Week 11 (Notes)

Analysis of Rehashing

- How many comparisons of keys occur on the average during both successful and unsuccessful search.

Types of Re-Hashing and Probing:

a. Random (Uniform Hashing)

b. Linear

c. Quadratic

Analysis of Random Probing

Prob[hitting an occupied cell in the a hash table]
=

Prob[hitting an empty entry in the a hash table]

=

Average number of comparisons for an unsuccessful search = $U(\lambda)$:

Let k probes are made for an unsuccessful search.

Prob[k unsuccessful searches]

=

$U(\lambda)$=

Analysis of Random Probing

Average number of comparisons for a successful search = *S(λ)*

= Number of unsuccessful searches or insertion steps (averaged over λ)

Why?

Such number depends on how the insertion was done on the first place and hence it depends on the loading factor λ. Approximate such insertion as a continuous function *U(λ)* and find its average

$$S(\lambda) = 1/\lambda \int_0^{\lambda} U(x)\ d(x) = 1/\lambda \int_0^{\lambda} 1/(1-x)\ d(x)$$

$$= 1/\lambda\ \ln\ [1/(1-\lambda)]$$

---> Complexity of search ?

Retrieval from a hash table with 20,000 items in 40,000 possible positions is no slower, on average, than retrieval from a table with 20 items in 40 possible positions.

Results for other Rehashing Methods

Linear: Avg. number of comparisons for:

Unsuccessful search: $1/2\ \left(1 + 1/[(1-\lambda)(1-\lambda)]\right)$

Successful search: $1/2\ \left(1 + 1/(1-\lambda)\right)$

Quadratic: Same as random probing.

Chaining

- Build a linked list of the records whose keys hash to the same address. (simplest method to resolve hash clashes)

Types of Chaining

- Coalesced Hashing
- Separate Chaining

Coalesced Hashing

- **Approach:**

Building a linked list using buckets of the table.

- **Limitation:**
  - Assumes Fixed Table Size

- **Advantages:**
  - Efficient in terms of Probing
  - Deletion is easier

Many variations are possible.

- The amount of time required for a search depends on the length of the lists associated with the items hash bucket.

Example of Coalesced Hashing

Separate Chaining

- This method is useful when items are added to the table, potentially growing beyond table size.
- **Approach:**

Building a linked list for the items hashing to the same value.

- **Advantages:**
  - Efficient in terms of Probing, since list can be ordered (using searching methods of dynamic ordered array)
  - Deletion is easier

- **Limitation:**
  - Extra space for pointers

Analysis of Hashing

Analysis of Chaining:

For an unsuccessful search of a separate chained hash table, each of the buckets is equally likely to be searched so the average time for an unsuccessful search is:

Example:

$(4 + 2 + 2 + 4 + 2 + 2 + 1)/11 = 7/11 = 1.545$

– If the lists are ordered, then we can cut this time in half (average).

– For a successful search, assume each record is equally likely to be sought. 7 elements can be found with 1 probe (operation), 6 with 2, 2 with 3, and 1 with 4.

Analysis of Chaining

Average length of a chain with given *n* elements in the table (excluding the target) = $(n-1)/t$ = λ

The average length of the chain with target

$1+\lambda$

Average # of comparisons for a successful search
= 1+(Average Length without target)/2

So, on the average, $1 + \lambda/2$ comparisons for a successful search.

Efficiency of Hashing Methods
(based on loading factor)

Perfect Hash Function

Given a set of n keys {*k1, k2,… kn*}, a perfect hash function ***h*** satisfies the following property:

h(ki) != ***h(kj)*** for all distinct i and j.

If for n keys, ***h*** fills up only the first n positions, then ***h*** is a **minimal perfect hash function.**

Perfect hash function depends on a given set of keys.

Perfect Hash Function

Ex:

Key Set = 17, 138, 173, 294, 306, 472, 540, 551

h(key) = (key + 25)/64

Hash values: 0, 2, 3, 4, 7, 8, 9, 10

Not Minimal since it requires a table of 11 positions to distribute 9 keys.

h(key) = (key - 7) /72   if key <= 306

h(key) = (key - 42) /72   if key > 306

Hash values: 0,1, 2, 3, 4, 7, 8,

MINIMAL !!

Various polynomial time algorithms exist to find a perfect hash function for a given set of keys.