HASHING I

Contents of today's lecture:

- Introduction to Hashing
- Hash functions.
- Distribution of records among addresses, synonyms and collisions.
- Collision resolution by progressive overflow or linear probing.

Reference: FOLK, ZOELLICK AND RICCARDI, File Structures, 1998. Sections 11.1,11.2,11.3,11.5.

Motivation

Hashing is a useful searching technique, which can be used for implementing indexes. The main motivation for Hashing is improving searching time.

Below we show how the search time for Hashing compares to the one for other methods:

- Simple Indexes (using binary search): $O(\log_2 N)$
- B Trees and B+ trees (will see later): $O(\log_k N)$
- Hashing: $O(1)$

What is Hashing ?

The idea is to discover the location of a key by simply examining the key. For that we need to design a hash function.

A **Hash Function** is a function $h(k)$ that transforms a key into an address.
An address space is chosen before hand. For example, we may decide the file will have 1,000 available addresses.

If U is the set of all possible keys, the hash function is from U to {0,1,...,999}, that is

$$h : U \longrightarrow \{0, 1, ..., 999\}$$

Example :

k	**ASCII code for the first 2 letters**	`product`	**h(k) =** `product` **mod 1,000**
BALL	66, 65	$66 \times 65 = 4,290$	290
LOWELL	76, 79	$76 \times 79 = 6,004$	004
TREE	84, 82	$84 \times 82 = 6,888$	888

RRN	FILE
000	
001	
⋮	⋮
004	LOWELL
⋮	⋮
290	BALL
⋮	⋮
888	TREE
⋮	⋮
999	

There is no obvious connection between the key and the location (randomizing).

Two different keys may be sent to the same address generating a **Collision**.

Can you give an example of collision for the hash function in the previous example ?

Answer:
LOWELL, LOCK, OLIVER, and any word with first two letters L and O will be mapped to the same address:

$$h(\text{LOWELL}) = h(\text{LOCK}) = h(\text{OLIVER}) = 4.$$

These keys are called **synonyms**. The address "4" is said to be the **home address** of any of these keys.

Avoiding collisions is extremely difficult **(do you know the birthday paradox?)**, so we need techniques for dealing with it.

Ways of reducing collisions:

1. **Spread out the records** by choosing a good hash function.
2. **Use extra memory**, i.e. increase the size of the address space (Ex: reserve 5,000 available addresses rather than 1,000).
3. **Put more than one record at a single address** (use of buckets).

A Simple Hash Function

To compute this hash function, apply 3 steps :

Step 1: transform the key into a number.

```
LOWELL  =  | L | O | W | E | L | L |   |   |   |   |   |   |
ASCII code: 76  79  87  69  76  76  32 32 32 32 32 32
```

Step 2: fold and add (chop off pieces of the number and add them together) and take the mod by a prime number

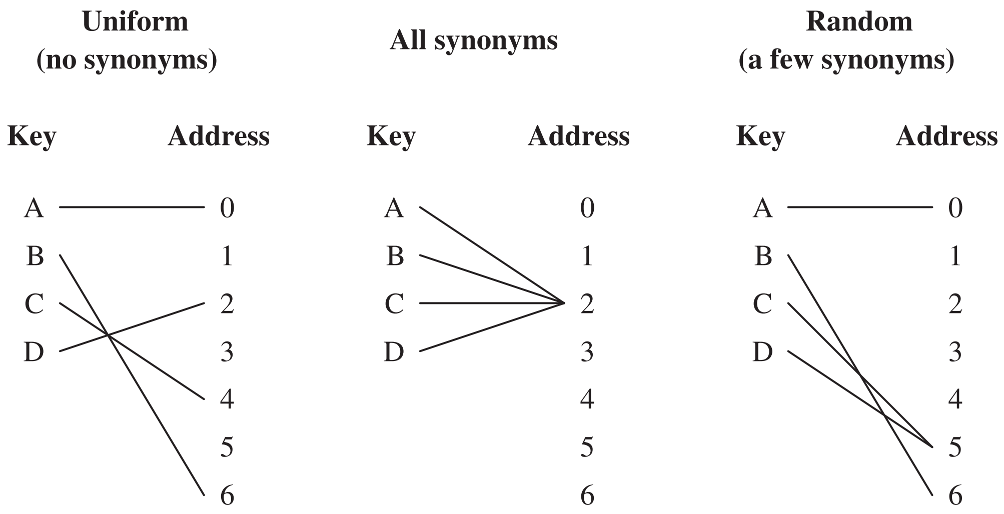
```
7679|8769|7676|3232|3232|3232|

7679+8769+7676+3232+3232+3232 = 33,820

33,820 mod 19937 = 13,883
```

Step 3: divide by the size of the address space (preferably a prime number).

```
13,883 mod 101 = 46
```

Distribution of Records among Addresses

There are 3 possibilities :



Uniform distributions are extremely rare.

Random distributions are acceptable and more easily obtainable.

Trying a **better-than-random** distribution, by preserving natural ordering among the keys :

- Examine keys for patterns.

  Ex: Numerical keys that are spread out naturally such as:
  keys are years between 1970 and 2000.
  $f(year) = (year - 1970) \bmod (2000 - 1970 + 1)$
  $f(1970) = 0, f(1971) = 1, \cdots, f(2000) = 30$

- Fold parts of the key.

  Folding means extracting digits from a key and adding the parts together as in the previous example.
  In some cases, this process may preserve the natural separation of keys, if there is a natural separation.

- Use prime number when dividing the key.

  Dividing by a number is good when there are sequences of consecutive numbers.

  If there are many different sequences of consecutive numbers, dividing by a number that has many small factors may result in lots of collisions. A prime number is a better choice.

When there is no natural separation between keys, try **randomization**.

You can using the following Hash functions:

- **Square the key and take the middle**:

  Ex: key = 453 $\quad$ $453^2 = 205209$
  Extract the middle = 52.
  This address is between 00 and 99.

- **Radix transformation**:

  Transform the number into another base and then divide by the maximum address.

  Ex: Addresses from 0 to 99
  key = 453 in base 11 : 382
  hash address = 382 mod 99 = 85.

Collision Resolution:Progressive Overflow

Progressive overflow/linear probing works as follows :

Insertion of key k:

- Go to the home address of k : h(k)
- If free, place the key there
- If occupied, try the next position until an empty position is found (the 'next' position for the last position is position 0, i.e. wrap around)

Example :

key k	Home address - h(k)
COLE	20
BATES	21
ADAMS	21
DEAN	22
EVANS	20

Complete Table:

0	
1	
2	
⋮	⋮
19	
20	
21	
22	

Table size = 23

Searching for key k:

- Go to the home address of k : h(k)
- If k is in home address, we are done.
- Otherwise try the next position until: key is found or empty space is found or home address is reached (in the last 2 cases, the key is not found)

0	DEAN
1	EVANS
2	
$\vdots$	$\vdots$
19	
20	COLE
21	BATES
22	ADAMS

Ex :
A search for 'EVANS' probes places : 20, 21, 22, 0, 1, finding the record at position 1.

Search for 'MOURA', if h(MOURA)=22, probes places 22, 0, 1, 2 where it concludes 'MOURA' in not in the table.

Search for 'SMITH', if h(SMITH)=19, probes 19, and concludes 'SMITH' in not in the table.

Advantage : Simplicity

Disadvantage : If there are lots of collisions, clusters of records can form, as in the previous example.

Search length

- Number of accesses required to retrieve a record.

```
average search length
= (sum of search lengths)/(numb.of records)
```

In the previous example :

0	DEAN
1	EVANS
2	
⋮	⋮
19	
20	COLE
21	BATES
22	ADAMS

key	**Search Length**
COLE	
BATES	
ADAMS	
DEAN	
EVANS	

Average search length = (1+1+2+2+5)/5 = 2.2.

Refer to figure 11.7 in page 489. It shows that a packing density up to 60% gives an average search length of 2 probes, but higher packing densities make search length to increase rapidly.

HASHING II

Contents of today's lecture:

- Predicting record distribution; packing density.
- Hashing with Buckets
- Implementation issues.

Reference: Folk, Zoellick and Riccardi, File Structures, 1998. Sections 11.3,11.4,11.6

Predicting Record Distribution

Throughout this section we assume a random distribution for the hash function.

Let N = number of available addresses, and
r = number of records to be stored.

Let $p(x)$ be the probability that a given address will have x records assigned to it.

It is easy to see that

$$p(x) = \frac{r!}{(r-x)!x!} \left[1 - \frac{1}{N}\right]^{r-x} \left[\frac{1}{N}\right]^{x}$$

For N and r large enough this can be approximated by:

$$p(x) \sim \frac{(r/N)^x \, e^{-(r/N)}}{x!}$$

Example : $N = 1{,}000$, $r = 1{,}000$

$$p(0) \sim \frac{1^0\, e^{-1}}{0!} = 0.368$$

$$p(1) \sim \frac{1^1\, e^{-1}}{1!} = 0.368$$

$$p(2) \sim \frac{1^2\, e^{-1}}{2!} = 0.184$$

$$p(3) \sim \frac{1^3\, e^{-1}}{3!} = 0.061$$

For N addresses,
the expected number of addresses with x records is

$$N \cdot p(x).$$

Complete the numbers below for the example above:

expected # of addresses with 0 records assigned to it =
expected # of addresses with 1 records assigned to it =
expected # of addresses with 2 records assigned to it =
expected # of addresses with 3 records assigned to it =

Reducing Collision by using more Addresses

Now, we see how to reduce collisions by increasing the number of available addresses.

DEFINITION: **packing density** $= r/N$

500 records to be spread over 1000 addresses result in **packing density** $= 500/1000 = 0.5 = 50\%$.

Some questions :

1. How many addresses go unused ? *More precisely: What is the* **expected** *number of addresses with no key mapped to it?*

   $N \cdot p(0) = 1000 \cdot 0.607 = 607$

2. How many addresses have no synonyms ? *More precisely: What is the expected number of address with only one key mapped to it?*

   $N \cdot p(1) = 1000 \cdot 0.303 = 303$

3. How many addresses contain 2 or more synonyms ? *More precisely: What is the expected number of addresses with two or more keys mapped to it ?*

   $$N \cdot (p(2) + p(3) + ...) = N \cdot (1 - (p(0) + p(1)) = 1000 \cdot 0.09 = 90$$

4. Assuming that only one record can be assigned to an address, how many overflow records are expected ?

   $1 \cdot N \cdot p(2) + 2 \cdot N \cdot p(3) + 3 \cdot N \cdot p(4) + ... =$
   $N \cdot [p(2) + 2 \cdot p(3) + 3 \cdot p(4) + ...] \sim 107.$

   The justification for the above formula is that there is going to be $(i-1)$ overflow records for all the table positions that have i records mapped to it, which are expected to be as many as $N \cdot p(i)$.
   Now, there is a simpler formula derived by students from 2001:

   ```
   expected # of overflow records =
   = (#records) - (expected # of nonoverflow records)
   ```

   $= r - (N \cdot p(1) + N \cdot p(2) + N \cdot p(3) + \ldots)$
   $= r - N \cdot (1 - p(0))$ (since probabilities add up to 1)
   $= N \cdot p(0) - (N - r)$

   ```
   =(expect. # of empty posit. for random hash function)
     - (# of empty positions for perfect hash function)
   ```

   Using this formula we get the same result as before:
   $N \cdot p(0) - (N - r) = 607 - 500 = 107$

5. What is the expected percentage of overflow records ?
   $107/500 = 0.214 = 21.4\%$

Note that using either formula, the percentage of overflow records depend only on the packing density ($PD = r/N$), and not on the individual values of N or r.
Indeed, using the formulas derived in 4., we get that the percentage of overflow records is:

$$\frac{r - N \cdot (1 - p(0))}{r} = 1 - \frac{1}{PD} \cdot (1 - p(0))$$

and the Poisson function that approximate $p(0)$ is a function of r/N which is equal to PD (for hashing without buckets).
So, hashing with packing density $PD = 50\%$ always yield 21% of records stored outside their home addresses.
Thus, we can compute the expected percentage of overflow records, given the packing density:

packing density %	% **overflow records**
10%	4.8%
20%	9.4%
30%	13.6%
40%	17.6%
50%	21.4%
60%	24.8%
70%	28.1%
80%	31.2%
90%	34.1%
100%	36.8%

Hashing with Buckets

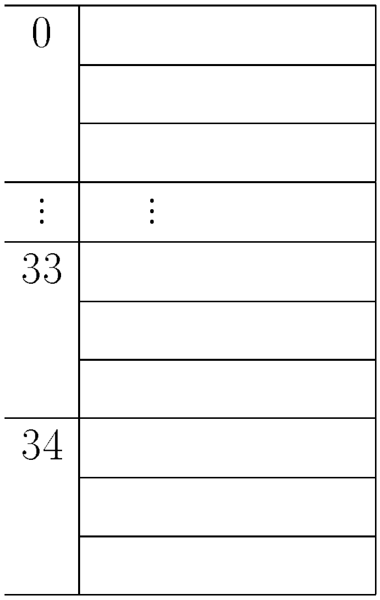
This is a variation of hashed files in which more than one record/key is stored per hash address.

bucket = block of records corresponding to one address in the hash table.

The hash function gives the **Bucket Address**.

Example: for a bucket holding 3 records, insert the following keys:

key	Home Address
LOYD	34
KING	33
LAND	33
MARX	33
NUTT	33
PLUM	34
REES	34



Effects of Buckets on Performance

We should slightly change some formulas:

$$\texttt{packing density} = \frac{r}{b \cdot N}$$

We will compare the following two alternatives:

1. Storing 750 data records into a hashed file with 1,000 addresses, each holding 1 record.

2. Storing 750 data records into a hashed file with 500 bucket addresses, each bucket holding 2 records.

- In both cases the packing density is 0.75 or 75%.
- In the first case r/N=0.75.
  In the second case r/N=1.50.

Estimating the probabilities as defined before:

	p(0)	**p(1)**	**p(2)**	**p(3)**	**p(4)**
1) r/N=0.75 (b=1)	0.472	0.354	0.133	0.033	0.006
2) r/N=1.50 (b=2)	0.223	0.335	0.251	0.126	0.047

Calculating the number of overflow records in each case:

1. **b=1** $(r/N{=}0.75)$:
   Number of overflow records =
   $= N \cdot [1 \cdot p(2) + 2 \cdot p(3) + 3 \cdot p(4) + \ldots]$
   $= r - N\,[p(1) + p(2) + p(3) + \ldots]$ (formula derived last class)
   $= r - N \cdot (1 - p(0))$
   $= 750 - 1000 \cdot (1 - 0.472) = 750 - 528 = 222.$
   This is about 29.6% overflow.

2. **b=2** $(r/N{=}1.5)$:
   Number of overflow records =
   $= N \cdot [1 \cdot p(3) + 2 \cdot p(4) + 3 \cdot p(5) + \ldots]$
   $= r - N \cdot p(1) - 2 \cdot N \cdot [p(2) + p(3) + \ldots]$ (formula for b=2)
   $= r - N \cdot [p(1) + 2[1 - p(0) - p(1)]$
   $= r - N \cdot [2 - 2 \cdot p(0) - p(1)]$
   $= 750 - 500 \cdot [2 - 2 \cdot (0.223) - 0.335] = 140.5 \cong 140.$
   This is about 18.7% overflow.

Indeed, the percentage of collisions for different bucket sizes is:

	Bucket Size				
Packing Density %	**1**	**2**	**5**	**10**	**100**
75%	29.6%	18.7%	8.6%	4.0%	0.0%

Refer to table 11.4 page 495 of the book to see the percentage of collisions for different packing densities and different bucket sizes.

Implementation Issues:

1. Bucket structure

   A Bucket should contain a counter that keeps track of the number of records stored in it. Empty slots in a bucket may be marked '//.../'.
   Ex: Bucket of size 3 holding 2 records:

2	JONES	/////////...///	ARNSWORTH

2. Initializing a file for hashing:

   - Decide on the **Logical Size** (number of available addresses) and on the number of buckets per address.
   - Create a file of empty buckets before storing records. An empty bucket will look like:

0	/////////...///	/////////...///	/////////...///

3. Loading a hash file:

   When inserting a key, remember to:
   - Wrap around when searching for available bucket.
   - Be careful with infinite loops when hash file is full.

HASHING III

Contents of today's lecture:

- Deletions in hashed files.
- Other collision resolution techniques:
  - double hashing,
  - chained progressive overflow,
  - chaining with separate overflow area,
  - scatter tables.
- Patterns of record access.

Reference: FOLK, ZOELLICK AND RICCARDI, File Structures, 1998. Sections 11.7,11.8,11.9

Making Deletions

Deletions in a hashed file have to be made with care.

Record	ADAMS	JONES	MORRIS	SMITH
Home Address	5	6	6	5

Hashed File using Progressive Overflow:

⋮	⋮
4	//////////
5	ADAMS
6	JONES
7	MORRIS
8	SMITH
⋮	⋮

Delete 'MORRIS'

If 'MORRIS' is simply erased, a search for 'SMITH' would be unsuccessful:

⋮	⋮	
4	//////////	$\leftarrow$ empty slot
5	ADAMS	
6	JONES	
7	//////////	$\leftarrow$ empty slot (WRONG: can't find 'SMITH' !!!)
8	SMITH	
⋮	⋮	

Search for 'SMITH' would go to home address (position 5) and when reached 7 it would conclude 'SMITH' is not in the file!

IDEA: use **TOMBSTONES**, i.e. replace deleted records with a marker indicating that a record once lived there:

⋮	⋮	
4	//////////	
5	ADAMS	
6	JONES	
7	######	← tombstone (CORRECT: will find 'SMITH')
8	SMITH	
⋮	⋮	

A search must continue when it finds a tombstone, but can stop whenever an empty slot is found. A search for 'SMITH' will continue when if finds the tombstone in position 7 of the above table.

Note: Only insert a **tombstone** when the next record is occupied or is a tombstone. If the next record is an empty slot, we may mark the deleted record as empty. Why ?

Insertions should be modified to work with **tombstones**: if either an empty slot or a tombstone is reached, place the new record there.

Effects of Deletions and Additions on Performance

The presence of too many tombstones increases search length.

Solutions to the problem of deteriorating average search lengths:

1. Deletion algorithm may try to move records that follow a tombstone backwards towards its home address.
2. Complete reorganization: re-hashing.
3. Use a different type of collision resolution technique.

Other Collision Resolution Techniques

1) Double Hashing

- The first hash function determines the home address.
- If the home address is occupied, apply a second hash function to get a number c (c relatively prime to N).
- c is added to the home address to produce an overflow addresses; if occupied, proceed by adding c to the overflow address, until an empty spot is found.

Example:

k (key)	ADAMS	JONES	MORRIS	SMITH
$h_1(k)$ (home address)	5	6	6	5
$h_2(k) = c$	2	3	4	3

Hashed file using double hashing:

0	
1	
2	
3	
4	
5	ADAMS
6	JONES
7	
8	SMITH
9	
10	MORRIS

0	XXXXX
1	XXXXX
2	XXXXX
3	XXXXX
4	XXXXX
5	XXXXX
6	XXXXX
7	XXXXX
8	XXXXX
9	XXXXX
10	XXXXX

Suppose the above table is full, and that a key $\overline{k}$ has $h_1(\overline{k}) = 6$ and $h_2(\overline{k}) = 3$.

Question: What would be the order in which the addresses would be probed when trying to insert $\overline{k}$?

Answer: 6, 9, 1, 4, 7, 10, 2, 5, 8, 0, 3.

2) Chained Progressive Overflow

- Similar to progressive overflow, except that synonyms are linked together with pointers.
- The objective is to reduce the search length for records within clusters.

Example X: Search lengths:

Key	Home	Progressive Overflow	Chained Progr. Overflow
ADAMS	20	1	1
BATES	21	1	1
COLES	20	3	2
DEAN	21	3	2
EVANS	24	1	1
FLINT	20	6	3
Average Search Length :		2.5	1.7

Progressive Overflow

	data
⋮	⋮
20	ADAMS
21	BATES
22	COLES
23	DEAN
24	EVANS
25	FLINT
⋮	⋮

Chained Progressive Overflow

	data	next
⋮	⋮	⋮
20	ADAMS	22
21	BATES	23
22	COLES	25
23	DEAN	-1
24	EVANS	-1
25	FLINT	-1
⋮	⋮	⋮

PROBLEM: Suppose that 'DEAN' home address is 22. Since 'COLES' is there, we couldn't have a link to 'DEAN' starting in its home address!

Solution:
Two-pass loading:
- First pass: only load records that fit into their home addresses.
- Second pass: load all overflow records.

Care should be taken when deletions are done.

key	**home address**
ADAMS	20
BATES	21
COLES	20
DEAN	22
EVANS	24
FLINT	20

table after first pass:

20	ADAMS	-1
21	BATES	-1
22	DEAN	-1
23		
24	EVANS	-1
25		

table after second pass:

20	ADAMS	23
21	BATES	-1
22	DEAN	-1
23	COLES	25
24	EVANS	-1
25	FLINT	-1

3) Chaining with a Separate Overflow Area

Move overflow records to a **Separate Overflow Area**.

A linked list of synonyms start at their home address in the Primary data area, continuing in the separate overflow area.

Example X, with separate overflow area:

primary data area

20	ADAMS	0
21	BATES	1
22		
23		
24	EVANS	-1
25		

overflow area

0	COLES	2
1	DEAN	-1
2	FLINT	-1
3		
	⋮	⋮

When the packing density is higher than 1 an overflow area is **required**.

4) Scatter Tables: Indexing Revisited

Similar to chaining with separate overflow, but the hashed file contains no records, but only pointers to data records. The scatter

Example X organized as scatter table:

index (hashed)

	⋮
20	0
21	1
22	
23	
24	4
	⋮

datafile (entry-sequenced, sorted, etc.)

	data	next
0	ADAMS	2
1	BATES	3
2	COLES	5
3	DEAN	-1
4	EVANS	-1
5	FLINT	-1

Note that the data file can be organized in many different ways: sorted file, entry sequenced file, etc.

Patterns of Record Access

Twenty percent of the students send 80 percent of the e-mails. *L. M.*

Using knowledge of pattern of record access to improve performance ...

Suppose you know that 80% of the searches occur in 20% of the items.

How to use this info to try to reduce search length in a hashed file ?

- Keep track of record access for a period of time (say 1 month).
- Sort the file in descending order of access.
- Re-hash using this order.

Records more frequently searched are more likely to be **at** or **close to** their home addresses.