Hashing

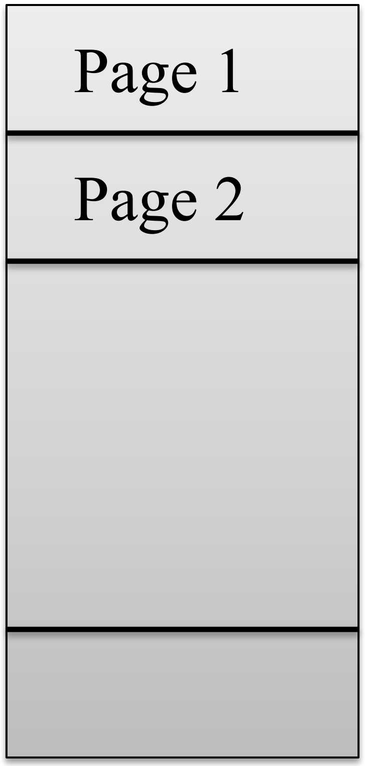
Davood Rafiei (Copyright 2001-2022)

- Given a search key, can we guess its location in the file?
- Goal:
  - Support equality searches in one disk access!

Method

- Build a hash table in file
- hash keys into addresses

  key → page



Hash Function

- Input: a field of a record; usually its key K (student id, name, …)
- Compute index function $H(K)$

$$H(K): K \rightarrow A$$

to find the address of K*.

$H(K)=A$ is the address of the record (or index entry) with key K

Hashing Function 1

Student id	Name	address
0234134	John	4
0349423	Mary	3
0428421	Jean	1
1324532	Sandy	2
2374734	Randy	4

Let some digits of the key, for example the last digit of the student id, represent the location.

More Hash Functions

- Folding
  - Replace the key by numeric code
    - ALBERT = 01 22 02 05 18 20
  - Fold and Add
    - 0122 + 0205 + 1820 = 2147
  - Take the modulo relative to the size of address space
    - 2147 mod 101 = 26
- Midsquare: Square key and take middle
  - $(453)^2 = 205209 \rightarrow 52$
- Radix Transformation
  - $(453)_{10} = (382)_{11} \rightarrow 382 \bmod 99 = 85$

Hashing Function 3

- concatenate the alphabetic positions of all letters, partition the result into equal parts, multiply each part by its position, fold and add, divide the result by the size of the address space (a prime number) and take the reminder.

Name		Address
John	10 15 08 14 → (1015*1 + 0814*2) mod 43 =	20
Mary	13 01 18 25 → (1301*1 + 1825*2) mod 43 =	6
Jean	10 05 01 14 → (1005*1 + 0114*2) mod 43 =	29
Sandy	19 01 14 04 25 → (1901*1 + 1404*2 + 0025*3) mod 43 =	11
Randy	18 01 14 04 25 → (1801*1 + 1404*2 + 0025*3) mod 43 =	40

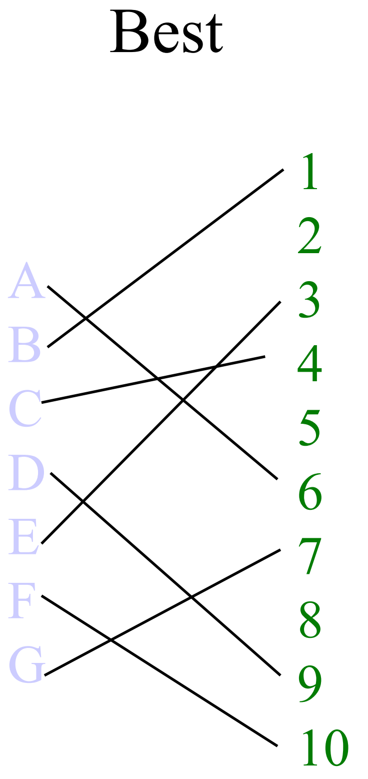
Hash Function Design Issues

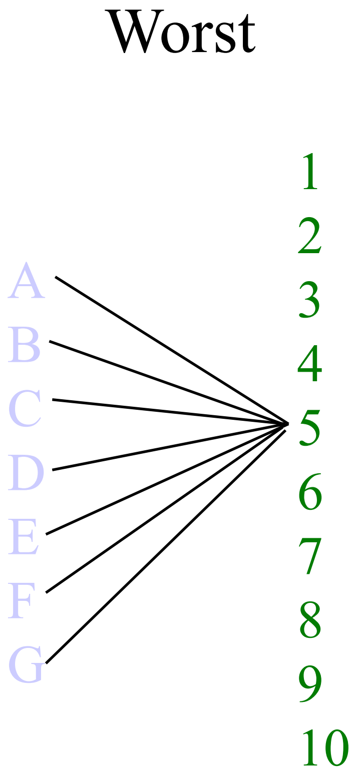
- Key space
  - The set of all possible values for keys
- Address space (*N*)
  - The set of all storage units
  - Physical location of file
- In general
  - Address space must accommodate all records in file
  - Address space is usually much smaller than key space

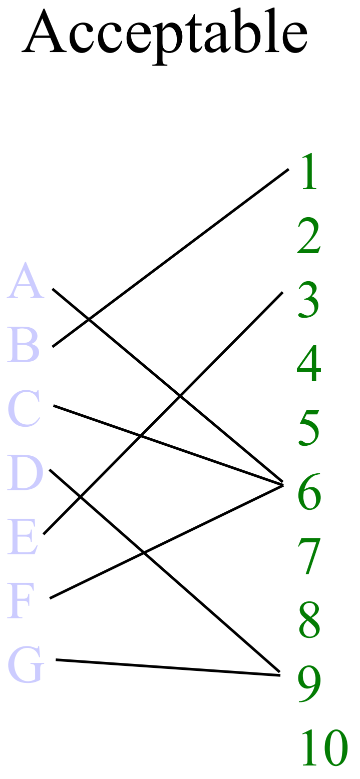
Features of Hashing

- Randomizing
  - Records are randomly spread over the whole storage space
- Collision
  - Two different keys may be hashed into the same address (synonyms)
  - To deal with it, two ways:
    - choose hashing functions that reduce collisions
    - rearrange the storage of records to reduce collisions

Good and Bad Functions







Choice of Hash Function

- Perfect hash function
  - One-to-one: No synonyms
    Key space = Address space
  - Not feasible for large and active files
- Desirable hashing function
  - Minimize collisions
  - Relatively smaller address space
- Tradeoff
  - The larger the address space, the easier it is to avoid collisions
  - The larger the address space, the worse the storage utilization becomes

A Hashing Function

1. Convert the key to a number (if it is not)
   key → **K**
2. Compute an address from the number
   address = **K** *mod* **M**

- **Suggestion:** Choose M to be a prime number (why?).

Collisions

- A key is mapped to an address that is full.
- Collision Resolution: Where to store the overflow key?
  - Static methods
    - **Linear probing**
    - Double hashing
    - Separate overflow
  - Dynamic methods
    - **Extendable hashing**
    - Linear hashing

Linear Probing

- For each key, generate a sequence of addresses A_0, A_1, A_2, …

A_0 = hash(key) mod M

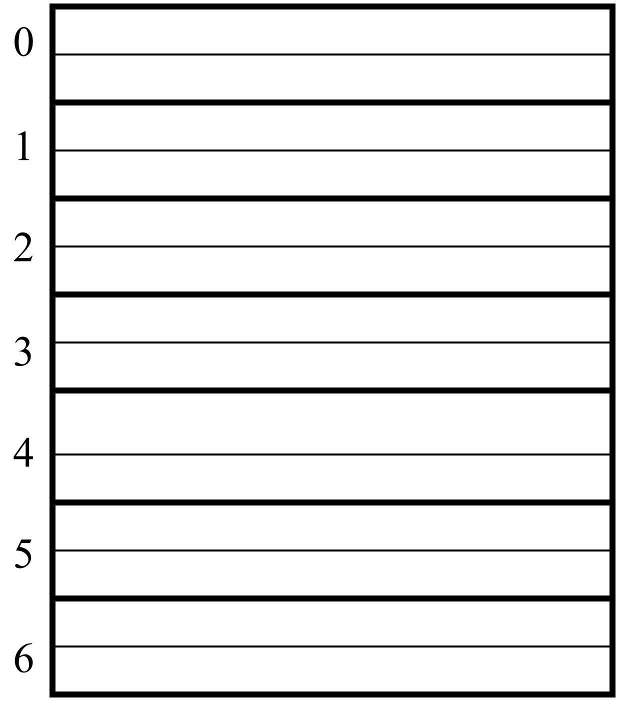
A_{i+1} = [A_i + step] mod M

M : file size (max # of addresses)

step: a constant

Example

Key	hash(key) = A_0	A_1	A_2	A_3	A_4
Mozart	1	2	3	4	5
Tchaikovsky	1	2	3	4	5
Ravel	3	4	5	6	0
Beethoven	5	6	0	1	2
Mendelssohn	5	6	0	1	2
Bach	3	4	5	6	0
Greig	3	4	5	6	0



$M = 7$
step = 1

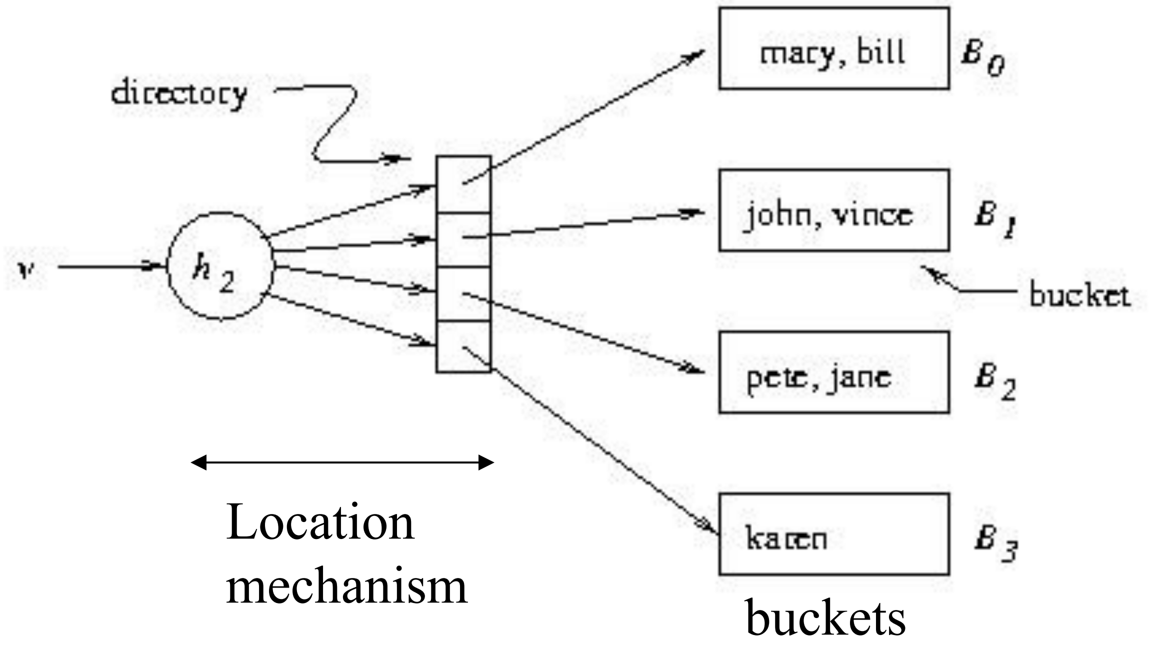
Linear Probing - Problems

- Performance degradation as more rows are added.
- Waste of space as more rows are deleted.
- These are problems for all static methods
- Solutions
  - Reorganization
  - Use a dynamic method

Extendable Hashing

- The address space is changed dynamically.
- The hash function is adjusted to accommodate the change.
- A common family of hash functions
  - $h_k(key) = h(key)$ mod 2^k (use the last k bits of $h(key)$)
  - At any given time a unique hash, h_k, is used

Extendable Hashing - Example



v	*h(v)*
pete	11010
mary	00000
jane	11110
bill	00000
john	01001
vince	10101
karen	10111

directory

00
01
10
11

The size of the directory corresponds to the currently active hash function h_k

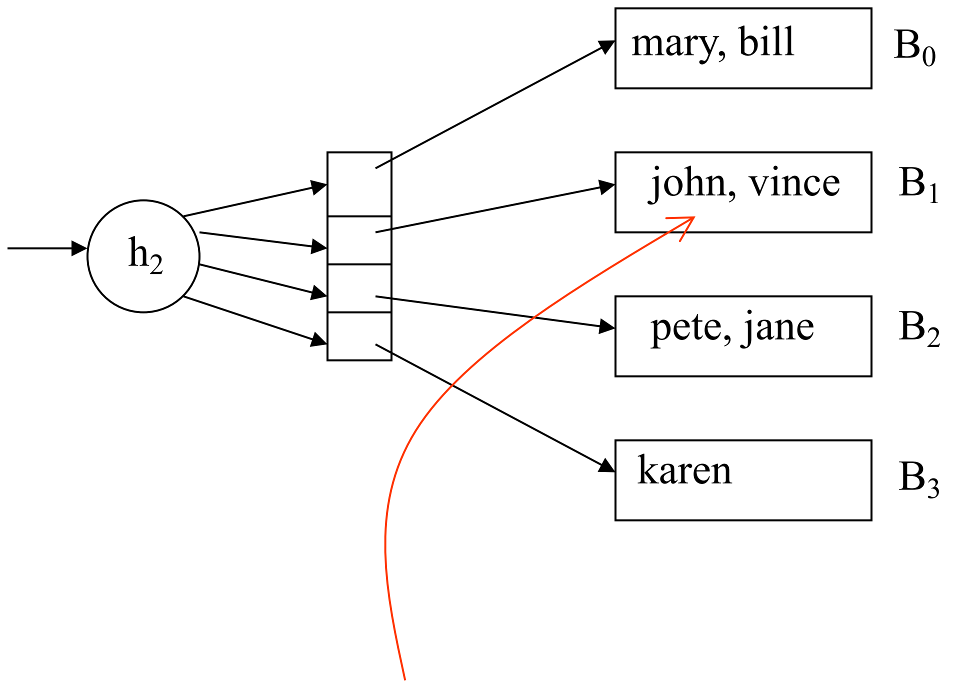
$h_k(key) = h(key) \bmod 2^k$

$k=2 \Rightarrow$ directory size = $2^2 = 4$

(use last $k=2$ bits of $h(key)$)

Example (con't)

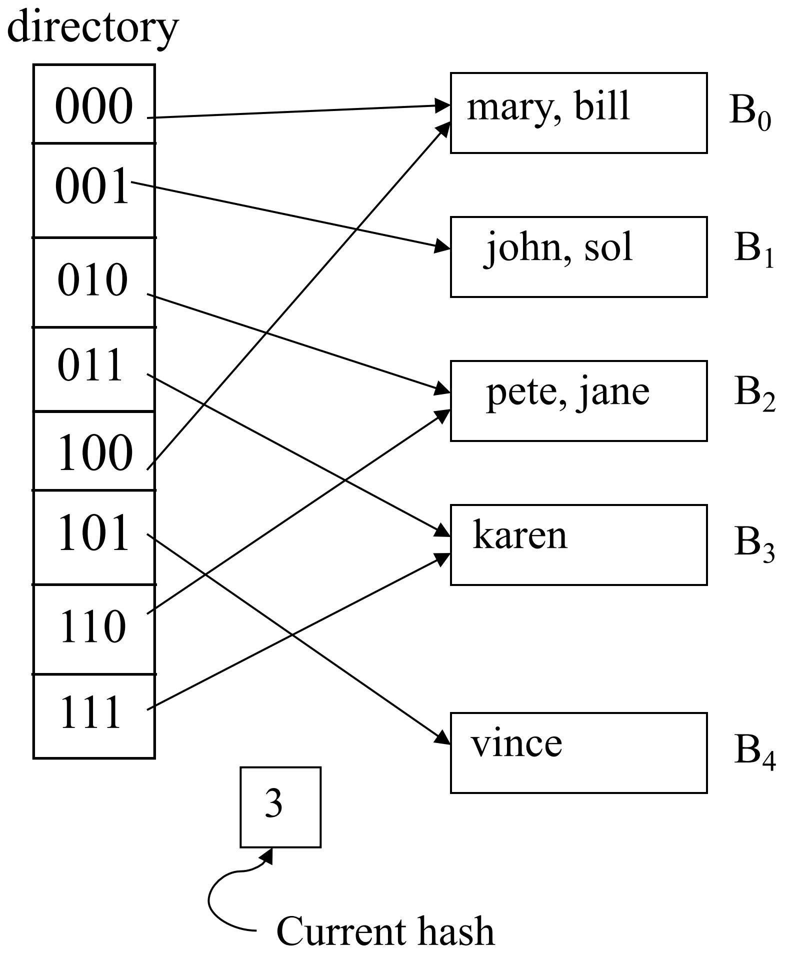
Next action: insert 'sol', where *h(sol) = 10001*.



v	*h(v)*
pete	11010
mary	00000
jane	11110
bill	00000
john	01001
vince	10101
karen	10111
sol	10001

sol, can't be stored here since the bucket is full

Example (con't)



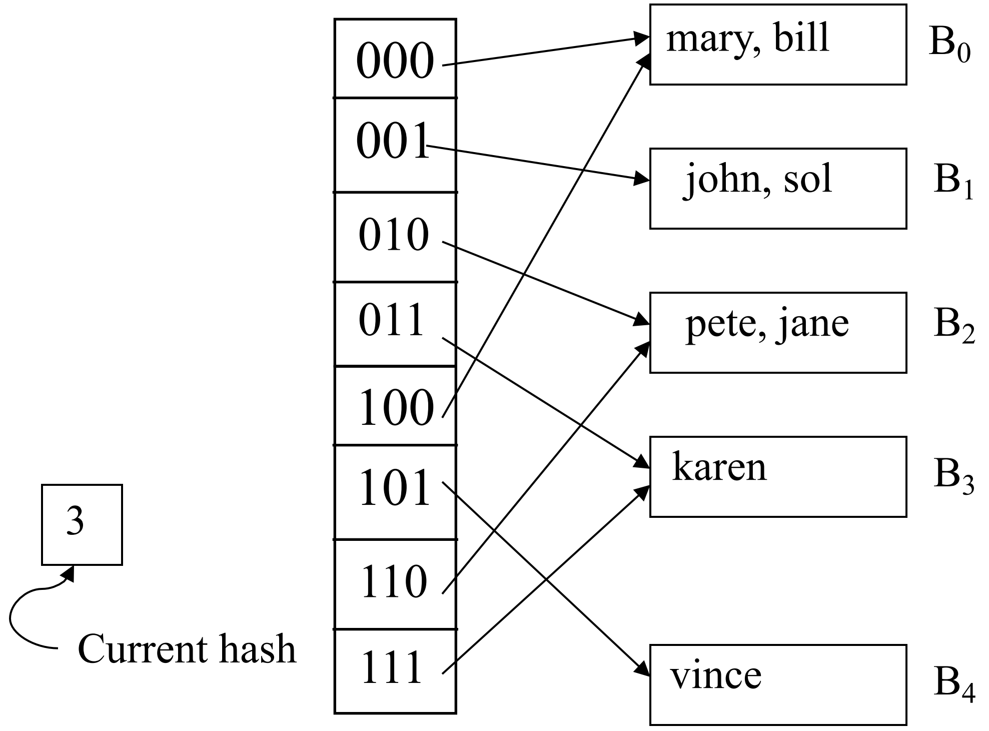
current_hash identifies current hash function.

Solution:

1. Split the overfilled bucket
2. Switch to h_3 (double the directory)
   $h_k(key) = h(key) \bmod 2^k$
   $k=3 \Rightarrow$ directory size $= 2^3 = 8$
   (use last $k=3$ bits of $h(key)$)
3. Update the pointers

v	*h(v)*
pete	11010
mary	00000
jane	11110
bill	00000
john	01001
vince	10101
karen	10111
sol	10001

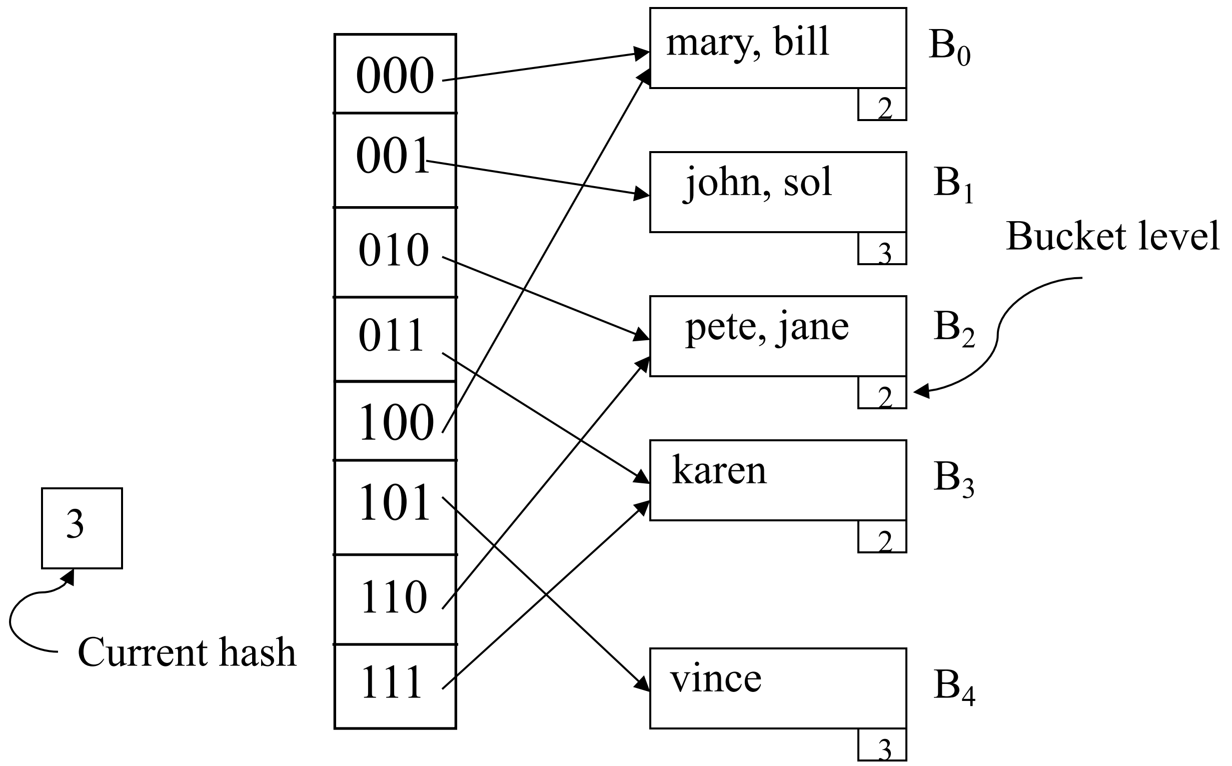
Example (con't)



- Next action: Insert judy, where *h(judy) = 00110*
- *B_2* overflows, but directory need not be extended

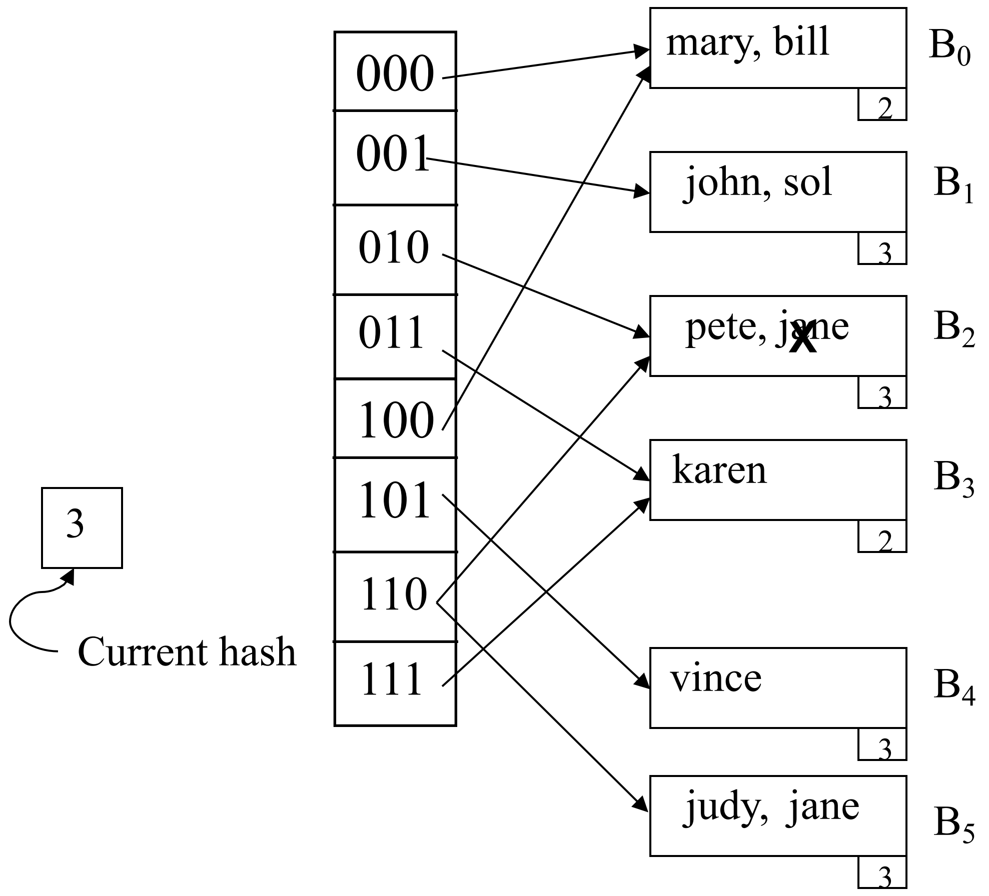
Need a mechanism for deciding whether the directory has to be doubled.

Example (con't)



Add a ***bucket level*** – if *current_hash > bucket_level[i],*
then do not enlarge directory

Example (con't)



v	*h(v)*
pete	11010
mary	00000
jane	11110
bill	00000
john	01001
vince	10101
karen	10111
sol	10001
judy	00110

v	*h(v)*
pete	11010
mary	00000
jane	11110
bill	00000
john	01001
vince	10101
karen	10111
sol	10001
judy	00110

Hash Indices - Summary

- Range search is not supported.
  - Since adjacent elements in range might hash to different buckets
- Partial key search is not supported.
  - Entire key must be provided
- But, an equality search on average takes only 1 disk access

Indexing in Sqlite

- Sqlite supports B+ tree index
- Create an index on author

```
CREATE INDEX authidx ON book (author);
```

```
CREATE TABLE book (
callno          char(10),
author          char(20),
title           char(30),
year            char(4),
PRIMARY KEY (callno));
```

- An unclustered (dense) index on author is created
- Can also make it unique

```
CREATE UNIQUE INDEX authidx ON book (author);
```

Indexing in Sqlite

- By default, only *unique* and *primary key* columns are indexed
- Can make it clustered on primary key

```
CREATE TABLE book (
callno          char(10),
author          char(20),
title           char(30),
year            char(4),
PRIMARY KEY (callno)) WITHOUT ROWID;
```

- Primary key is required for *without rowid* tables

Indexing in Oracle
(un-clustered index)

- Create an un-clustered index on author:

```
CREATE TABLE book (
callno          char(10),
author          char(20),
title           char(30),
year            char(4),
PRIMARY KEY (callno)
);

CREATE INDEX authidx ON book (author);
```

- Result: an un-clustered dense index on author.

Indexing in Oracle
(clustered index on primary key)

- Create a clustered index on callno:

```
CREATE TABLE book (
callno          char(10),
author          char(20),
title           char(30),
year            char(4),
PRIMARY KEY (callno)
)
ORGANIZATION INDEX;
```

- This syntax allows a clustered index on the primary key of the table only.

Indexing in Oracle
(clustered index on non-primary key columns)

- Create a clustered index on author:

```
CREATE TABLE book (
callno          char(10),
author          char(20),
title           char(30),
year            char(4),
PRIMARY KEY (callno)
)
cluster authcl(author);

CREATE INDEX authidx on cluster authcl;
```

- An Oracle cluster may contain rows from more than one table.

Indexing in DB2

- Create un-clustered indexes on callno and author:

```
CREATE INDEX callno_idx on book (callno)
CREATE INDEX auth_idx on book (author)
```

- Can make (only) one index clustered:

```
CREATE INDEX auth_idx on book (author) cluster
```

Data must be (preferably) sorted on clustering column(s) in the OS file.

Choosing an Index

```
Ex 1  SELECT  E. Id
      FROM    Employee E
      WHERE   E.Salary < :upper  AND  E.Salary > :lower
```

- a range search on Salary.
- Suppose the primary key is employee id; it is likely that there is a main, clustered index on that attribute that is of no use for this query.
- Choose a secondary, B^+ tree index with search key Salary

Choosing an Index

Ex 2
```
SELECT  T.StudId
FROM    Transcript T
WHERE   T.Grade = :grade
```

- an equality search on Grade.
- Suppose the primary key is (StudId, Semester, CrsCode); it is likely that there is a main, clustered index on these attributes that is of no use for this query.
- Choose a secondary, B+ tree or hash index with search key Grade

Choosing an Index

Ex 3

```
SELECT   T.CrsCode, T.Grade
FROM  Transcript T
WHERE  T.StudId = :id  AND  T.Semester = ‘F2000’
```

- Equality search on StudId and Semester.
- If the primary key is (StudId, Semester, CrsCode) it is likely that there is a main, clustered index on this *sequence* of attributes.
- If the main index is a B$^+$ tree it can be used for this search.
- If the main index is a hash it cannot be used for this search. Choose B$^+$ tree or hash with search key StudId or (StudId, Semester)

Choosing an Index

Ex 3 (con't)

```
SELECT   T.CrsCode, T.Grade
FROM  Transcript T
WHERE  T.StudId = :id  AND  T.Semester = 'F2000'
```

- Suppose Transcript has primary key (CrsCode, StudId, Semester). Can this index be useful (independent of being hash or B^+ tree)?