EE 441 – CH9

HASHING

Instructor: UĞUR HALICI

HASH CODING AND HASH TABLES

Hashing is a method of storing records according to their key values. It 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:

a) Storing a file record by record.

b) 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. We take the key and we use a function to map the key into one 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. In general, one bucket may contain more than one (say r) records. In our discussions we shall assume r=1 and each bucket holds exactly one record.

Let
n: number of buckets
r: bucket size
n*r= hash table size
N is no of distinct possible key values

Definitions:

key density: k = (n*r)/N

loading factor : LF =i/(r*n)

Two key values are synonyms with respect to f, if f(key1)=f(key2).

Synonyms are entered into the same bucket if r>1 and there is space in that bucket.

When two nonidentical keys are mapped into the same bucket, this is a collision.

When a key is mapped by f into a full bucket this is an overflow.

The hash function f,

a) Must be easy to compute,

b) Must be a uniform hash function.

(a random key value should have an equal chance of hashing into any of the n buckets.)

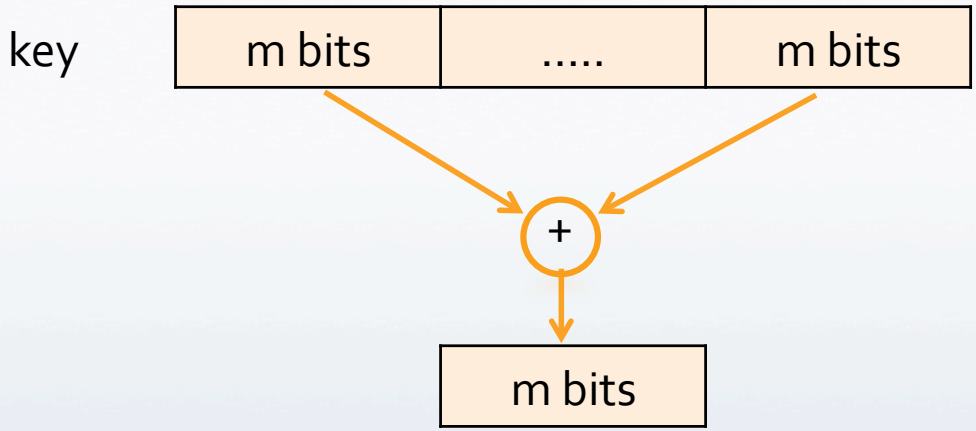
c) Should minimize the number of collisions.

Some hash functions used in practical applications :

1) f(key)=key mod n can be a hash function,

However n should never be a power of 2, n should be a prime a number.

2) Ex-or'ing the first and the last m bits of the key:



Notice that the hash table will now have a size $n=2^m$, which is a power of 2.

3) Mid-squaring:

a) take the square of the key.

b) then use m bits from the middle of the square to compute the hash address.

4) Folding:

The key is partitioned into several parts. All exept 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.

Example: key = 12320324111220, part length=3,

123	203	241	112	20
P1	P2	P3	P4	P5

a)

```
  123
  203
  241
  112
+  20
-----
  699
```

b)

```
  123
  302
  241
  211
+  20
-----
  897
```

Handling Collisions - Overflows :

Consider r=1, so there is one slot per bucket. All slots must be initialized to 'empty' (for instance, zero or minus one may denote empty).

1) Linear Probing:

0	key4
1	
2	
3	
4	
5	key1
6	key2
7	key3

f(key1)=5
f(key2)=5, collision
go to the next empty location (location 6)
and store key2 there
f(key3)=6, collision, store at 7
f(key4)=5, collision, store at 0

- When we reach the end of the table, we go back to location 0.
- Finding the first empty location sometimes takes a lot of time.
-Also, in searching for a specific key value, 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 cycle (unsuccessful termination, table is full)

2) Random probing

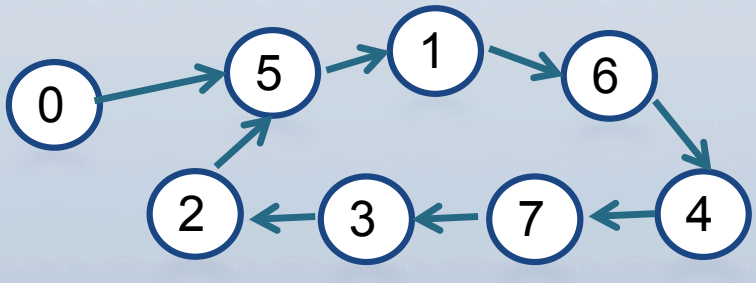
When there is a collision, we start a (pseudo) random number generator. The pseudo-random number i is generated by using the hash address that causes the collision. A new hash address is found by adding the PRN generated to the original address. If the new address results in collision again then PRN generator is called by the previous PRN and the new PRN generated is added to the original hash address to find out a new hash address.

PRN generator should generate numbers between 1 and n-1 and it should not repeat a number before all the numbers between 1 and n-1 are generated exactly once.

That is, PRN generates the pseudo random sequence (h_1, h_2, .., h_{n-1})
such that $PRN(h_1)=h_2$, $PRN(h_1)=h_2$, … $PRN(h_{n-1})=h_1$,
h_i E {1,2, …,n-1}, $h_i \neq h_j$. For completeness assume that $h(0)=h_1$

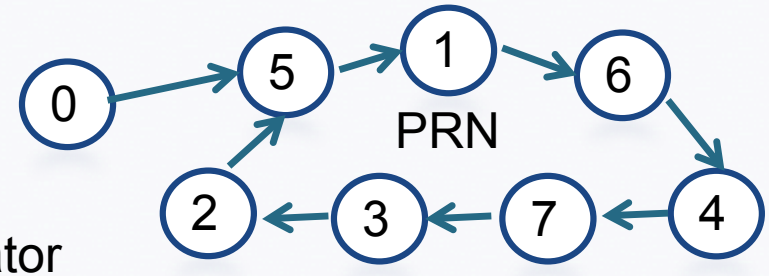
For example consider a hash table of size=8 and the following PRN sequence: (5,1,6,4,7,3,2), i.e.

PRN(5)=1, PRN(1)=6, PRN(6)=4, PRN(4)=7,PRN(7)=3, PRN(3)=2, PRN(2)=5, and also PRN(0)=5



2) Random probing (continues)

0	key4
1	key5
2	
3	key1
4	key6
5	key2
6	key3
7	



f(key1)=3
f(key2)=3 → collision

Then, call PRN generator
PRN(3)=2
Add (3+2)%8=5 and store key2 at location 5.

f(key3)=5 collision
so call PRN(5)=1, (5+1)%8=6, store

f(key4)=3 collision,
call PRN(3)=2, (3+2)%8=5 collision
call PRN(2)=5, (3+5)%8=0, store
f(key5)=0 collision
call PRN(0)=5, (0+5)%8=5 collision
call PRN(5)=1, (0+1)%8=1, store
f(key6)=3 collision
call PRN(3)=2, (3+2)%8=5 collision
call PRN(2)=5, (3+5)%8=0 collision
call PRN(5)=1, (3+1)%8=4 store

2) Random probing (continues)

In searching, say key2, given the same hash address 3, PRN(3) will give us the same number, that is 2, so key2 shall be found at location 5.

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)

3) 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. After the k^{th} hash function, if still there is collision the linear probing or random probing may be used.

0	
1	
2	
3	
4	key3
5	key1
6	key4
7	key2

Example 2 level hash function, then linear probing
f1(key1)=5
f1(key2)=5, collision, use f2(key2)=7;
f1(key3)=5, collision, use f2(key3)=4;
f1(key4)=7, collision, use f2(key4)=4, collision, use linear prob.

- Also, in searching for a specific key value, we have to continue first through the series of hash fuctions, then linear probing

3. Chaining

We modify entries of the hash table to hold a key part (and the record) and a link part. Initially all the link values are null. 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.

0		∧
1		∧
2		∧
3	key1	~~∧~~ ~~6~~ 7
4	key3	∧
5	key2	∧
6	key4	∧
7	key5	~~∧~~ 6

f(key1)=3, store
f(key2)=5, store
f(key3)=4, store

f(key4)=3 →collision,
Put key4 to bucket 6, as if inserted after key1
i.e. copy link for bucket 3 to bucket 6,
adjust link for bucket 3 to point 6

f(key5)=3 →collision,
Put key5 to bucket 7, as if inserted after key1
i.e. copy link for bucket 3 to bucket 7,
adjust link for bucket 3 to point7

But now, what happens if f(key6)=7 ?
key5 with f(key5)=3 was stored there.
So, take key5 out, put key6 to bucket 7, and then put key5 to another available bucket and change link of key1.

4) Chaining with overflow

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

0		∧
1		∧
2		∧
3	key1	~~∧~~ 0 ~~1~~ 3
4	key3	∧
5	key2	∧
6		∧
7	key6	~~∧~~ 2

OA: overflow area

0	key4	∧
1	key5	~~∧~~ 0
2	key7	∧
3	key8	~~∧~~ 1
4		∧
5		∧
6		∧
7		∧

f(key1)=3, store
f(key2)=5, store
f(key3)=4, store

f(key4)=3 →collision,
Put key4 to OA[0], as if inserted after key1
i.e. copy link for bucket 3 to overflow bucket 0,
adjust link for bucket 3 to point overflow 0

f(key5)=3 →collision,
Put key5 to bucket 1, as if inserted after key1
i.e. copy link for bucket 3 to OA[1],
adjust link for bucket 3 to point OA[1];

f(key6)=7, store
f(key7)=7, collision, insert to OA[2];
f(key8)=3, collision, insert to OA[3];

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 = (1-LF/2) / (1-LF)
2. Random probing : AVP = (1/LF)*ln(1-LF)
3. Chaining with overflow: AVP = 1 + (LF/2)

where LF is the loading factor.

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 key values from HT's :

Consider Linear Probing

0	key4
1	
2	
3	
4	
5	key1
6	key2
7	key3

f(key1)=5
f(key2)=5, collision, store at 6
f(key3)=6, collision, store at 7
f(key4)=5 collision, store at 0

To delete key1, we have to put a special sign into location 5 because there might have been collision previously and we may break the chain if we set that bucket to empty.

However then we shall be wasting some empty locations, LF is increased and AVP is increased. We can not 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 a solution