod 1) | = | 10 x .66...| = 6

  + HSize can be any integer not just a prime

  - Method requires more operations than a simple integer division

Universal Hash Functions

- Universal hash function defined by HSize and a vector
  $a = (a_0, a_1,...,a_r)$ where $0 \leq a_i <$ HSize
- Each key can be represented by r + 1 integers, that is, $k = (k_0, k_1,...,k_r)$ where each component $0 \leq a_i <$ HSize.

$$h_a(k) = \left(\sum_{i=0}^{r} a_i \cdot k_i \right) mod\ HSize$$

- Example: Hashing strings of length 3 in a hash table of size 100.

  Let a = (35,10,21).

  h_a(xyz) = (35x120 + 10x121 + 21x122) mod 100 = 72

  + HSize can be any number bigger than the component of the keys

  + A random universal hash function has guaranteed good properties no matter what set is to be hashed.

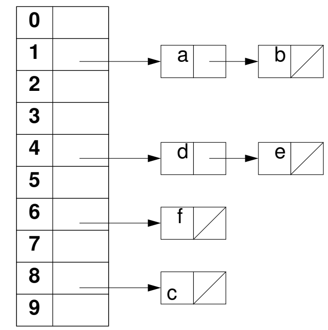
What to do About Collisions

- Open Hashing - Collisions overflow into linked lists

  Load factors > 1 possible.
- Closed Hashing - If a collision occurs find another place in the table for the entry.

  Load factor must be ≤ 1

Open Hashing or Hashing using Chaining



- Collisions h(a) = h(b), h(d) = h(e)
- Choose ordered or unordered chain where appropriate. Common case is the unordered chain because equality checks are easier and there is little advantage with the ordered chain.

Open Hashing Properties

- Load Factor λ

  Unsuccessful search costs λ comparisons on average

  Successful search costs $1 + \lambda/2$ comparisons on average
- Generally, comparisons are to be avoided so choosing λ between 1/2 and 1 is a wise choice.

Closed Hashing (Open Addressing)

- General method of insertion: Let h: keys to {0,...,HSize}

  1. Given a key k we first try to hash to h(k). This is the first "probe".

  2. If h(k) is occupied then we hash to h(k) + f(1) mod HSize. This is the second probe.

  3. If h(k) + f(1) mod HSize is occupied then we hash to h(k) + f(2) mod HSize. This is the third probe.

  Etc.
- The probe sequence is h(k) + f(i) mod HSize for i = 0,1,2,... assuming f(0)=0.

  If i reaches HSize -1 then the insertion fail, but it may happen for some f's that there is still space in the table.

  It is important to avoid long sequences of probes, so that load factors < 1/2 are commonly used in closed hashing.

Linear Probing

- f(i) = i is linear probing
- Probe sequence is:
  h(k), h(k) + 1 mod HSize, h(k) + 2 mod HSize, ...
- Linear probing insertion algorithm, assuming T is not full.

```
compute h = h(k);
while T(h) is not empty do h := h + 1 mod HSize;
T(h) := k
```


Linear Probing Example

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

Performance of Linear Probing

- If there is an available slot then linear probing will find it.
- The expected number of probes on insertion is rougly

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

Examples: $\lambda = .5$: expected number of probes is 2.5
$\lambda = .9$: expected number of probes is 50.5

- Linear probing suffers from primary clustering
- Do not use linear probing if load factor is > 1/2
- Linear probing or any other closed hashing method does not allow physical deletions. We must do lazy deletion.

Quadratic Probing

- $f(i) = i^2$ is quadratic probing
- Probe sequence is:
  h(k), h(k) + 1 mod HSize, h(k) + 4 mod HSize, h(k) + 9 mod HSize, ...
  Note that the $h_{i+1} = h_i + 2i + 1$ where h_0 is the first probe, h_1 is the second probe and so on.
- Quadratic probing insertion is guaranteed to find an available slot provided the load factor is < 1/2.
- Quadratic probing insertion algorithm, assuming T has load factor < 1/2.

```
compute h = h(k);
if T(h) is not available then {
  i := 1;
  h := h + 1;
  while T(h) is not available do {
     h := h + 2i + 1 mod HSize;
     i := i +1 };
T(h) := k
```