

ICS 421 Spring 2010

Indexing (2)

Asst. Prof. Lipyeow Lim
Information & Computer Science Department
University of Hawaii at Manoa

Hash Indexes

- *As for any index, 3 alternatives for data entries k^* :*
 - Data record with key value k
 - $\langle k, \text{rid of data record with search key value } k \rangle$
 - $\langle k, \text{list of rids of data records with search key } k \rangle$
 - Choice orthogonal to the *indexing technique*
- Hash-based indexes are best for *equality selections*. **Cannot** support range searches.
- Static and dynamic hashing techniques exist; trade-offs similar to ISAM vs. B+ trees.

The Hashing Idea (i)



DOW Number	DOW String
1	Monday
2	Tuesday
3	Wednesday
4	Thursday
5	Friday
6	Saturday
7	Sunday

How do we get the day of the week (DOW) string from the DOW number ?

```
String DOWstring[8] = { "invalid",  
                        "Monday",  
                        "Tuesday",  
                        "Wednesday",  
                        "Thursday",  
                        "Friday",  
                        "Saturday",  
                        "Sunday" };
```

```
Print ( "Day 4 of the week is " + DOWstring[4] );
```

- What if we want to use an array of 7 slots ?
- Essential idea: get an array index/address directly from the key field

The Hashing Idea (ii)

data

Key

DOW Number	DOW String
1	Monday
2	Tuesday
3	Wednesday
4	Thursday
5	Friday
6	Saturday
7	Sunday

How do we get the numeric day of the week (DOW) from the DOW string ?

```
Int hashfn(string day) {  
    foreach i do sum += day[i]  
    return sum % 7;    }
```

```
String DOWnum[7];
```

```
DOWnum[hashfn("Monday")] = 1;  
DOWnum[hashfn("Tuesday")] = 2;
```

```
...
```

```
Print ( "Monday is day " +  
DOWnum[hashfn("Monday")] + " of the week );
```

- What do we do if two strings map to the same hash value ?

Hash Indexes in Databases

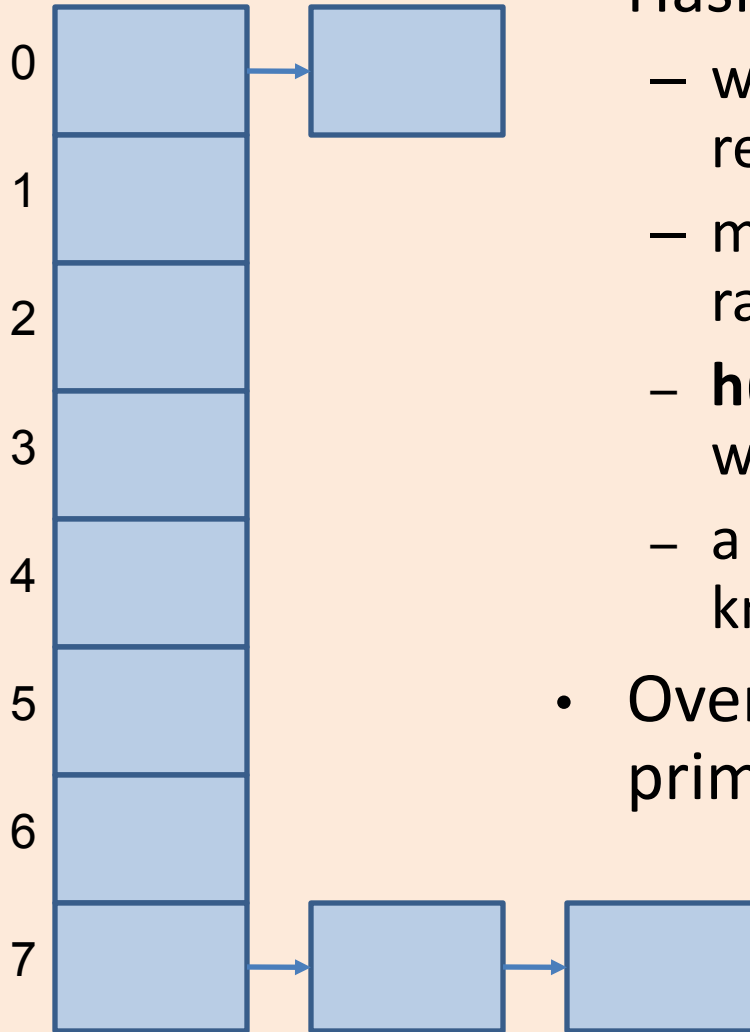
Array
of Pages



- Conceptually an array of pages or buckets
- $h(k) \bmod M = \text{bucket ID for key } k$
- M is the number of buckets in array
- *Data entries k^** :
 - Data record with key value k
 - $\langle k, \text{rid of data record with search key value } k \rangle$
 - $\langle k, \text{list of rids of data records with search key } k \rangle$
 - Choice orthogonal to the *indexing technique*
- Hash-based indexes are best for *equality selections*. **Cannot** support range searches.

Static Hashing

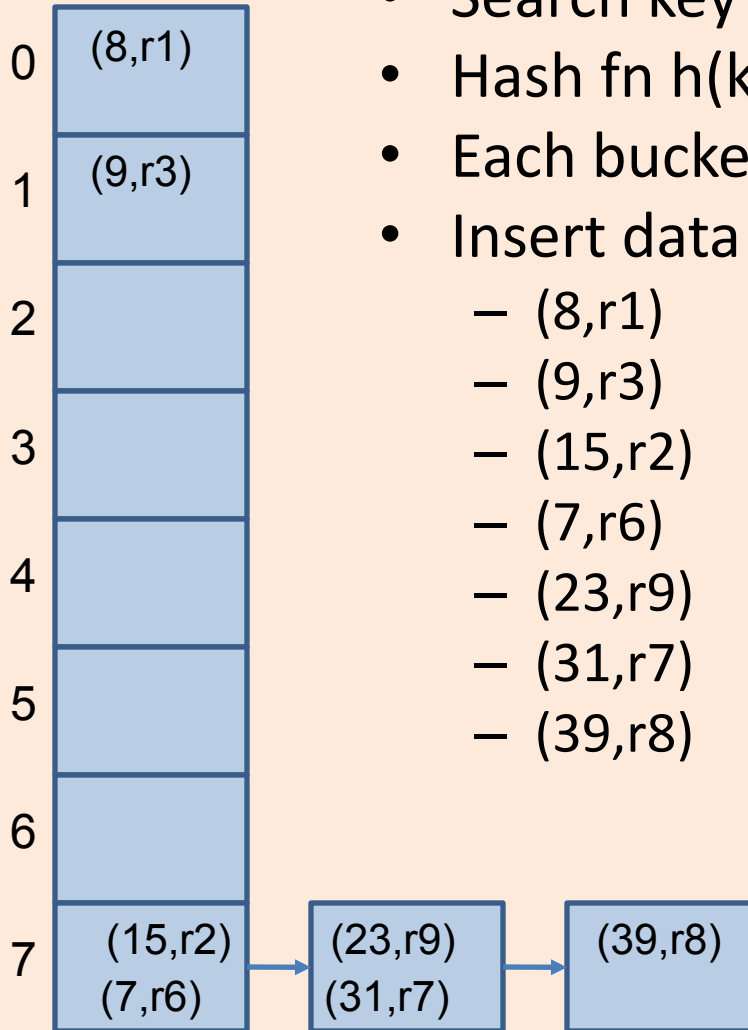
Array
of pages
/buckets



- Hash fn
 - works on *search key* field of record *r*.
 - must distribute values over range $0 \dots M-1$.
 - $h(\text{key}) = (a * \text{key} + b)$ usually works well.
 - *a* and *b* are constants; lots known about how to tune **h**.
- Overflow buckets used when primary buckets are full

Example : Static Hashing

Array
of pages
/buckets



- Search key is Sailors.age
- Hash fn $h(k) = k \bmod 8$
- Each bucket/page can hold 2 entries
- Insert data entries

- (8,r1)
- (9,r3)
- (15,r2)
- (7,r6)
- (23,r9)
- (31,r7)
- (39,r8)

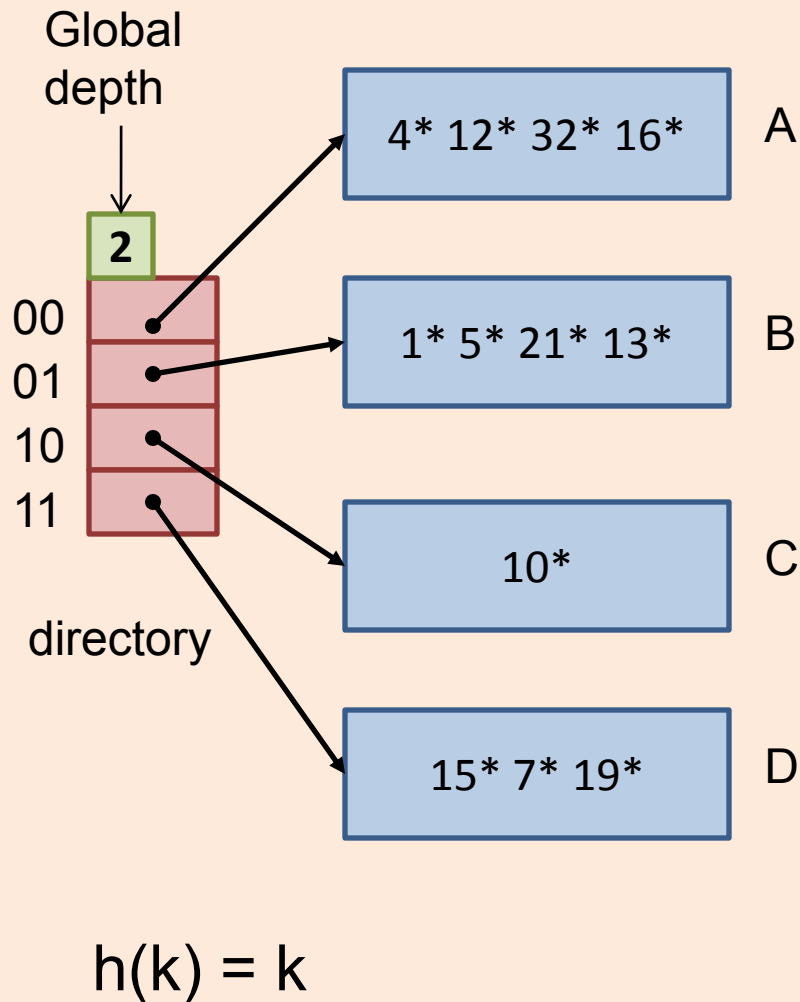
How many page IOs do i need to find RIDs of sailors aged 8?

How many page IOs do i need to find RIDs of sailors aged 31?

Extendible Hashing

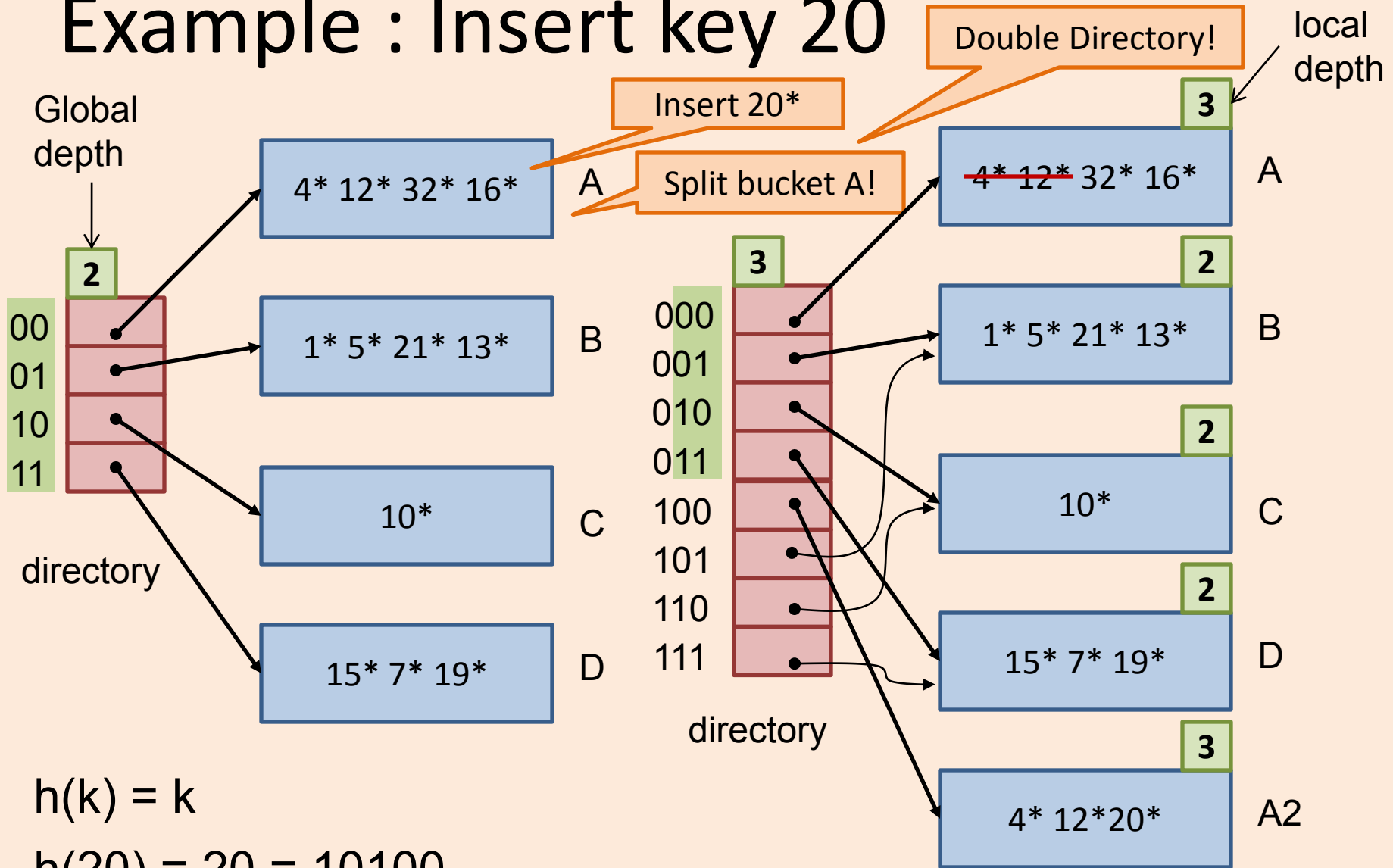
- Situation: Bucket (primary page) becomes full. Why not re-organize file by *doubling* # of buckets?
 - Reading and writing all pages is expensive!
 - Idea: Use directory of pointers to buckets, double # of buckets by *doubling the directory*, splitting just the bucket that overflowed!
 - Directory much smaller than file, so doubling it is much cheaper. Only one page of data entries is split. *No overflow page!*
 - Trick lies in how hash function is adjusted!

Example : Extendible Hashing



- Directory is array of size 4.
- Each bucket holds 4 entries.
- To find bucket for r , take last '*global depth*' # bits of $h(r)$; we denote r by $h(r)$.
 - If $h(r) = 5 = \text{binary } 101$, it is in bucket pointed to by 01.
- **Insert**: If bucket is full, *split* it (allocate new page, redistribute).
- If necessary, double the directory.

Example : Insert key 20



$$h(k) = k$$

$$h(20) = 20 = 10100$$

Points to Note

- 20 = binary 10100. Last **2** bits (00) tell us r belongs in A or A2. Last **3** bits needed to tell which.
 - *Global depth of directory*: Max # of bits needed to tell which bucket an entry belongs to.
 - *Local depth of a bucket*: # of bits used to determine if an entry belongs to this bucket.
- When does bucket split cause directory doubling?
 - Before insert, *local depth* of bucket = *global depth*. Insert causes *local depth* to become $>$ *global depth*; directory is doubled by *copying it over* and 'fixing' pointer to split image page. (Use of least significant bits enables efficient doubling via copying of directory!)
- If directory fits in memory, equality search answered with one disk access; else two.

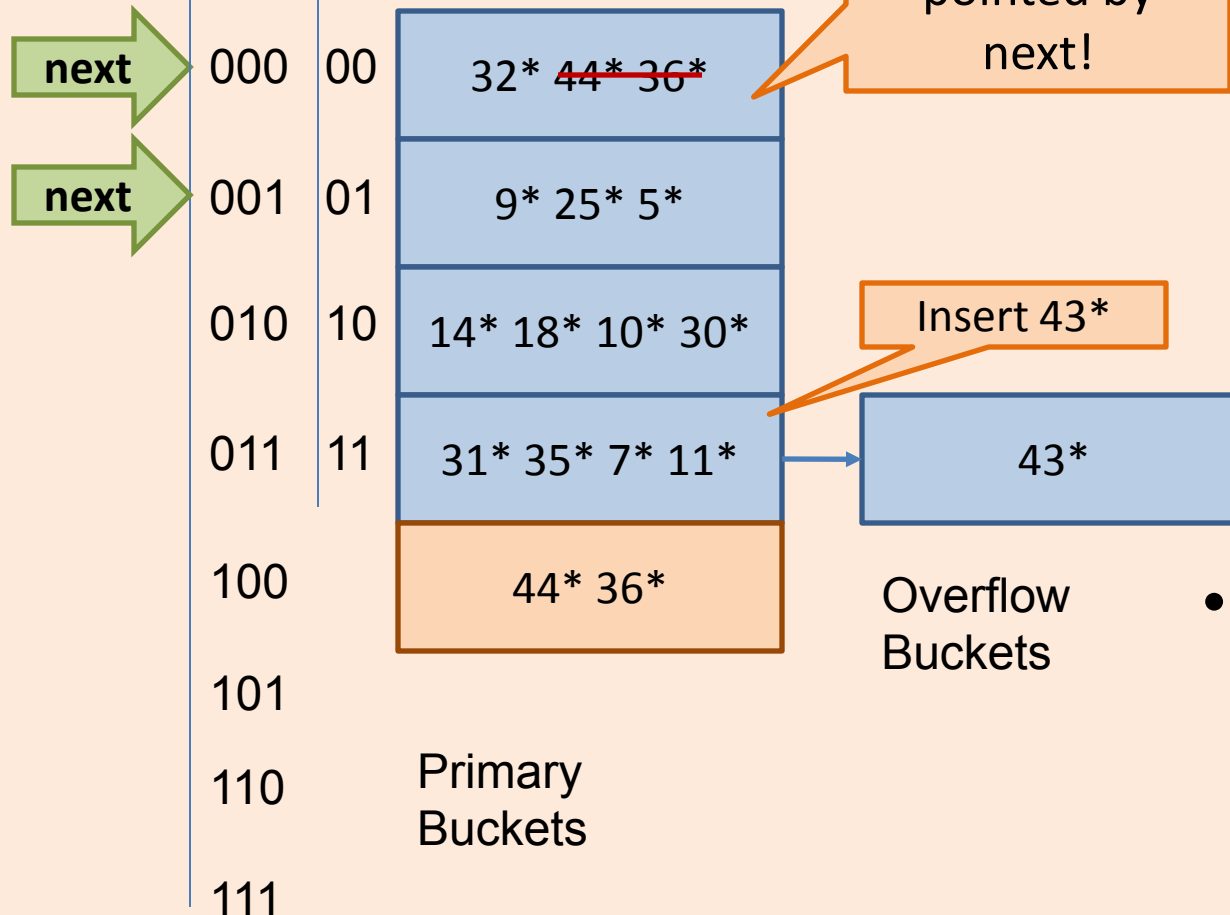
Linear Hashing

- This is another dynamic hashing scheme, an alternative to Extendible Hashing.
- LH handles the problem of long overflow chains without using a directory, and handles duplicates.
- Idea: Use a family of hash functions h_0, h_1, h_2, \dots
 - $h_i(\text{key}) = h(\text{key}) \bmod(2^i N)$; N = initial # buckets
 - h is some hash function (range is *not* 0 to $N-1$)
 - If $N = 2^{d_0}$, for some d_0 , h_i consists of applying h and looking at the last d_i bits, where $d_i = d_0 + i$.
 - h_{i+1} doubles the range of h_i (similar to directory doubling)

Example: Linear Hashing

$$h_1(k) = k \bmod 8$$

$$h_0(k) = k \bmod 4$$



- **Insert:** Find bucket by applying $h_{Level} / h_{Level+1}$:
 - If bucket to insert into is full:
 - Add overflow page and insert data entry.
 - (Maybe) Split Next bucket and increment Next.
- Since buckets are split round-robin, long overflow chains don't develop!

Summary

- Hash-based indexes: best for equality searches, cannot support range searches.
- Static Hashing can lead to long overflow chains.
- Extendible Hashing avoids overflow pages by splitting a full bucket when a new data entry is to be added to it. (*Duplicates may require overflow pages.*)
 - Directory to keep track of buckets, doubles periodically.
 - Can get large with skewed data; additional I/O if this does not fit in main memory.

Summary (Cont.)

- Linear Hashing avoids directory by splitting buckets round-robin, and using overflow pages.
 - Overflow pages not likely to be long.
 - Duplicates handled easily.
 - Space utilization could be lower than Extendible Hashing, since splits not concentrated on `dense' data areas.
 - Can tune criterion for triggering splits to trade-off slightly longer chains for better space utilization.
- For hash-based indexes, a *skewed* data distribution is one in which the *hash values* of data entries are not uniformly distributed!