Overview

Goal:

Support equality searches in one disk access!

Method

1. Build a hash table in file
2. Hash keys into addresses

$$\text{key} \rightarrow \text{page}$$

Hash Functions

Description

Compute index function $H(K)$ to find the address of K^*

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

- K : a field of a record, usually its key
- A : the address of the record (or index entry)

Examples

Example 1

Let it be some digits of a key (e.g., the last digit of a student id)

Example 2

Folding

Example

1. Replace the key by numeric code
   - ALBERT = 01 22 02 05 18 20
2. Fold and Add
   - 0122 + 0205 +1820 = 2147
3. Take the modulo relative to the size of address space
   - 2147 mod 101 = 26

Example 3

Midsquare

Square key and take middle

Example

$$453^2 = 205209 \Rightarrow 52$$

Example 4

Radix Transformation

$$453_{10} = 382_{11} \rightarrow 382 \bmod 99 = 85$$

Example 5

Concatenate the alphabetic positions of all letters, partition the result into equal parts, multiple each part by its position, fold and add, divide the result by the size of the address space (a prime number) and take the remainder

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

Generally

- 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 while storage space

Collision

- Two different keys may be hashed into the same address (**synonyms**)
- Deal with it
  - Choose hashing functions that reduce collisions
  - Rearrange the storage of records to reduce collisions

Choice of Hash Function

Perfect Hash Function

ONE-to-ONE
- No Synonyms
- The size of the key space is equal to size of the address space

Desirable Hashing Function

- Minimize collisions
- Relatively smaller address space

Tradeoffs

- The larger the address space, the easier it is to avoid collision
- The larger the address space, the worse the storage utilization becomes

A Hashing Function

Method

1. Convert the key **K** to a number (if it is not)
2. Compute and address from the number

$$\text{address} = K \bmod M$$

- Suggestion: Choose M to be a prime number

Collisions

A key is mapped to an address that is full

Collision Resolution

Static Methods

1. **Linear Probing
2. Double Hashing
3. Separate overflow

Dynamic Methods

1. **Extendable hashing**
2. Linear hashing

Linear Probing

For each key, generate a sequence of addresses $A_0, A_1, A_2, \ldots$

$$A_0 = H(K) \bmod M$$

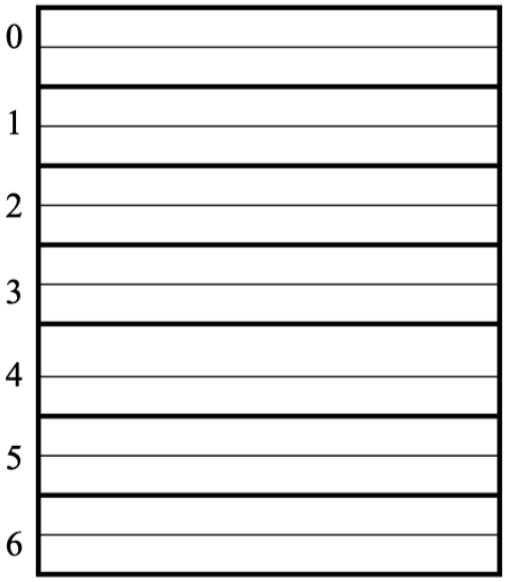
$$A_{i+1} = [A_i + \text{step}] \bmod M$$

- K : is the entry key
- H : is the hashing function
- A : is the address
- $step$: a constant

Example

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

Problems

1. performance degradation as more rows are added
2. waste of space as more rows are deleted
3. these are problems for all static methods

Solution?

1. Reorganization
2. Use Dynamic Method

Extendable Hashing

1. The address space is changed dynamically
2. The hash function is adjusted to accommodate the change

A common family of hash functions

- Use the last k bits of $h(\text{key})$

$$h_k(\text{key}) = h(\text{key}) \; mod \; 2^k$$

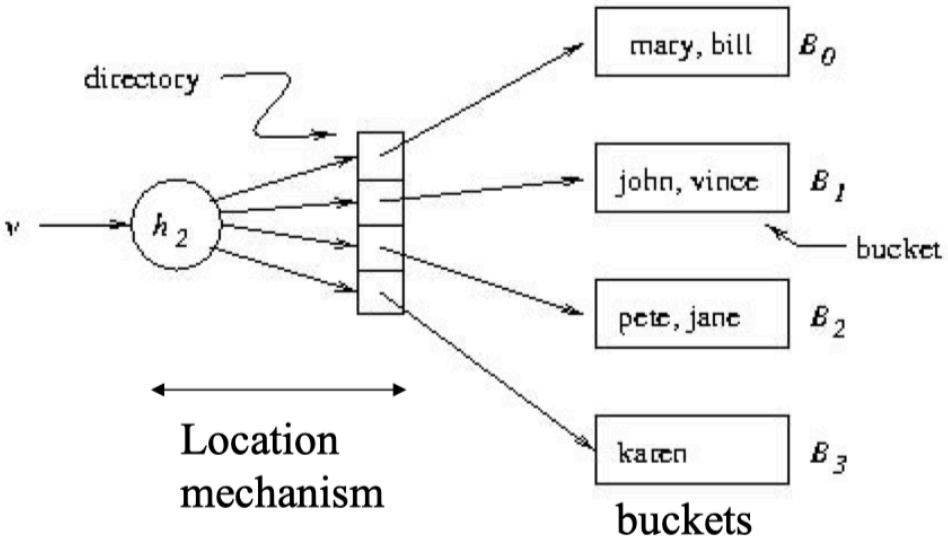
- At any given time a unique hash h_k is used

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

$$\text{directory size} = 2^k$$

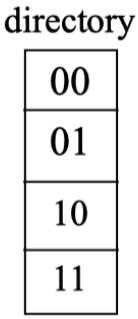
- *current_hash* identifies current hash function
- We need a mechanism for deciding whether the directory has to be doubled
  - if `curent_hash > bucket_level[i]`, then do not enlarge directory

Example



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

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



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

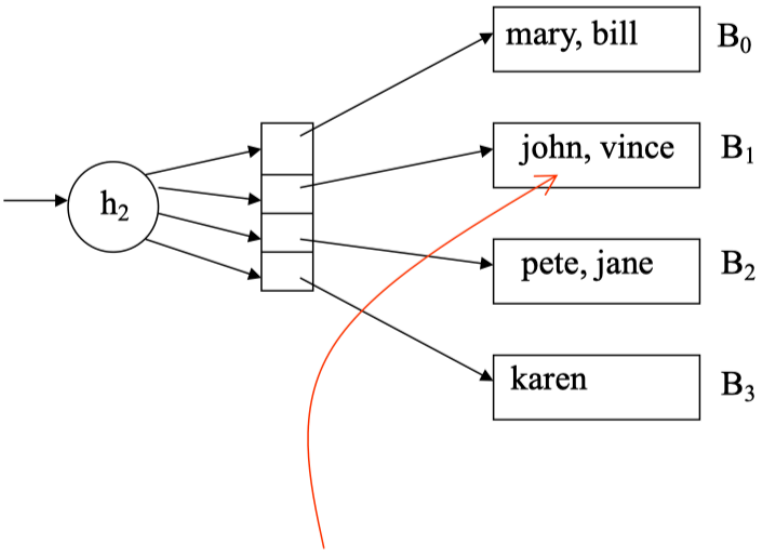
$k=2 \Rightarrow$ directory size $= 2^2 = 4$

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

18

Next action: insert 'sol'

then $h(sol) = 10001$

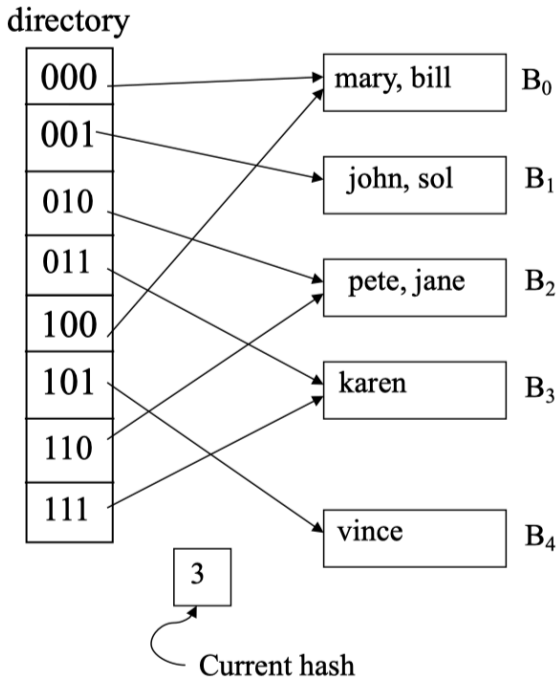


v	h(v)
pete	11010
mary	00000
jane	11110
bill	00000
john	01001
vince	10101
karen	10111
sol	10001

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

Solution

1. Split the overfilled bucket
2. Switch to h_3 (doubles the directory) - now we use the last 3 bits of $h(key)$
3. Update the pointers



current_hash identifies current hash function.

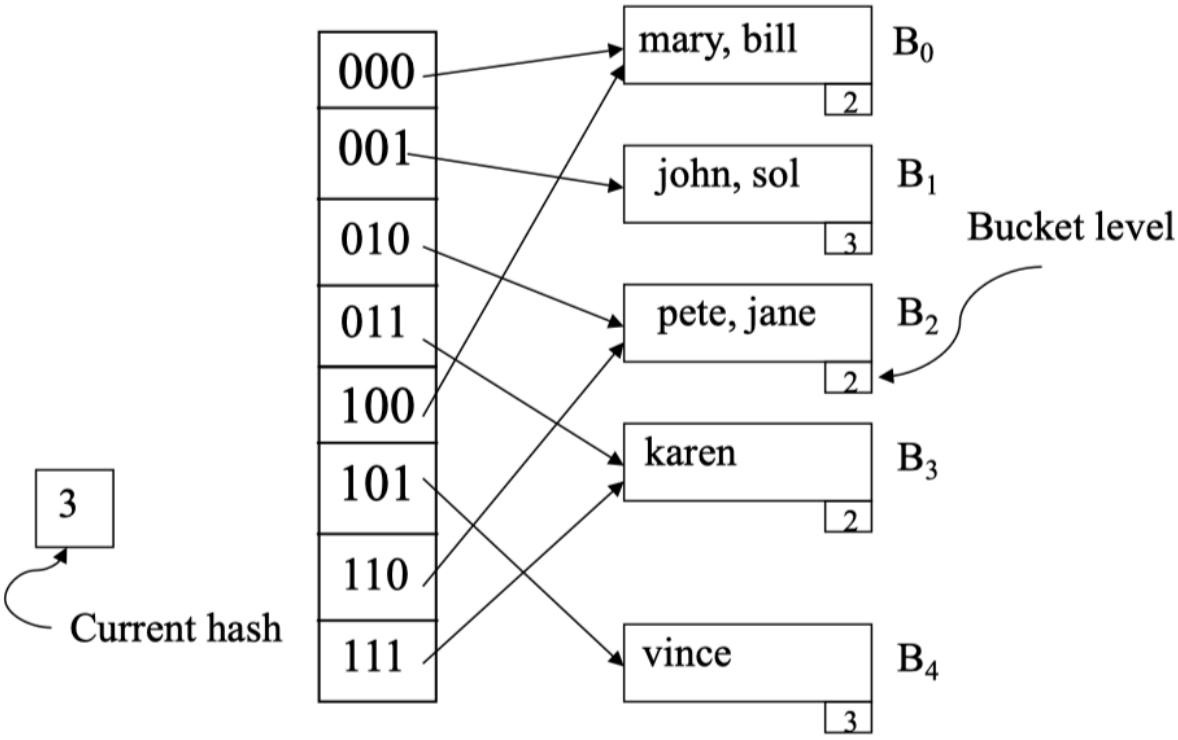
Solution:

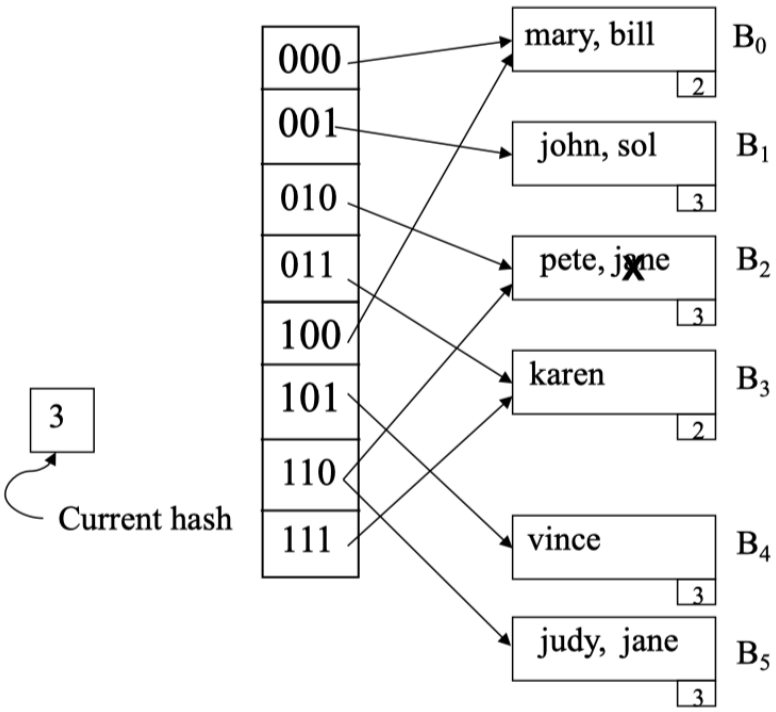
1. Split the overfilled bucket
2. Switch to h_3 (double the directory)
   $h_k(key) = h(key) \mod 2^k$
   $k=3 \Rightarrow$ directory size $= 2^3 = 8$
   (use last $k=3$ bits of $h(key)$)
3. Update the pointers

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

Next action: Insert Judy

- $h(judy) = 00110
- B_2 overflows but directory need extended





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

Summary - Hash Indices

- Range search is not supported
  - Since adjacent elements 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**