Week 10 (Notes)

Tables and Hashing

- Objective in searching and insertion: Retrieving a key with 0(1) accesses of a search table.

  -----> Tree methods cannot do it.

- Most Effective Method: Access key i's record as table [key i]. Not usually practical. Why?

- Looking for some function (mapping) f, such that:

  f(key i) ---> j

  where table [j] contains key i's record.

- Static (Fixed Size Table) and Dynamic Hashing

Table Access

- Row and Column Major Indexing
Indexing function??

- Access Table: An auxilliary array to find data stored elsewhere.

Hash Function

- A function that transforms a key into a table index is a hash function

- If r is a record whose key hashes into index j, then j is the hash key of r.

-An Example of a hash function:

Table of size 1000, keys 0..999. **key % 1000**.

Observation: Note two distinct values 75 & 1075 map to the same location using the above hash function -- **collision**.

Hash Function

– Make table (size t) larger than the # of items (n) to insert to reduce possibility of having two keys yielding the same value.

– Partitioning of a hash table into buckets (b) each with s slots. (one slot holds one record). $t=sb$.

Loading Factor: $\lambda = n/t$

Identifier Density: n/T , T is the distinct possible combinations to form a key.

Choosing a Hash Function

Note: A good hash function minimizes collisions and spreads records uniformly through the table. Also it should be easy to compute.

– Should depend on the entire key

Limitation: Items in a hash table are not stored sequentially by key nor is there a practical method for traversing the items in key sequence.

Problem of overflow: A new identifier mapped to a full bucket. When s=1, collision and overflow occurs simultaneously.

Advantage: Close to constant time access of a record given its key and a "perfect hash function"

Some Hash Functions

1. The division method – remainder after dividing by tablesize

    Table of size 1000, keys 0..999.

f(key) = key % 1000.

(Best results happen if tablesize is prime).

2. Midsquare method – key is multiplied by itself and some middle digits of the squared value of the key are used as the index.

3. Folding method – breaks key into parts which are summed or XORed together to give hash value.

4. Random (Uniform) – Use a random number generator with output (hash value) dependent on the key.

Hash Collision and Overflow

A hash collision happens when two distinct keys, k_i and k_j , map to the same location in hash table, i.e. $f(k_i) = f(k_j)$

Collision and Overflow Handling Techniques

(1) **Chaining** - create a linked list of all records that hash to the same location.

(2) **Rehashing:** Use a **rehash function** to relocate the item which can't be placed, and then to locate the item when it wasn't in the location given by the original hash function.

General Requirements for Hashing

- Designing a good hash function

- Resolving Collisions

The hashing (and rehashing) procedure used for insertion is also used for searching

Rehashing (Open Addressing)

If the # of elements to put into table is known in advance, then it may not be worth using the linked list method (why?).

Open addressing hashing methods have been devised to store n records in a table of size t, where $t > n$.

Collision resolution is a key part of these methods uses a **rehash function** to resolve collisions.

Insertion Methods

- Random (Uniform)
- Linear Probing
- Quadratic Probing
- General Rehsahing (Double Hashing)

- Deletion

- Analysis

Random Hashing (Uniform)

Use a pseudo-random number. Seed can be some function of key.

Note: Every key is equally likely to be placed at any empty position of of the hash table.

Linear Probing

The simplest open–addressing method –– When there is a collision, just probe the next position in the table –––––> Simplest Rehashing.

- Given hash function, **key % 1000**;
- Rehash function is **j = (j+1) % 1000**
- where **key % 1000 = j** initially.

3 possible outcomes of linear probe:

(1) the key matches; terminate search successfully.

(2) there is no record present in the spot; terminate search unsuccessfully.

(3) there is a record present and the key doesn't match; probe the next position, continuing until either the key or an empty position is found, or we eventually return to f(x) (Table is full).

Linear Probing

- **Insertion**: After an unsuccessful search, insert it in the empty position found in table.

- **Problem:** Tendency of Clustering. Clustering causes more time for search to find an empty spot.

- **Implementation:** A special key value is required to signal an empty spot in the table.

Quadratic Probing

- Increment function is $\boldsymbol{i^2}$

- Rehash function is $j := (j + \boldsymbol{i^2}) \% 1000$

$$\mathbf{0 \le \boldsymbol{i} \le \boldsymbol{b} - 1}$$

General Rehash Function

- Primary Hash Function h_1

- Secondary Hash Function h_2

h_1 and h_2 should be different functions otherwise a different more complicated clustering phenomenon could occur.

- Example of a Rehash Function: *rh*

$rh(i,key) = (i + h_2(key)) \% \text{Tablesize}$

$i = h_1(key)$

Example

Deletion in Open-Addressing

Given open addressing method, can we easily delete an item from the Hash table?

Suppose we want to delete the item key = 47, with $h(47) = i$.
Can we just delete 47?

Deletion Problem in Open-Addressing

What if we have 3 values that originally hashed to location i, but 2 had to be rehashed to K and L.

47 was the first then placed in location **i**. Subsequent items were placed in **k** and **l** after rehash!

If we simply delete 47, how can we find 33 and 21?

Need to mark a node as deleted rather than as empty so that search can find values rehashed.

Is this a problem for chaining?