EE441 Data Structures
Chapter 14
Hashing

Hashing

- The best sorting time so far was O(nlogn); best search time (binary search) was O(logn).
- Ideal: Organizing n items O(n), and searching one item in O(1) time, i.e., independently of the data volume.
- This would be realized if a *one-to-one mapping of key values to storage addresses* can be designed.
  - e.g. If keys consist of 4-digit integers, simply using keys as addresses in a 10000 item table would work
  - However, this would be very inefficient if the # of items to be stored is much less than 10000
  - Usually the mapping of key values to storage addresses (called HASH FUNCTION) is a many-to-one function.
    - e.g. if only up to 100 items with 4-digit integer keys will be stored, *(key mod 100)* can be used as the hash function.

Hashing

- Hashing: A method of storing records according to their key values.
  - Provides access to stored records in constant time, O(1), so it is comparable to B-trees in searching speed.
  - Therefore, hash tables are used for:
    - Storing a file record by record.
    - Searching for records with certain key values.

- In hash tables, the main idea is to distribute the records uniquely on a table, according to their key values.
  - Take the key and use a function to map it into a location of the array: f(key)=h, where h is the hash address of that record in the hash table.
  - If the size of the table is n, say array [1..n], we have to find a function which will give numbers between 1 and n only.
  - Each entry of the table is called a bucket (storage location).
  - In general, one bucket may contain more than one (say r) records (here, we'll assume r=1 and each bucket holds exactly one record).

Hash Coding: definitions

- Key density:

r: bucket size

n: number of buckets

n*r: hash table size

N: number of possible distinct key values

$$k = \frac{n * r}{N}$$

- Loading factor:

i: number of items in the table

$$lf = \frac{i}{n * r}$$

e.g. items with 4-digit keys stored in a 100-element array:
k=100/10000=0.001

Some Definitions

- Two key values are synonyms with respect to a hash function, f,

if f(key1)=f(key2).

- Synonyms are entered into the same bucket if r>1 and there is space in that bucket.
- When a key is mapped by f into a full bucket, this is an overflow!
- When a key is mapped by f into a storage location (bucket), which is already filled by a different key, this is a collision!

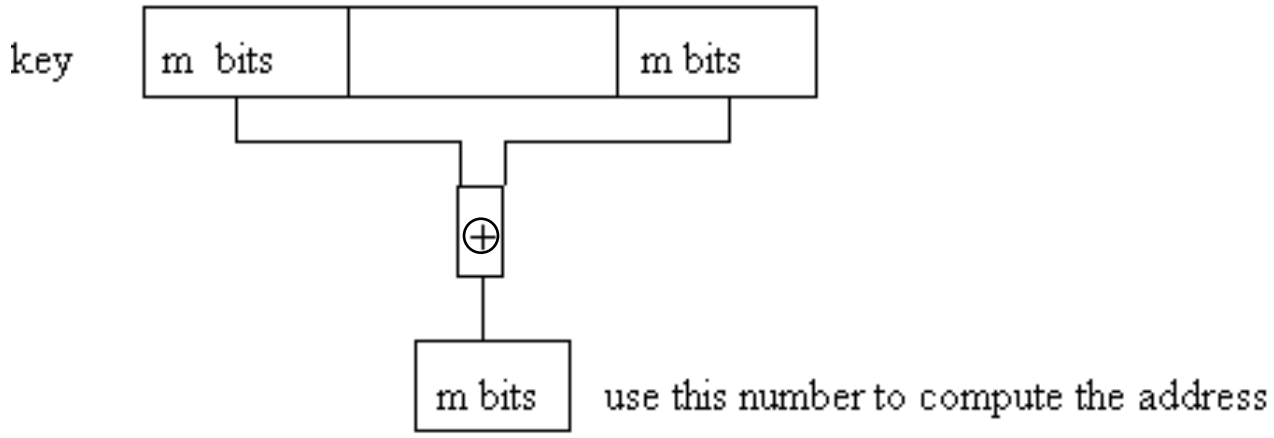
Properties of Hash Functions

- A hash function must:
  - be easy to compute
  - be uniformly distributed (i.e., a random key should have an equal chance of hashing into any one of the n addresses)
  - minimize the number of collisions
  - It is in the form:
    - `int HF(int key)`
    - `int HF(char *key)`

Some Hash Functions

1. f(key)= key mod n

   n is usually chosen as a prime number to ensure that hashing is uniformly distributed
2. To obtain an m-bit address, X-OR the first and last m bits of the key value



NOTE: Here, hash table size is n, which is a power of 2 ($n=2^m$)

Some Hash Functions

3. Mid-squaring:
   - Interpret the key value as numeric (even if it is actually non-numeric)
   - Take the square of the key value
   - Use the middle m-bits of the square as hash address
4. Folding:
   - Partition the key into m-bit integers
   - All except the last part have the same length.
   - These parts are added together to obtain the hash address for the key. There are two ways of doing this addition:
     a) Add the parts directly
     b) Fold at the boundaries.

e.g. key = 12320324111220, part length=3
123|203|241|112|20, then the hash address is
either a) 123+203+241+112+20=699 or b) 123+302+241+211+20=897

Handling Collisions

1. Linear Probing:
   - i. Initially all locations marked ‘empty’
   - ii. When new items are stored, filled locations marked ‘full’
   - iii. In case of collisions, newcomers are stored at the next available location, found via probing by incrementing the pointer (mod n) until either an empty location is found or starting point is reached.

f(key1)=2
f(key2)=2, but location 2 is full.
So, go to the next empty location and store key2 there.
But now if f(key3)=2, another collision!

When searching to find a specific key, search stops when

- The key is found (success)
- An empty location is found (failure)
- Probing returns to the hash address (table completely full and item not found)

NOTE: *Deleted items must be marked `deleted’ and not simply as ‘empty’ because an item stored earlier at a lower position due to collisions must still be found.*

Handling Collisions

2. Random Probing:

When there is a collision, we start a (pseudo) random number generator.

e.g. f(key1)=3, f(key2)=3 →collision!

i. Then, start the pseudo random number generator and get a number, say 7.

ii. Add 3+7=10 and store key2 at location 10.

iii. The pseudo-random number i is generated by using the hash address that causes the collision. It should generate numbers between 1 and n and it should not repeat a number before all the numbers between 1 and n are generated exactly once.

In searching, given the same hash address, for example 3, it will give us the same number 7, so key2 shall be found at location 10.

We carry out the search until:

a) We find the key in the table,

b) Or, until we find an empty bucket, (unsuccessful termination)

c) Or, until we search the table for one sequence and the random number repeats. (unsuccessful termination, table is full)

Handling Collisions

3. Chaining:

- We modify entries of the hash table to hold a key part (and the record) and a link part.
- When there is a collision, we put the second key to any empty place and set the link part of the first key to point to the second one.
- Additional storage is needed for link fields.

2		
3	key1	6
4		
5		
6	key2	

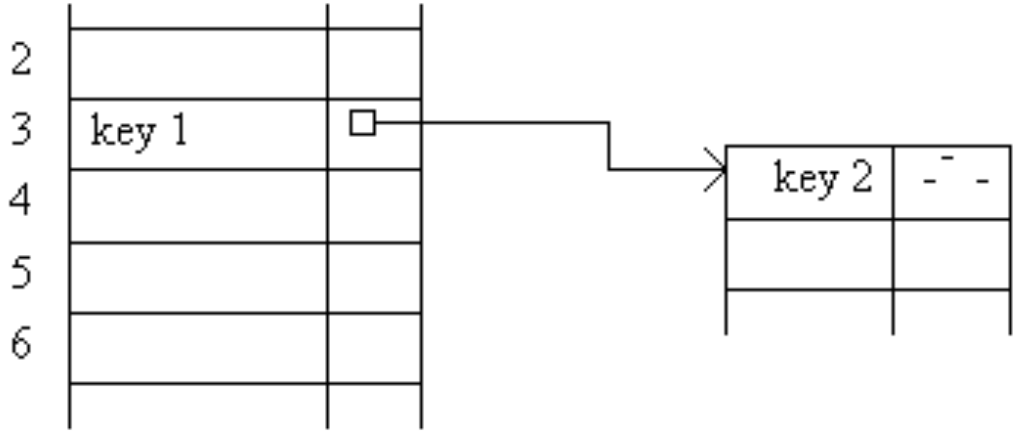
f(key1)=3
f(key2)=3→collision
- Put key2 to bucket 6.

But now what happens if f(key3)=6?
- Take key2 out, put key3 to bucket 6,
- Then put key2 to another available bucket and change link of key1.

Handling Collisions

4. Chaining with Overflow:

In this method, we use extra space for colliding items.



- f(key1)=3 goes into bucket 3
- f(key2)=3 collision, goes into the overflow area

Handling Collisions

5. Rehashing:
   - Use a series of hash functions.
   - If there is a collision, take the second hash function and hash again, etc...
   - The probability that two key values will map to the same address with two different hash functions is very low.

Average number of probes (AVP) calculation :

- Calculate the probability of collisions, then the expected number of collisions, then average. (See Horowitz and Sahni)

1. Linear probing :

$$AVP = \frac{(1- LF/2)}{1-LF}$$

where LF is the loading factor

2. Random probing :

$$AVP = \frac{1}{LF} \ln (1-LF)$$

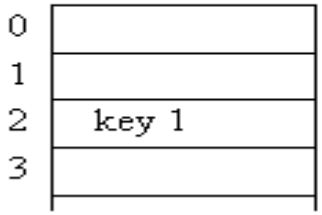
3. Chainig with overflow

$$AVP = 1+ \frac{LF}{2}$$

Handling Collisions

LF	LINEAR P	RANDOM P	CHAINING W.O.
0.1	1.06	1.05	1.05
0.5	1.50	1.39	1.25
0.9	5.50	2.56	1.45

Deleting the key values from HT's :



- To delete key1, we have to put a special sign into location 2, because there might have been collisions, and we can break the chain if we set that bucket to empty.
- However then we will be wasting some empty locations, LF is increased and AVP is increased.
- We cannot increase the hash table size, since the hash function will generate values between 1 and n (or, 0 and n-1).
  - Using an overflow area is one solution!