# Week3_Course_part4

# Principles of Data Layout and Index

## Index Structure Review

Database System - Nankai

# Principles of Data Layout and Index

How to lay out data on disk?

上周课

关系模式与关系

属性(Attribute) → 字段(Field)

元组(tuple) → 记录(Record)

记录集合 → 存储块（Block）

块的集合→ 存储关系(Relation)

How to find a record quickly?

本周课

磁盘的结构与访问特征

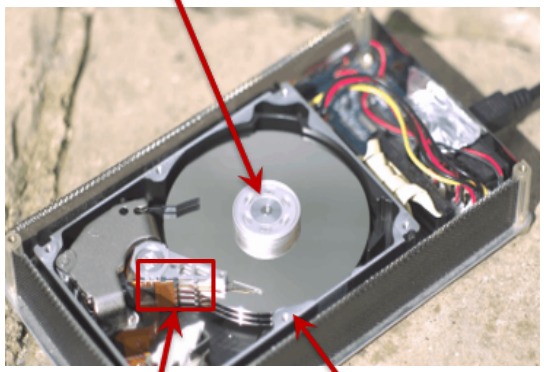
Conventional indexes

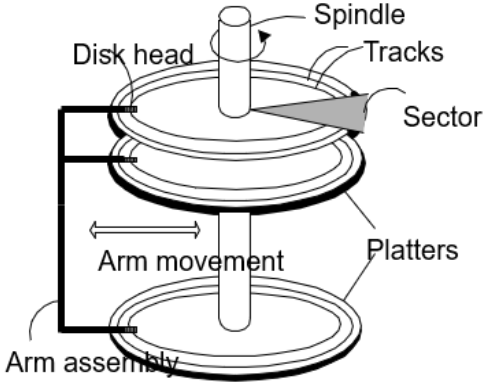
B+Trees

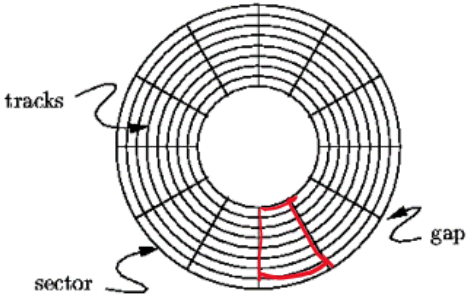
Hashing schemes

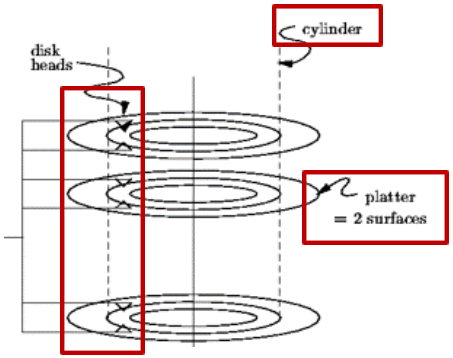
Database System - Nankai

# 磁盘结构




- Platter (2 surfaces)
- Head
- Cylinder
- Track
- Sector (physical)
- Block (logical)
- Gap

Database System - Nankai

# Physical vs. Logical I/O

- Sector: indivisible unit of disk I/O
- Blocks
  - logical (DBMS/OS) units of disk storage
  - transferred btw disk and main memory buffers
  - typically 4… 56 KB, consisting of one or more sectors
- Disk Access Time

I want block X →  → block X in memory

Time = Seek Time +
Rotational Delay +
Transfer Time +
Other

Database System - Nankai

# 磁盘访问特性

- Seek time
  - time to move heads to proper cylinder. (一般7.5～14ms)
- Rotational latency
  - time for desired block to come under the head.
    - 7200转/每分钟的硬盘，旋转一周时间为8.33毫秒，平均4.17毫秒
- Transfer time
  - time during which block passes under head.
- Miscellaneous
  - including time for disk controller to process data, contention for controller, bus, etc. Neglect this.

# 磁盘访问效率举例

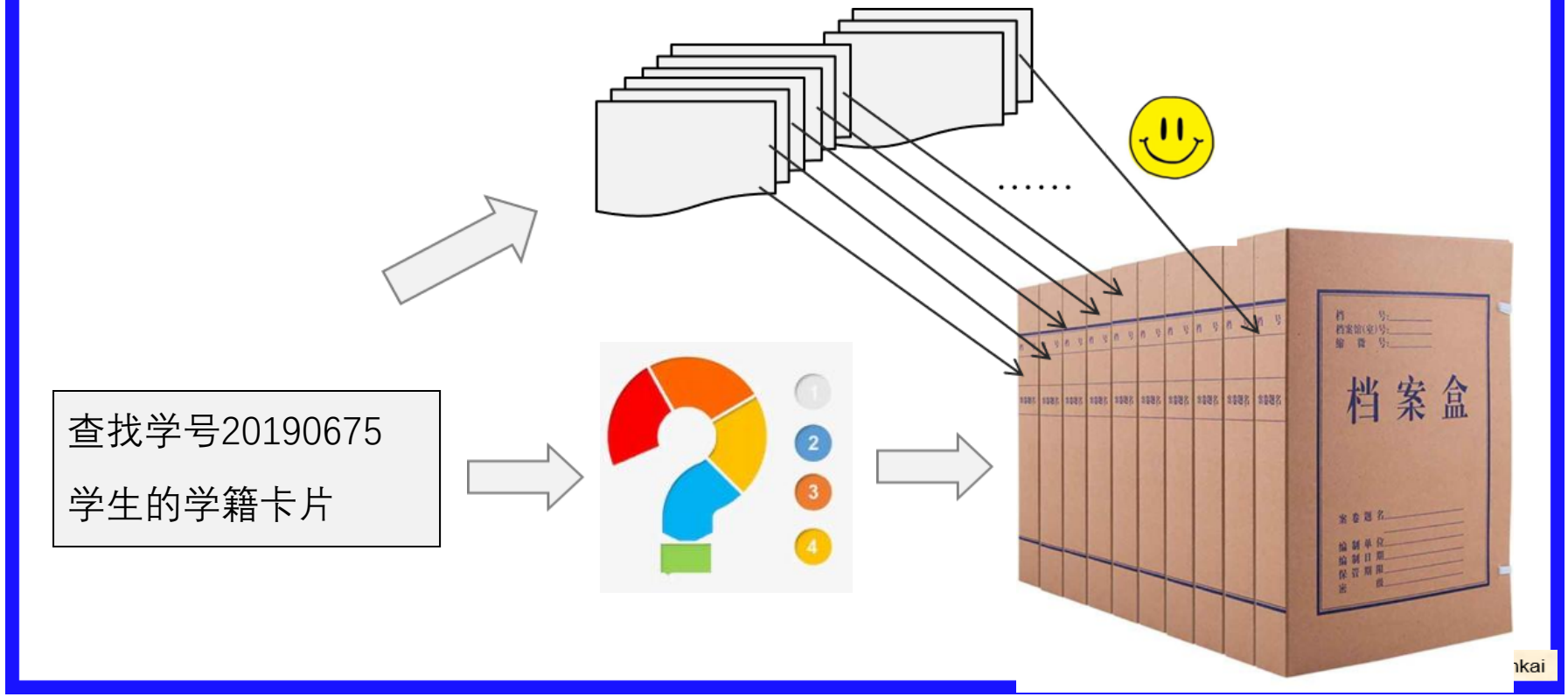
- 全国有14亿人口，需要构建数据库存储人口的基本信息，每位公民要存储身份证号、姓名、家庭住址、联系方式等信息；
- 如果磁盘块是4K(4096字节,8个扇区)，一个块可以存储100个人的记录，则14亿人口需要存储到1400万个数据块中；
- 如果在此数据库中需要根据身份证号查找人员的姓名及其他信息，就必须依次将块读入内存，逐个匹配身份证号，命中记录的平均读取块数大概有700万个。对7200转的磁盘来说，平均随机访问一个块的时间是14ms左右，读取700万个数据块要花费27个小时，即使数据按柱面存储也要花费近10个小时。

# How to find a record quickly, given a key

Next

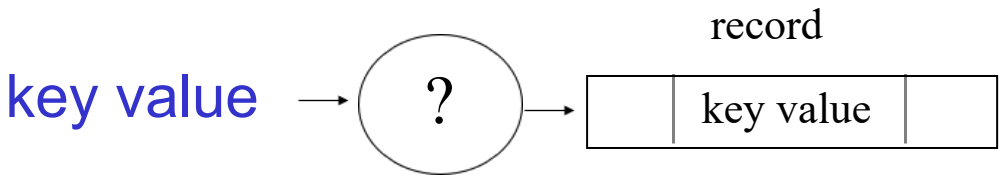
- 磁盘的结构与访问特征
- Conventional indexes
- B+Trees
- Hashing schemes

# 索引的直观描述



# Index Structures

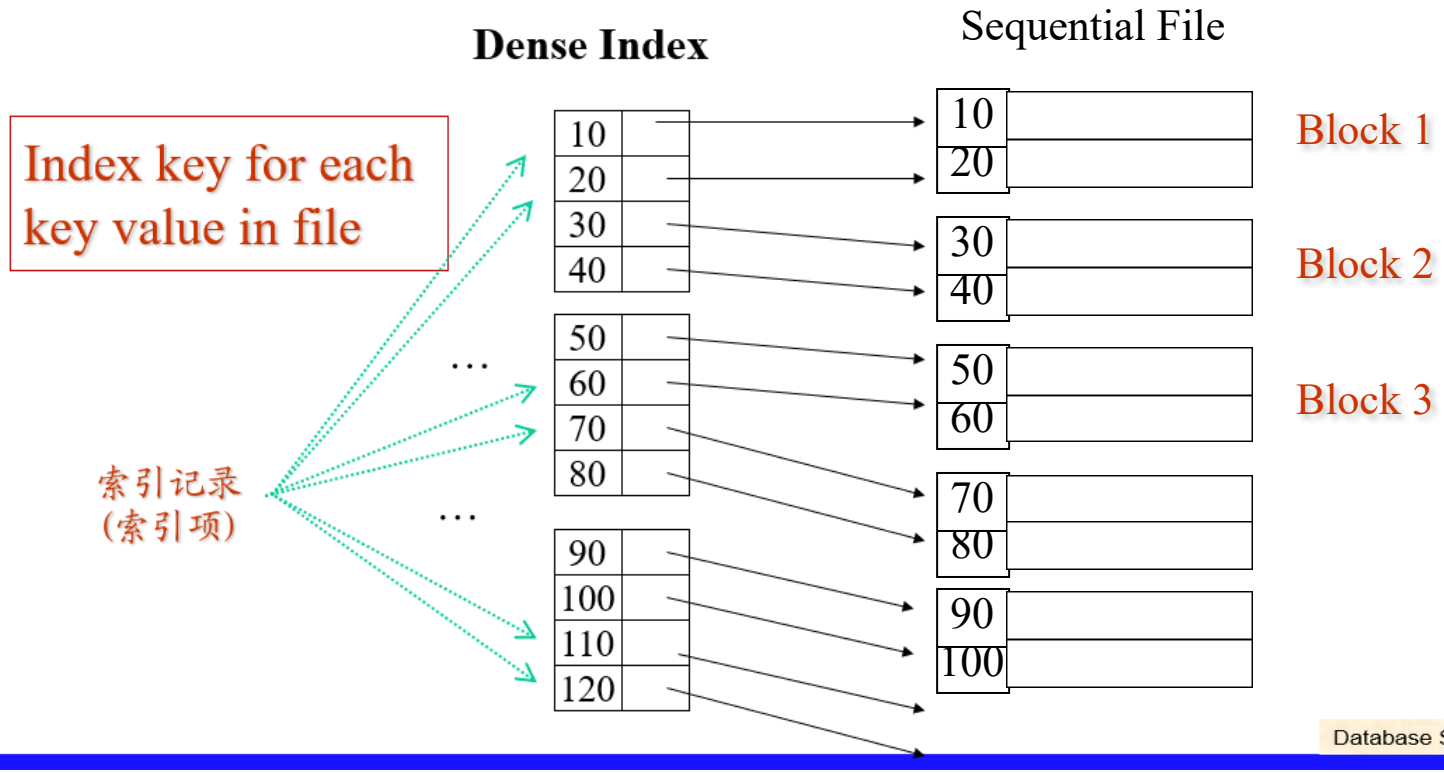
- Data structure for locating records with given search key efficiently



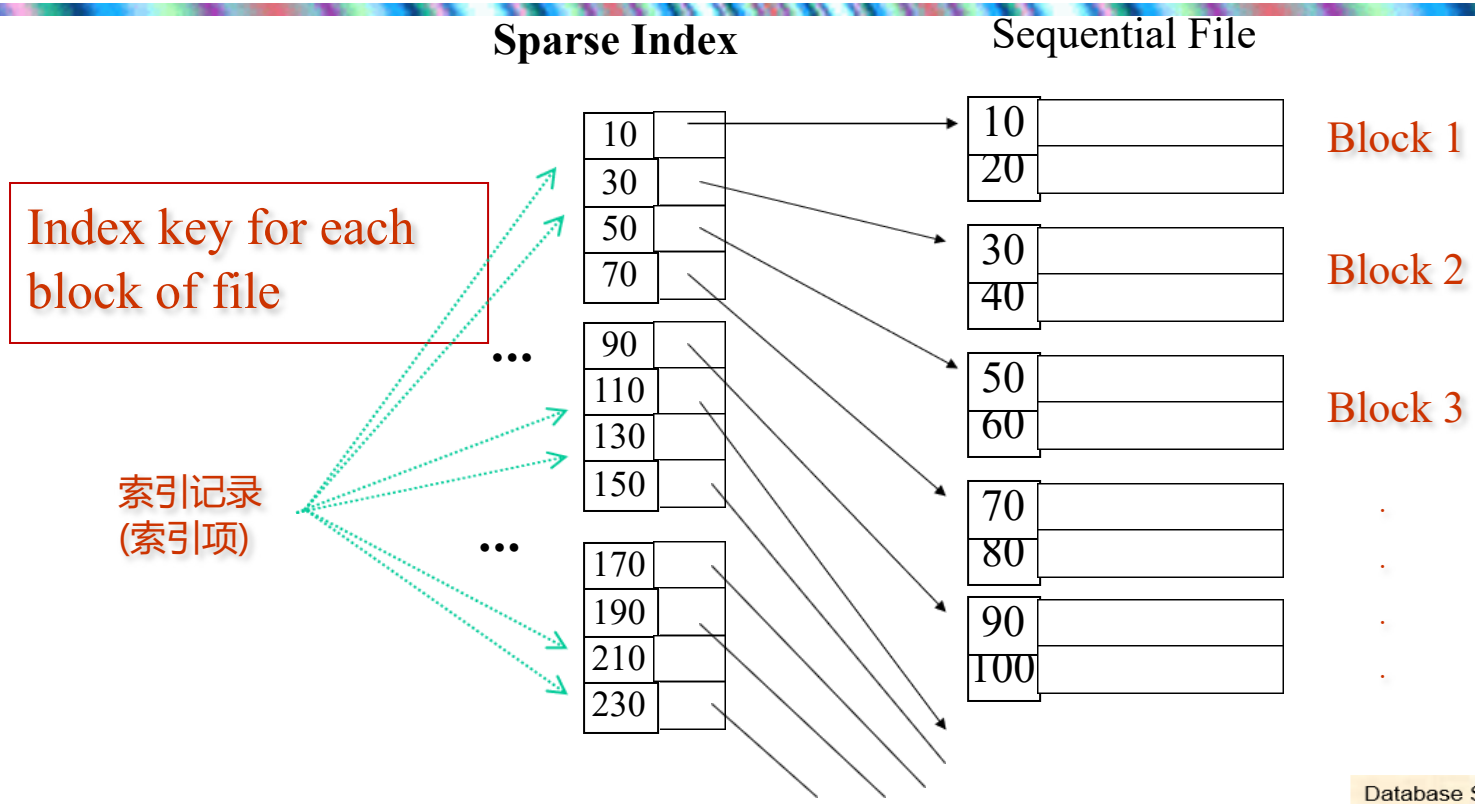

Also facilitates a full scan of a relation (or the extent of an object class) if records not stored on physically consecutive blocks

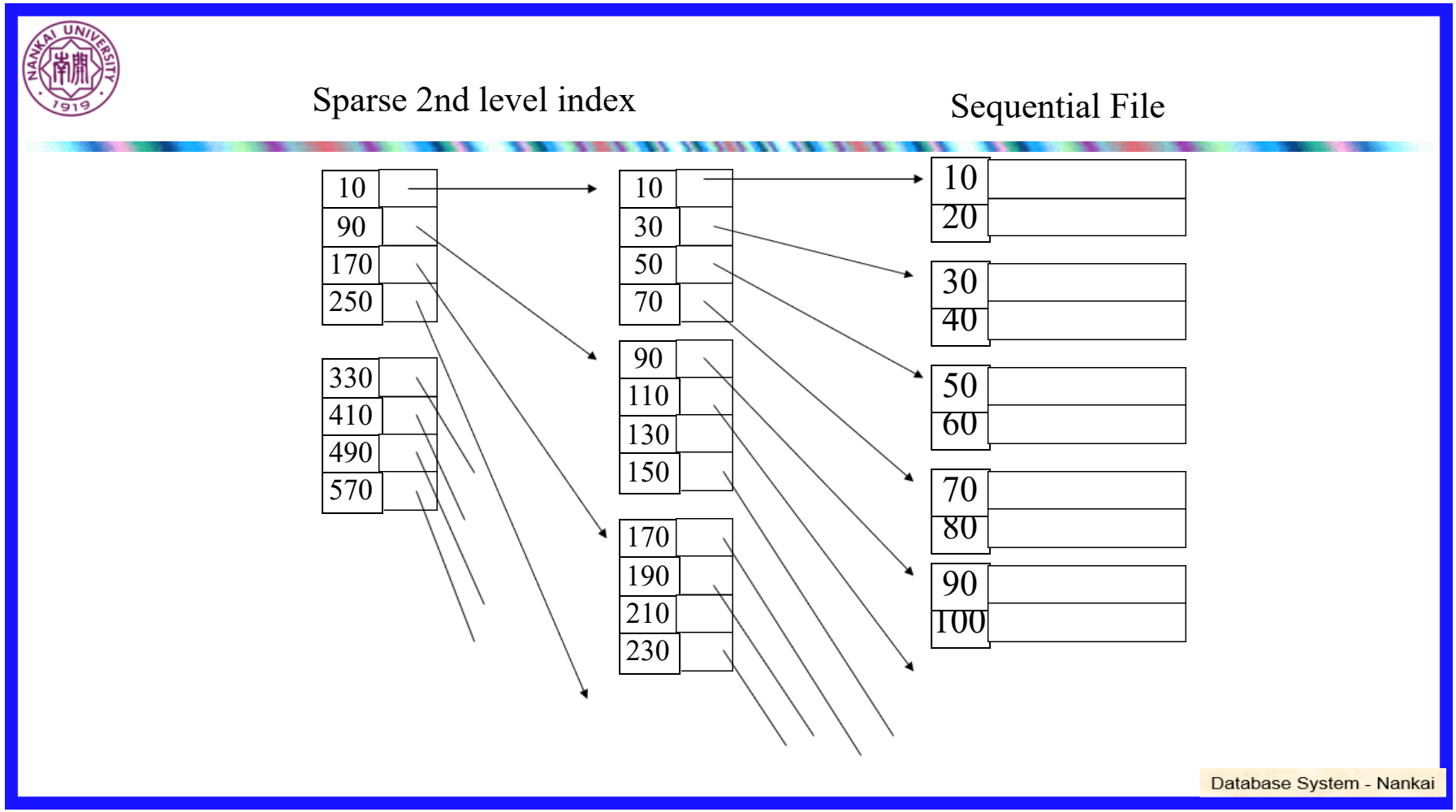
Database System - Nankai

# Dense Index（稠密索引）



# Sparse Index（稀疏索引）

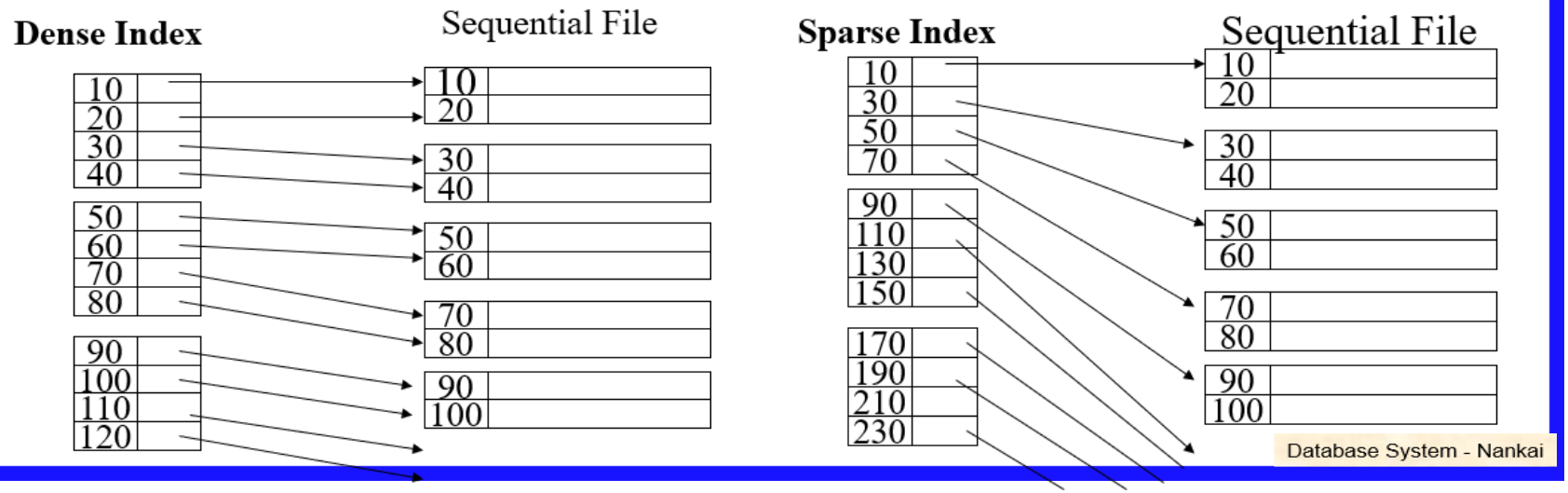



Sparse 2nd level index
Sequential File
10
90
170
250
330
410
490
570
10
30
50
70
90
110
130
150
170
190
210
230
10
20
30
40
50
60
70
80
90
100
Database System - Nankai

# Sparse vs. Dense Tradeoff

Sparse: Less index space per record
-> can keep more of index in memory

Dense: Can tell if any record exists without accessing file



# Terms

- Search key ( ≠ primary key 搜索键或者查找键)
- Primary index（主索引）
  - index locations determine locations of data blocks
- Secondary index（辅助索引）
- Dense index (contains all search key values 稠密索引)
- Sparse index（稀疏索引）
- Multi-level index（多级索引）

Database System - Nankai

# Index Structure - Summary

- The Role of index （深刻理解索引的作用）
- Dense index (contains all search key values 稠密索引)
- Primary index（主索引）
- Sparse index（稀疏索引）
- Multi-level index（多级索引）

请在学堂云上完成相关练习,加深对知识的理解!

单选题 1分

## 课前预习索引部分--互动交流

某日某趟高铁直达列车有1500人乘坐，每个座位用车厢座位号标识，每个乘坐人用身份证号标识，关系表的模式为：某日乘车（车厢座位号，身份证号、姓名），表中的元组按车厢座位号排序存储的。请问如果需要快速查找某个座位上的乘客信息，应该构建如下哪个索引?

A 在车厢座位号属性上，构建稀疏索引

B 在车厢座位号属性上，构建稠密索引

C 在身份证号属性上，构建稀疏索引

D 在身份证号属性上，构建稠密索引

提交

Database System - Nankai

单选题 1分

## 课前预习索引部分--互动交流

某日某趟高铁直达列车有1500人乘坐，每个座位用车厢座位号标识，每个乘坐人用身份证号标识，关系表的模式为：某日乘车（车厢座位号，身份证号、姓名），表中的元组按车厢座位号排序存储的。请问如果需要快速查找某身份证号的乘客坐在哪个位置，应该构建如下哪个索引?

A 在车厢座位号属性上，构建稀疏索引

B 在车厢座位号属性上，构建稠密索引

C 在身份证号属性上，构建稀疏索引

D 在身份证号属性上，构建稠密索引

提交

Database System - Nankai

单选题 1分

# 课前预习索引部分--互动交流

某日某趟高铁直达列车有1500人乘坐，每个座位用车厢座位号标识，每个乘坐人用身份证号标识，关系表的模式为：某日乘车（车厢座位号，身份证号、姓名），表中的元组按车厢座位号排序存储的。请问如果需要快速查找某个姓名的乘客坐在哪个位置，应该构建如下哪个索引?

A 在车厢座位号属性上，构建稀疏索引

B 在车厢座位号属性上，构建稠密索引

C 在姓名属性上，构建稀疏索引

D 在姓名属性上，构建稠密索引

提交

Database System - Nankai

# Week3_Course_part5

# Principles of Data Layout and Index

## Conventional Indexes

Database System - Nankai

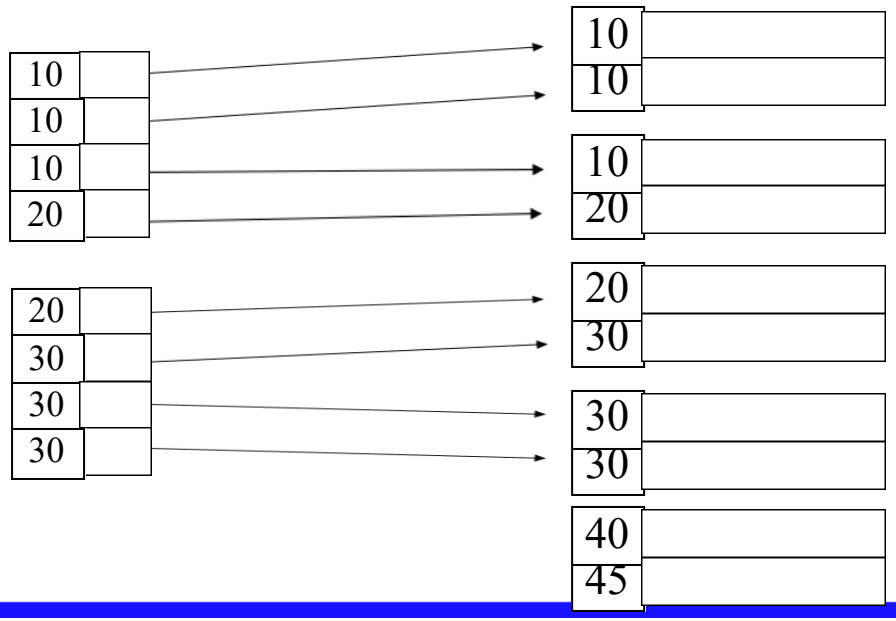
# Next:

- Duplicate keys
- Deletion/Insertion
- Secondary indexes

Database System - Nankai

# Duplicate keys

## Dense index, one way to implement?

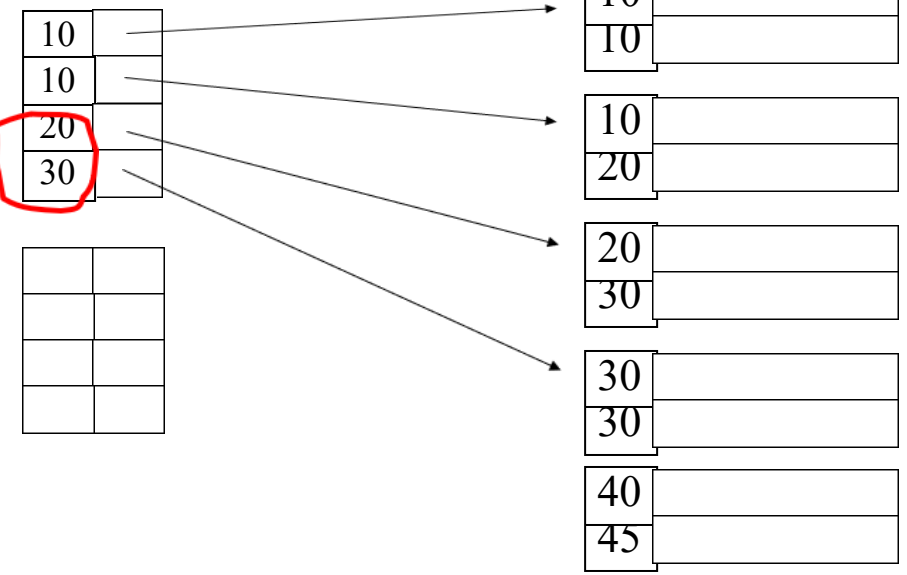



Database System - Nankai

# Duplicate keys

## Sparse index, one way?

careful if looking for 20 or 30!

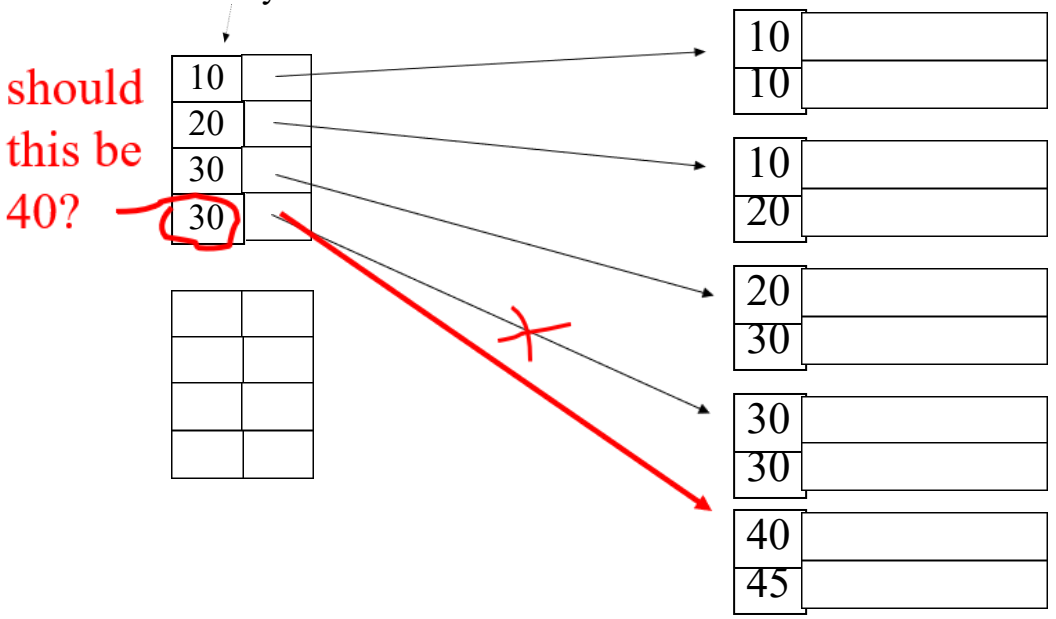



Database System - Nankai

# Duplicate keys

## Sparse index, another way?

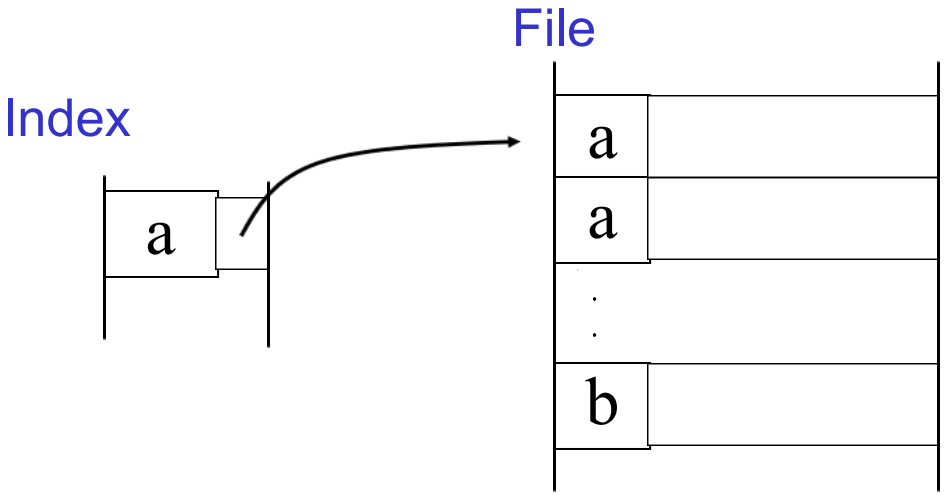
- first new key from block




Database System - Nankai

# Duplicate values, primary index

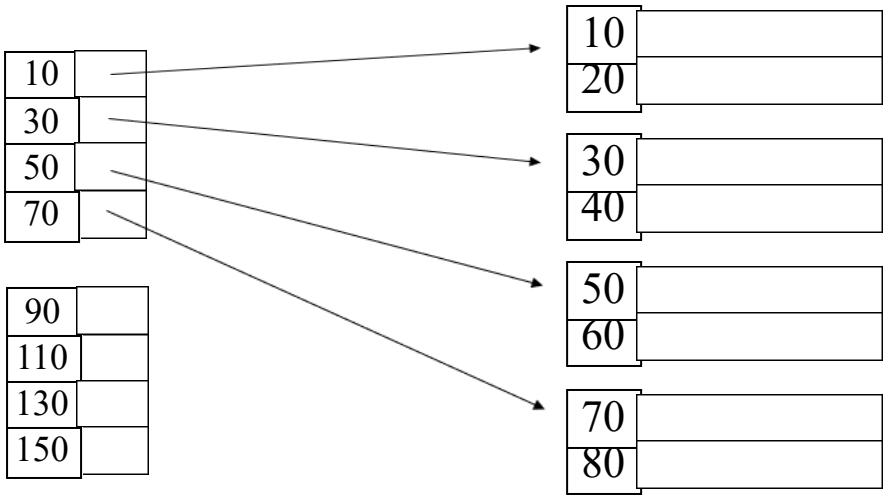
- Index may point to first instance of each value only




Database System - Nankai

# Index update operations

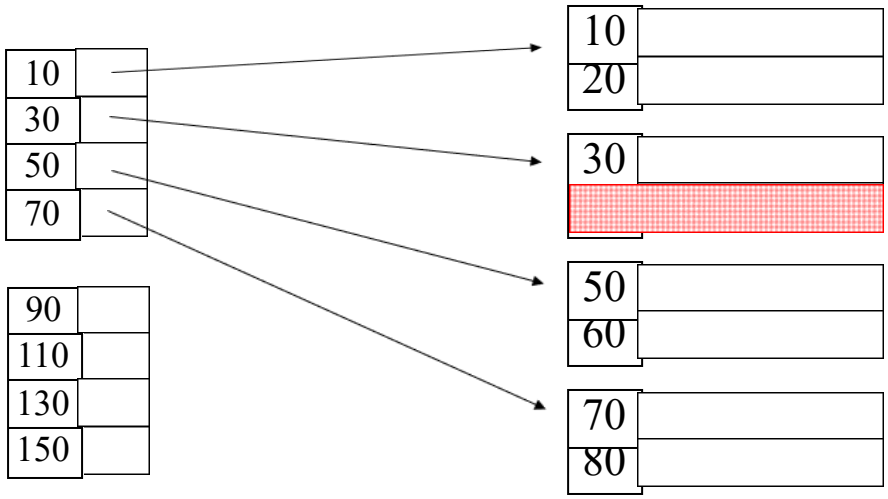
## Deletion from sparse index




Database System - Nankai

# Deletion from sparse index

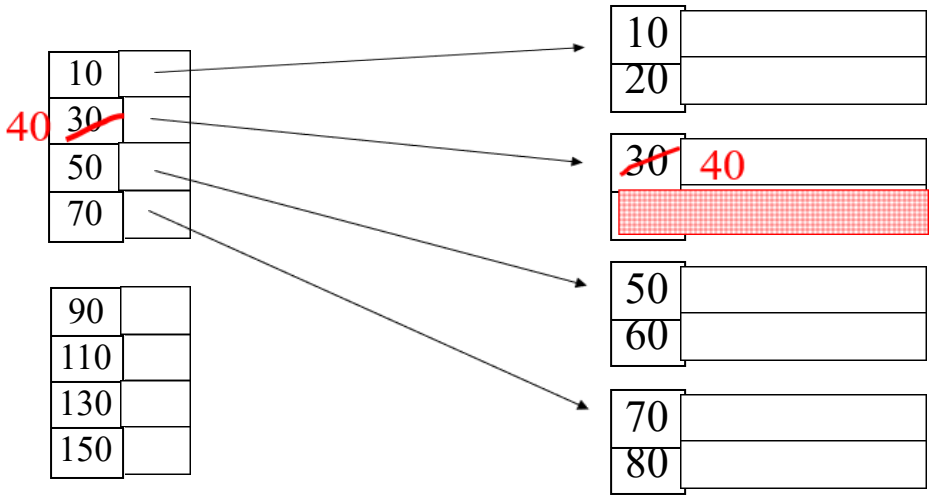
- delete record 40




Database System - Nankai

# Deletion from sparse index

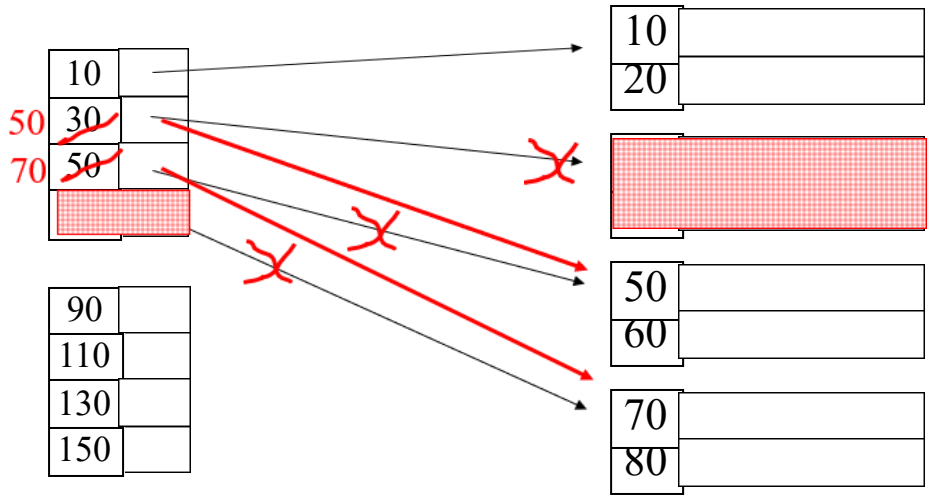
- delete record 30




Database System - Nankai

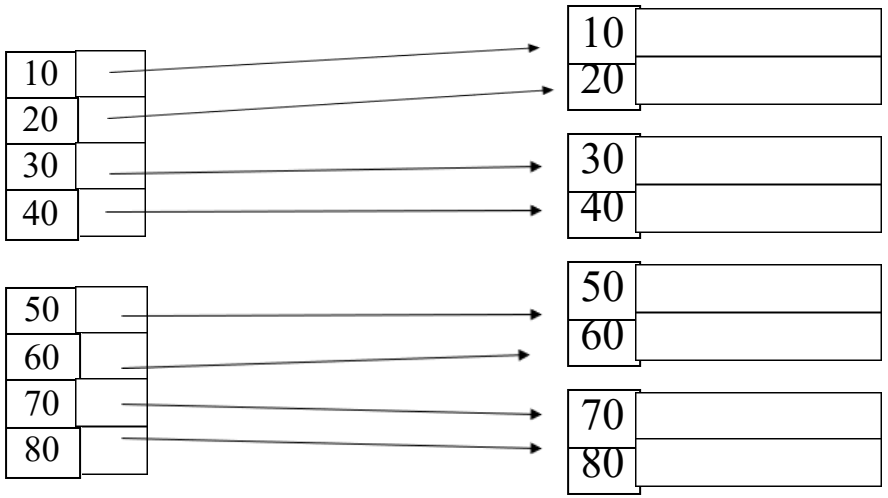
# Deletion from sparse index

- delete records 30 & 40




Database System - Nankai

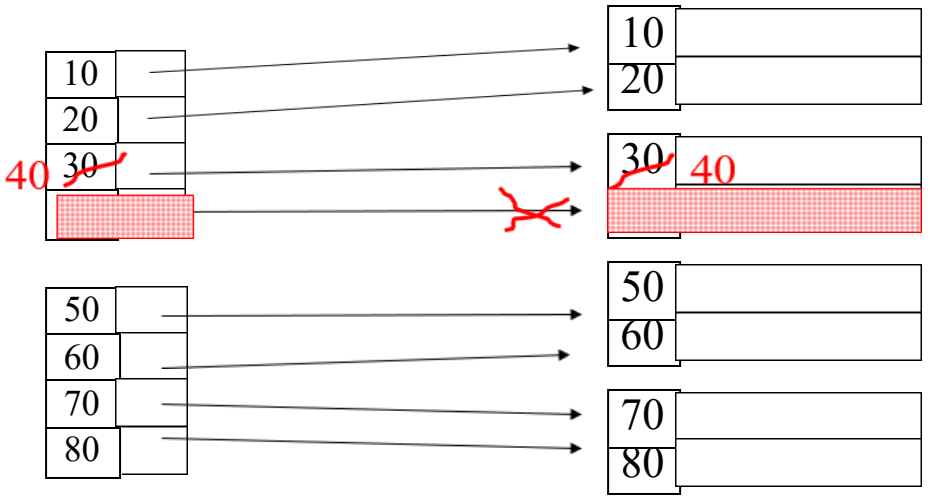
# Deletion from dense index




Database System - Nankai

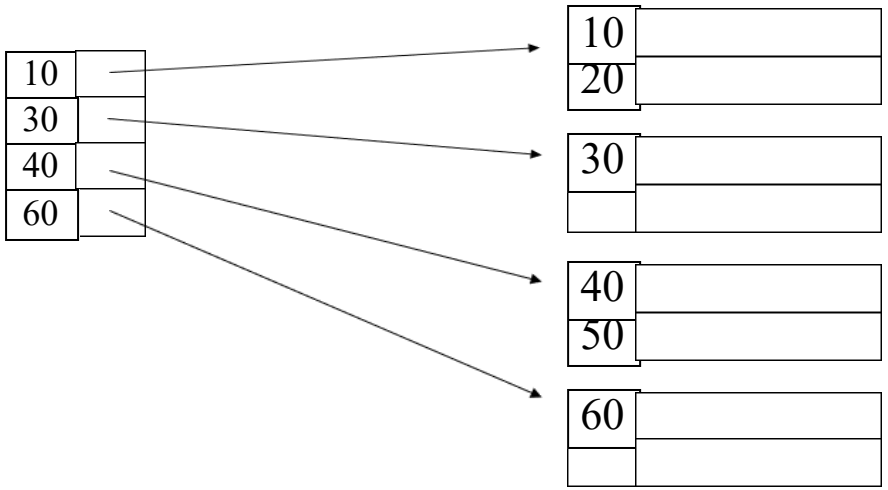
# Deletion from dense index

- delete record 30




Database System - Nankai

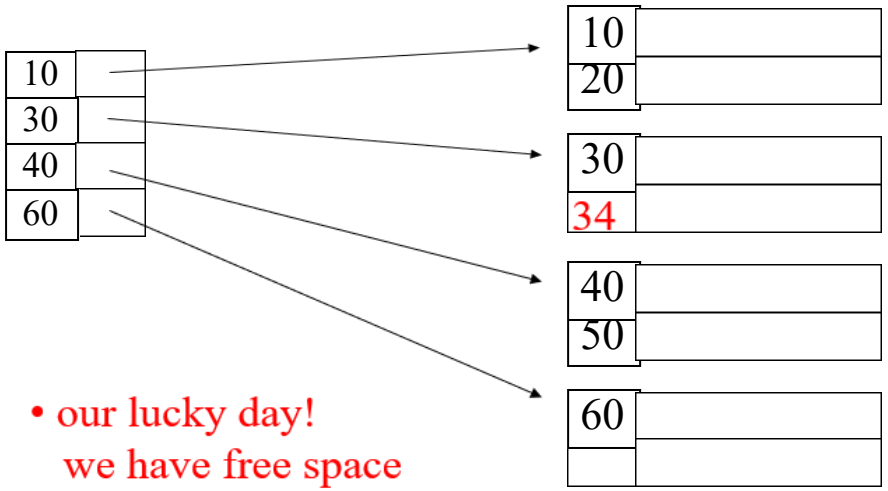
# Insertion, sparse index case




Database System - Nankai

# Insertion, sparse index case

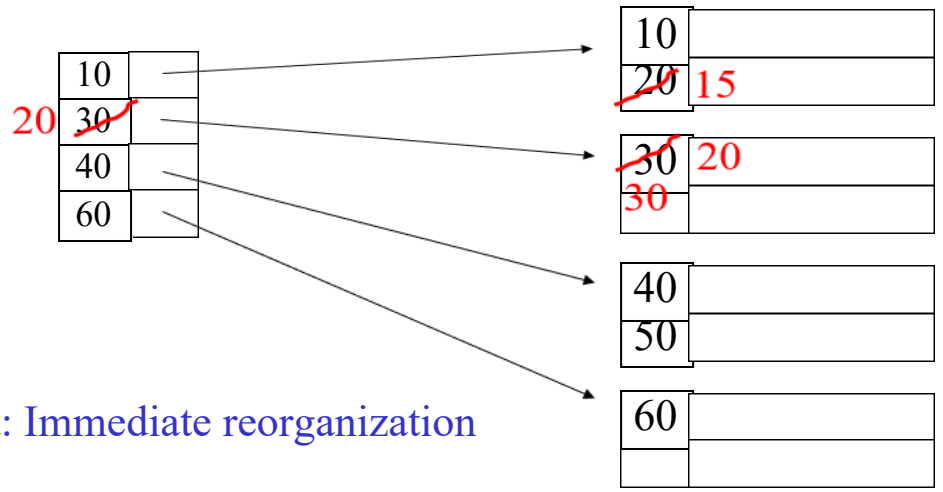
- insert record 34




- our lucky day!
  we have free space
  where we need it!

# Insertion, sparse index case

- insert record 15

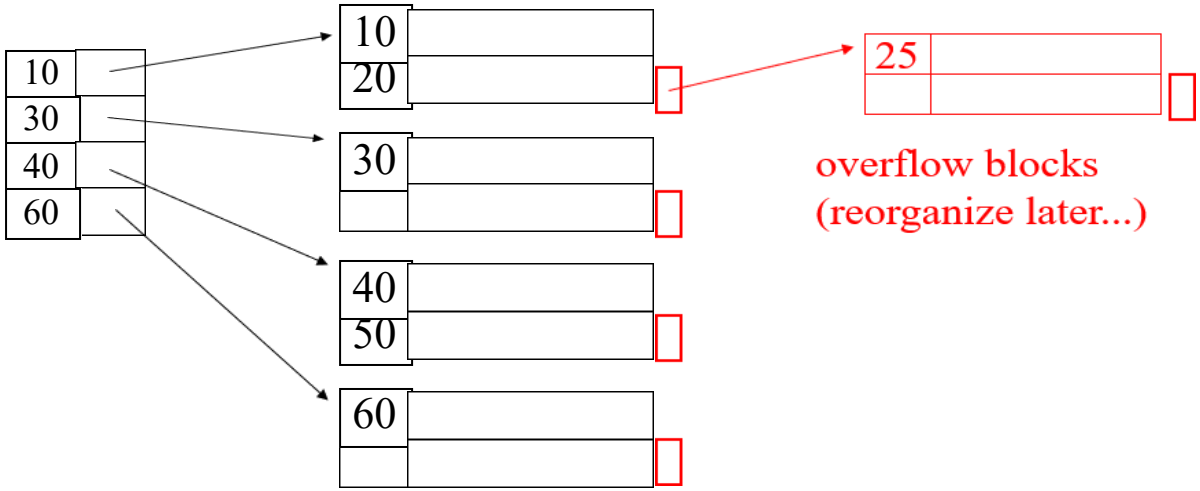



- Illustrated: Immediate reorganization
- Variation:
  - insert new block (chained file)
  - update index

Database System - Nankai

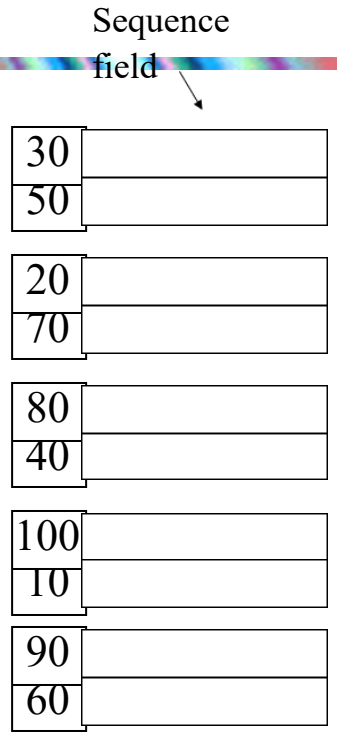
# Insertion, sparse index case

- insert record 25




Database System - Nankai

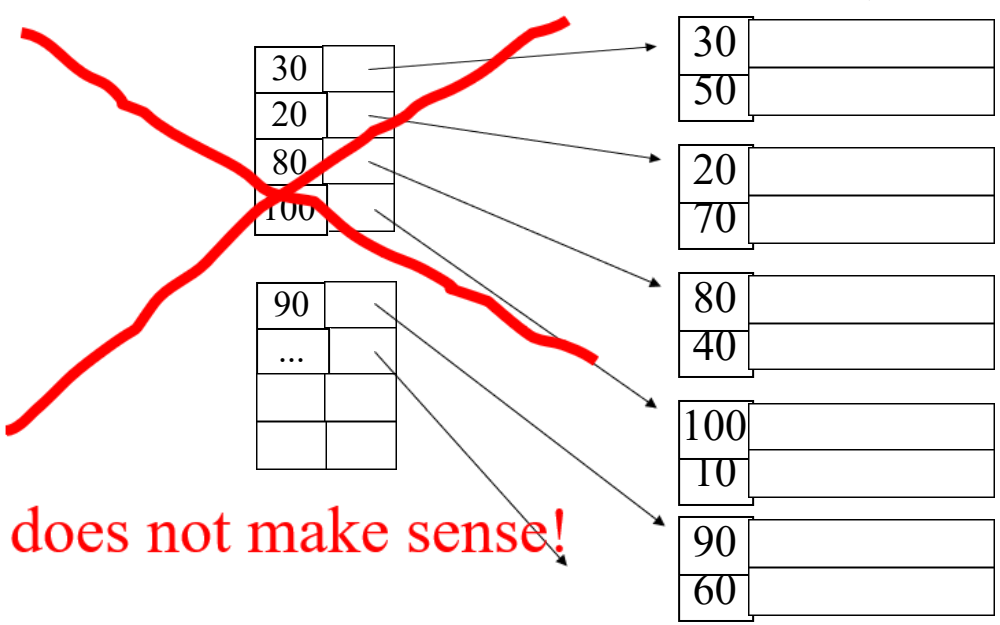
# Secondary indexes




Database System - Nankai

# Secondary indexes

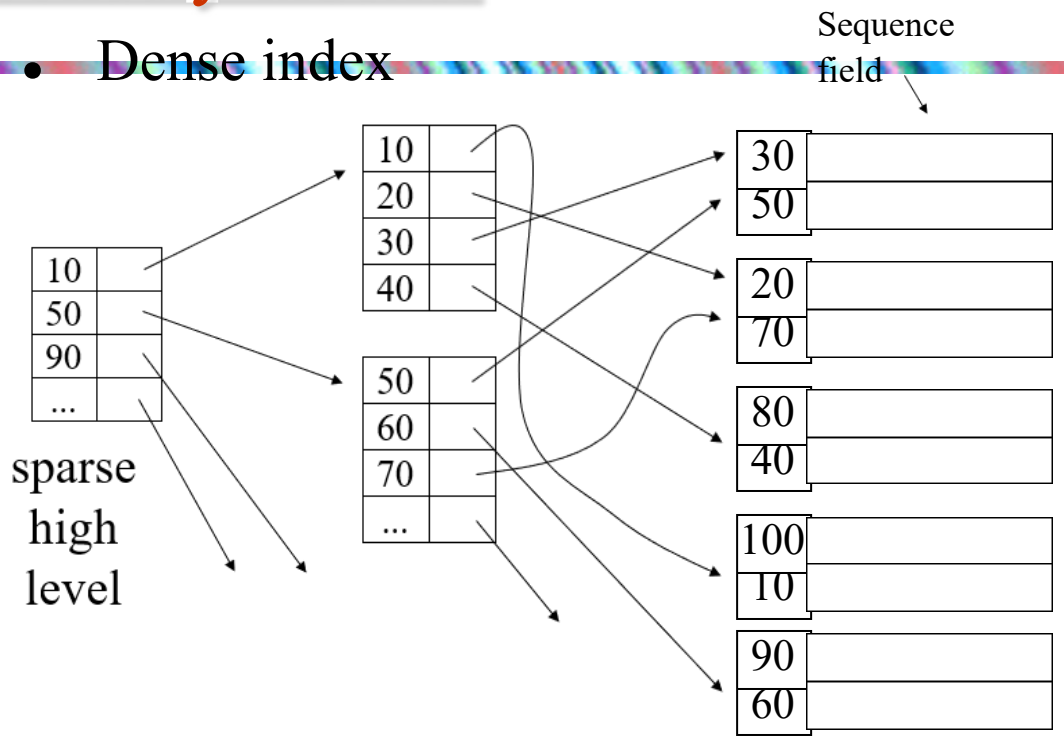
- Sparse index




Database System - Nankai

# Secondary indexes

- Dense index




Database System - Nankai

# With secondary indexes:

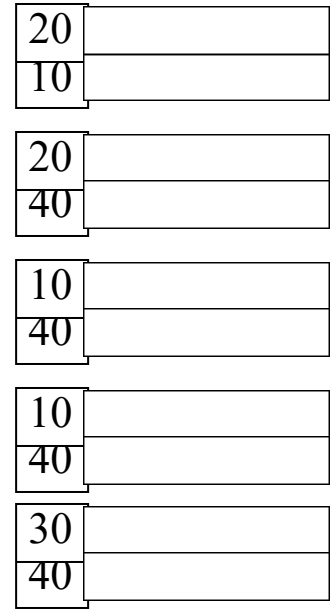
- Lowest level is dense
- Other levels are sparse

**Also: Pointers are record pointers**

**(not block pointers; not computed)**

Database System - Nankai

# Duplicate values & secondary indexes




Database System - Nankai

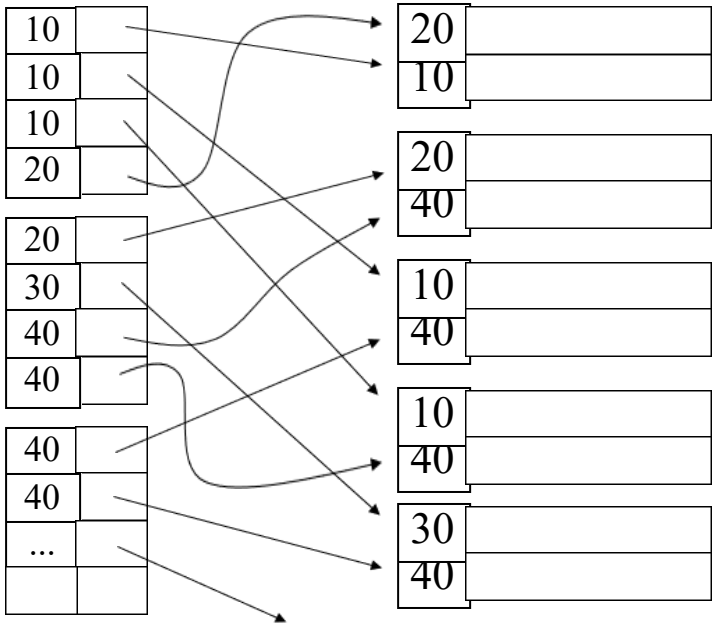
# Duplicate values & secondary indexes

one option...

**Problem:**

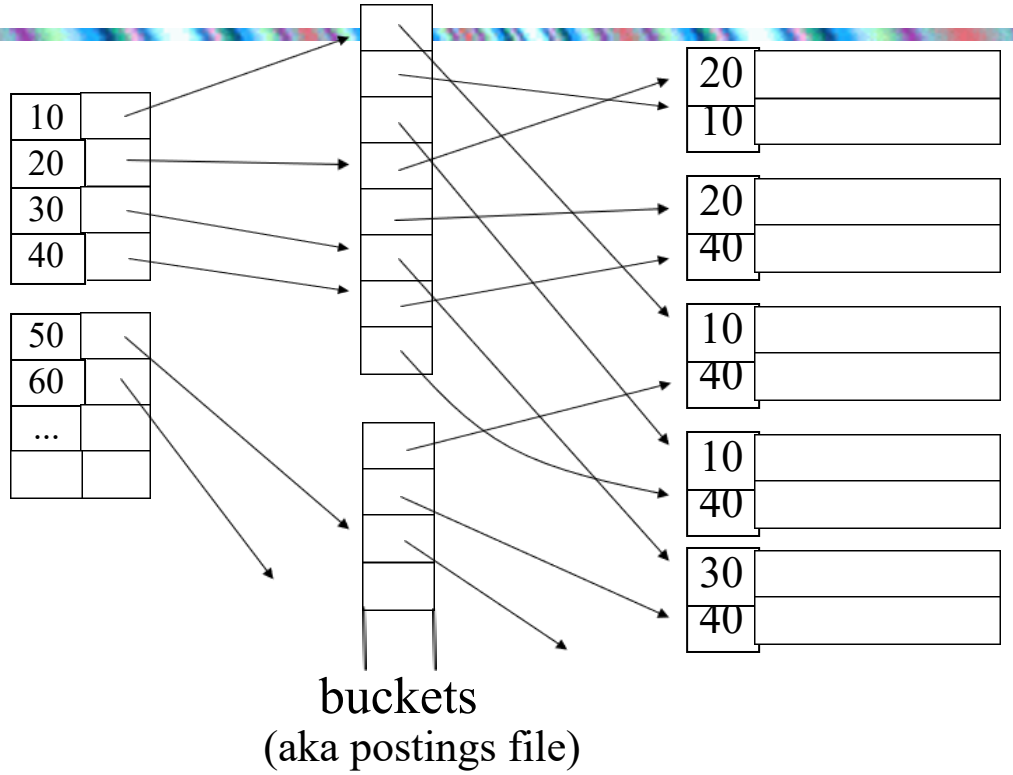
excess overhead!

- disk space
- search time




Database System - Nankai

# Duplicate values & secondary indexes



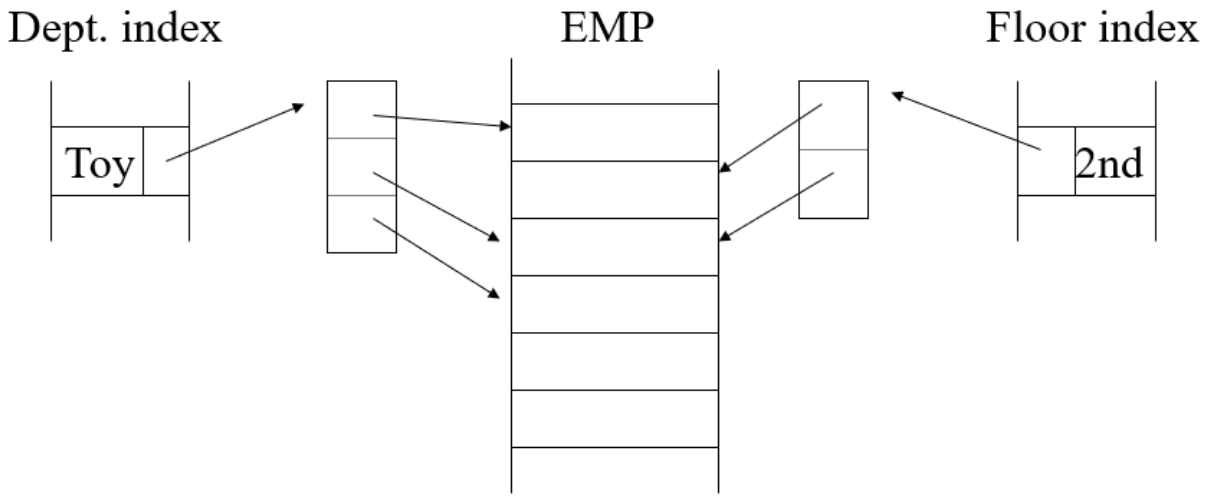

Database System - Nankai

# Why "bucket" idea is useful

Example:

| Indexes | Records |
| --- | --- |
| Name: primary | EMP(name,dept,floor,...) |
| Dept: secondary | |
| Floor: secondary | |

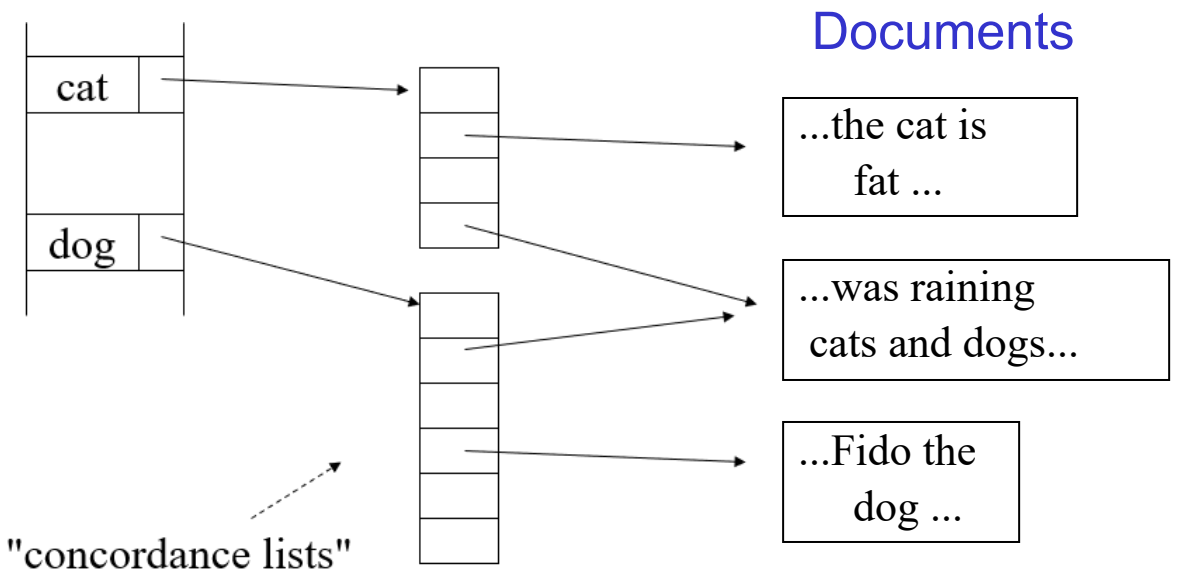
# Query: Get employees in (Toy Dept) AND (2nd floor)




→ Intersect toy bucket and 2nd Floor bucket to get set of matching EMP's

Database System - Nankai

# This idea used in text information retrieval



# IR QUERIES

- Find articles with "cat" and "dog"
- Find articles with "cat" or "dog"
- Find articles with "cat" and not "dog"

- **Find articles with "cat" in title**
- **Find articles with "cat" and "dog" within 5 words**

Database System - Nankai

# Conventional indexes

Advantage:

- Simple
- Index is sequential file
  good for scans

**Disadvantage:**

**- Inserts expensive, and/or**

**- lose sequentiality & balance**

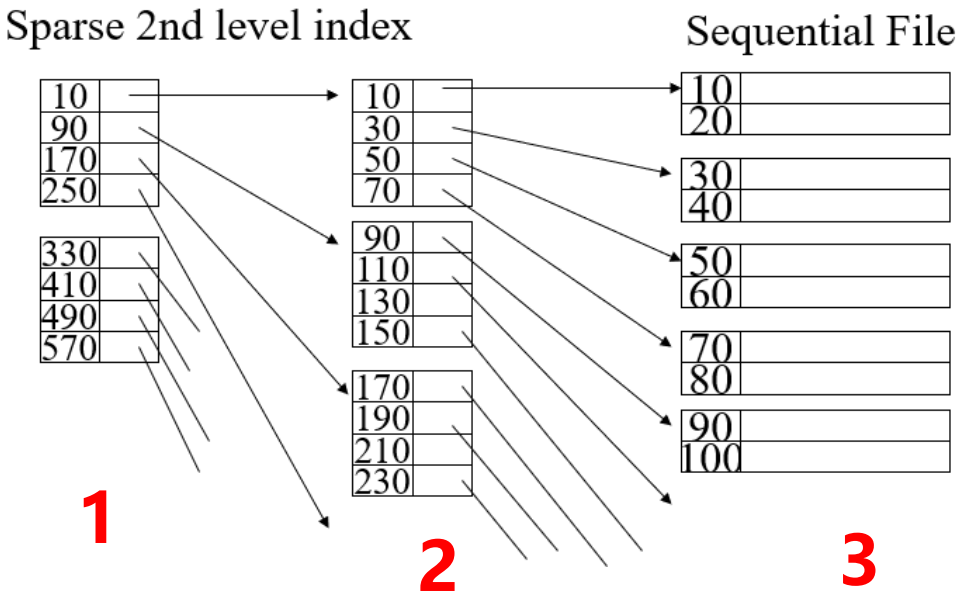
# Summary so far

- Conventional index
  - Basic Ideas: sparse, dense, multi-level…
  - Duplicate Keys
  - Deletion/Insertion
  - Secondary indexes
    - Buckets of Postings List/File

单选题 1分

# 互动交流一

下图索引哪个部分是主索引（Primary index）？




A 1

B 2

C 3

D 以上都不是

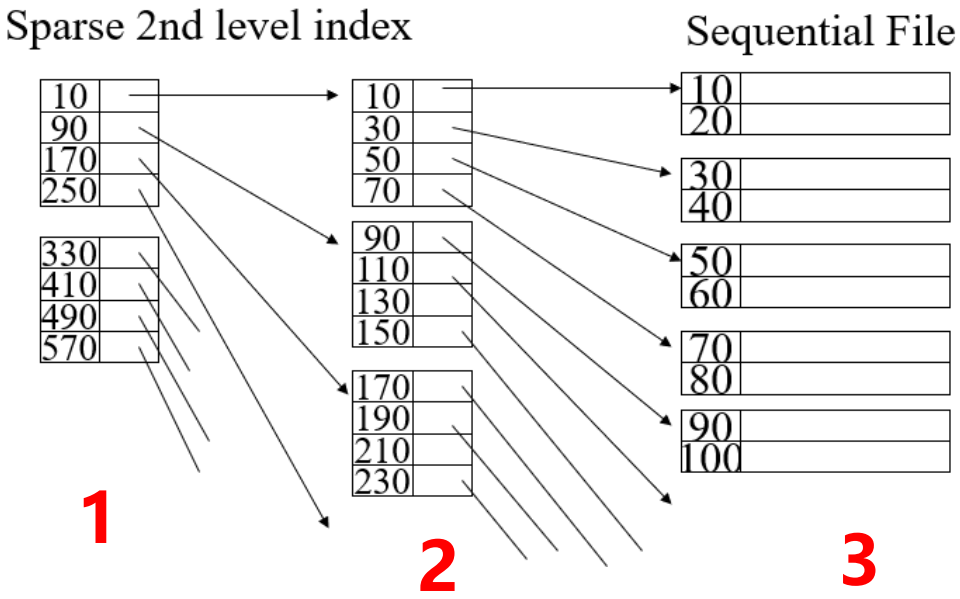
提交

Database System - Nankai

多选题 1分

# 互动交流二

下图索引哪个部分是稀疏索引（sparse index）？




A 1

B 2

C 3

D 以上都不是

提交

单选题 1分

# 互动交流三

判断以下描述是否正确？

A secondary index is one in which keys don’t necessarily have to be ordered.

A TRUE

B FALSE

提交

Database System - Nankai

# Week3_Course_part6

# Principles of Data Layout and Index

## B+ Tree Indexes

Database System - Nankai

# Principles of Data Layout and Index

How to find a record quickly?

本周课

NEXT ⇒

磁盘的结构与访问特征

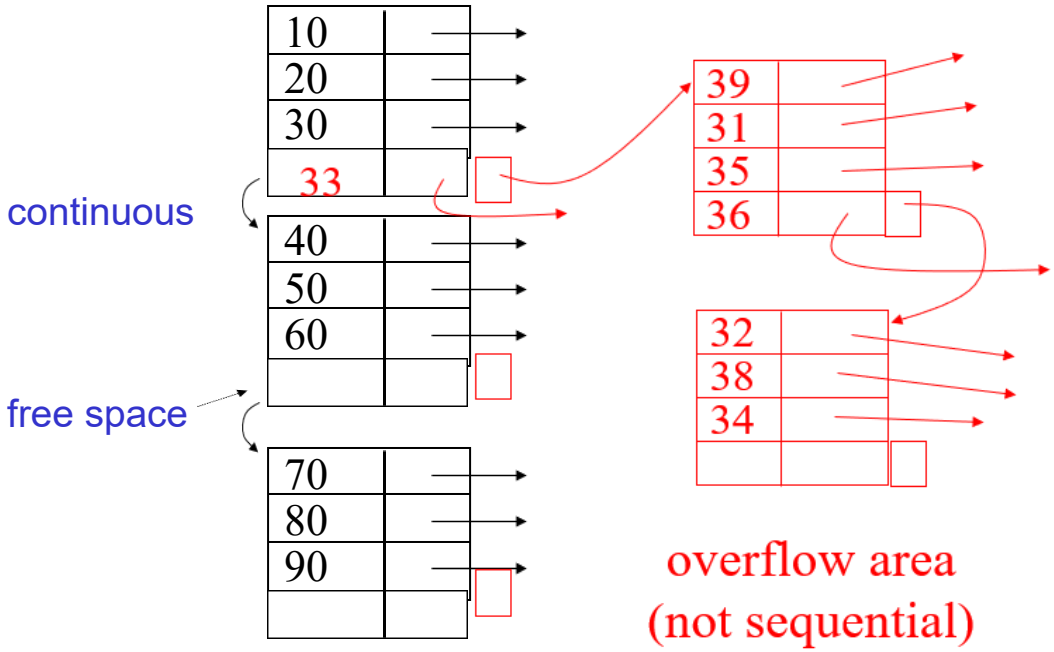
Conventional indexes

B+Trees

Hashing schemes

Database System - Nankai

# Example Index (sequential)




Database System - Nankai

# Outline:

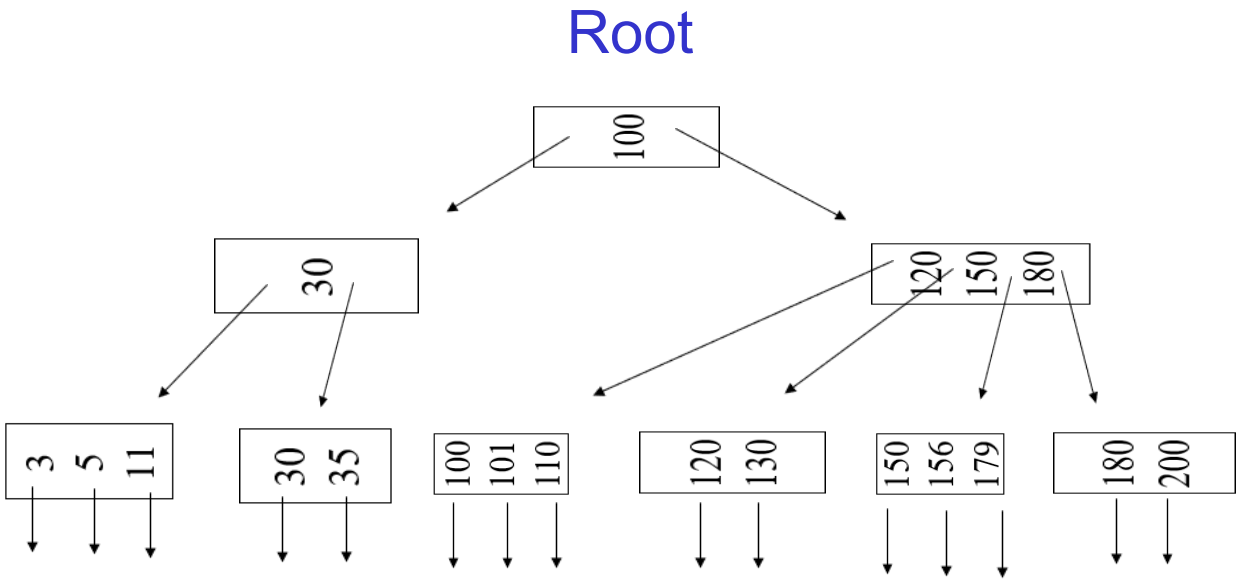
- Conventional indexes
- B-Trees ⇒ NEXT
  - a commonly used index structure that adapts well to insertions and deletions
- Hashing schemes

Database System - Nankai

- B-Trees
  - a commonly used index structure
  - nonsequential, “balanced” (acces paths to different records of equal length)
  - adapts well to insertions & deletions
  - consists of blocks holding at most *n* keys and *n*+1 pointers, and at least half of this
  - (We consider a variation actually called a B+ tree)

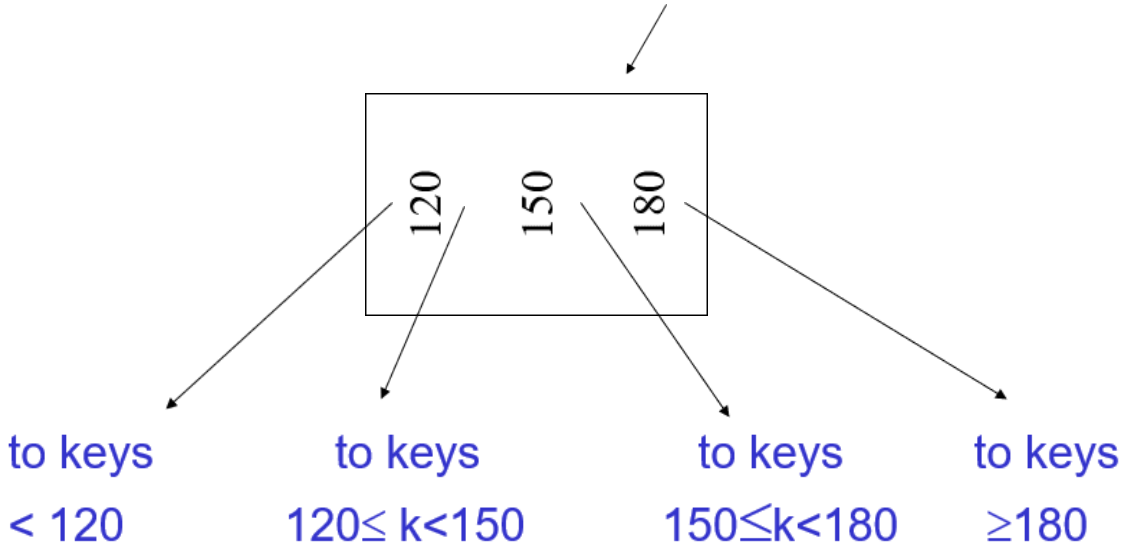
Database System - Nankai

# B+Tree Example n=3



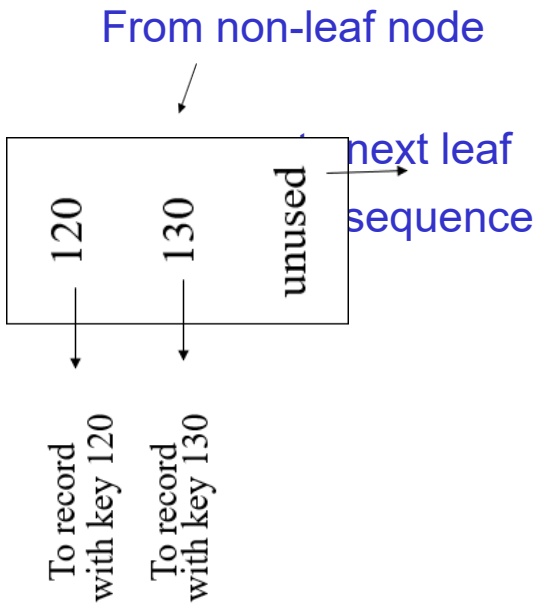

Database System - Nankai

# Sample non-leaf




Database System - Nankai

# Sample leaf node:



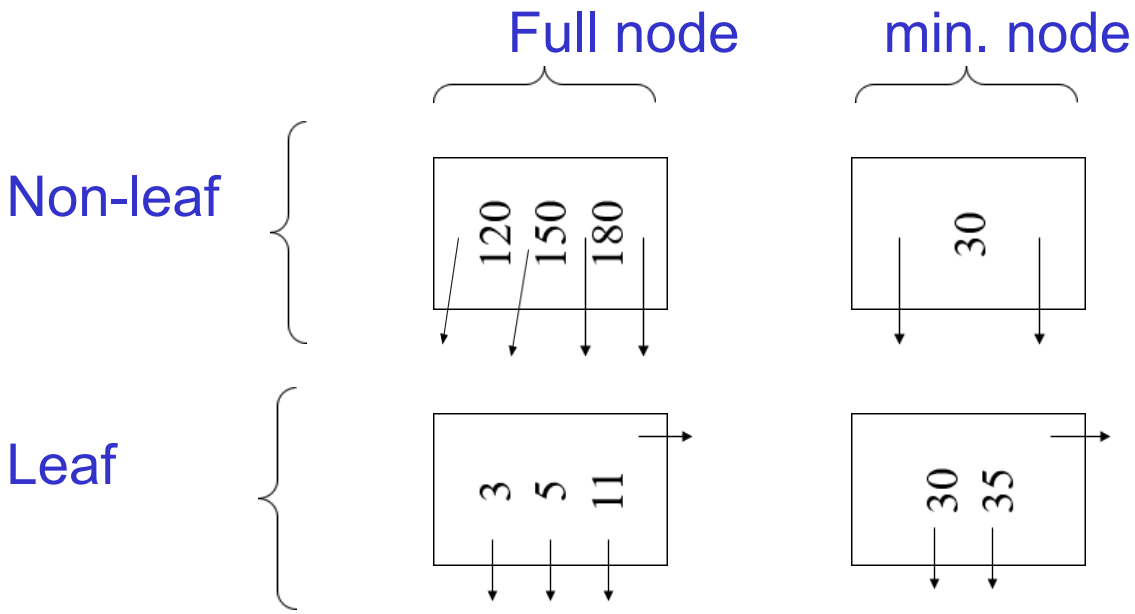
# Don’ t want nodes to be too empty

- Number of pointers in use:
  - at internal nodes at least $\lceil (n+1)/2 \rceil$ (to child nodes)
  - at leaves at least $\lfloor (n+1)/2 \rfloor$ (to data records/blocks)

$$\lceil x \rceil = \min\{n \in Z \mid n \geq x\}$$
$$\lfloor x \rfloor = \max\{n \in Z \mid n \leq x\}$$

Database System - Nankai

n=3




Database System - Nankai

# B+tree rules

(1) All leaves at the same lowest level (balanced tree)

(2) Pointers in leaves point to records
except for "sequence pointer"

(3) Number of pointers/keys for B+tree

| | Max ptrs | Max keys | Min ptrs→data | Min keys |
|---|---|---|---|---|
| Non-leaf (non-root) | n+1 | n | $\lceil (n+1)/2 \rceil$ | $\lceil (n+1)/2 \rceil - 1$ |
| Leaf (non-root) | n+1 | n | $\lfloor (n+1)/2 \rfloor$ | $\lfloor (n+1)/2 \rfloor$ |
| Root | n+1 | n | $2^{(*)}$ | 1 |

(*) 1, if only one record in the file

# Insert into B+tree

First lookup the proper leaf

(a) simple case

- leaf not full: just insert (key, pointer-to-record)

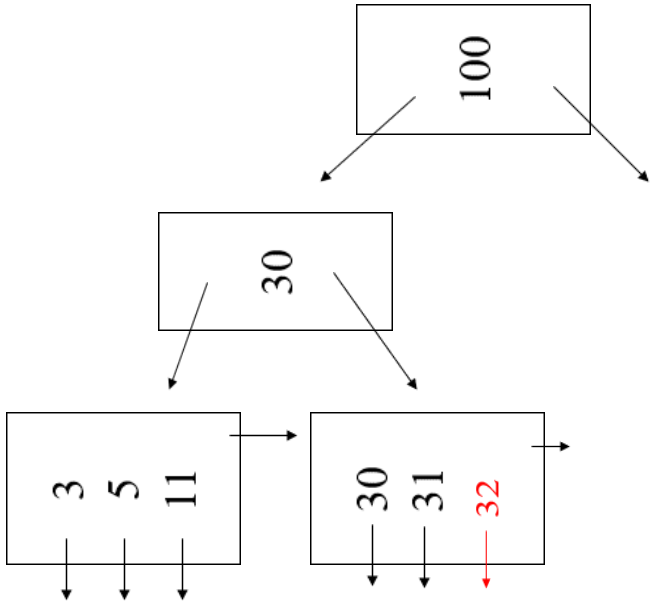
(b) leaf overflow

(c) non-leaf overflow

(d) new root

# (a) Insert key = 32

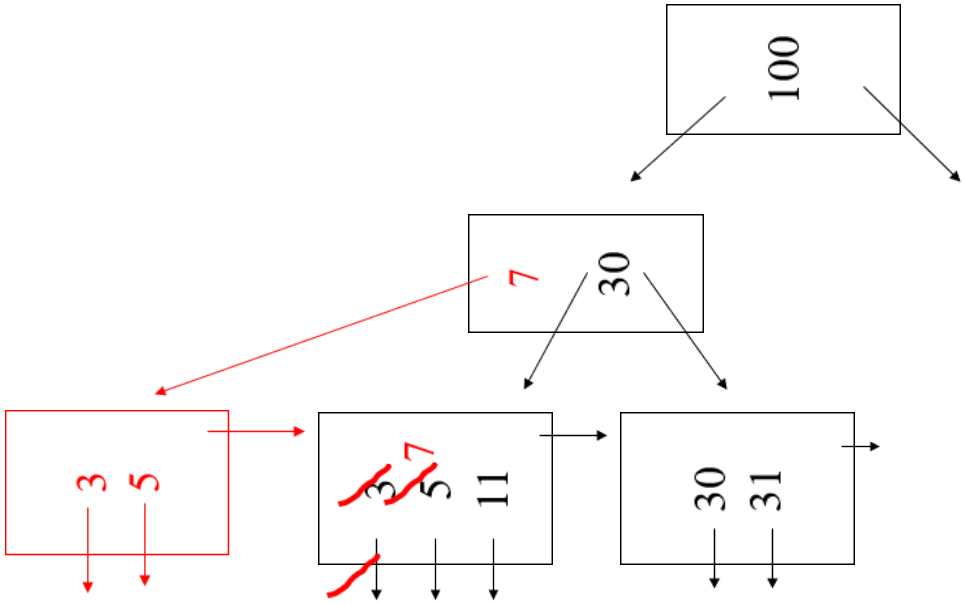
n=3




Database System - Nankai

# (b) Insert key = 7

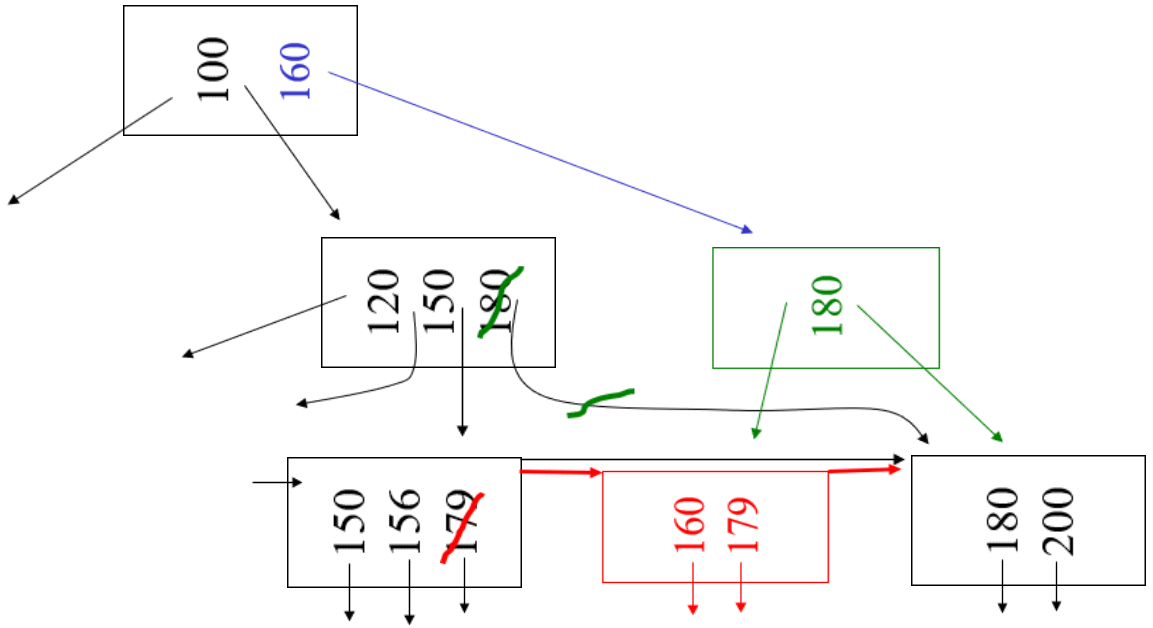
n=3




Database System - Nankai

# (c) Insert key = 160

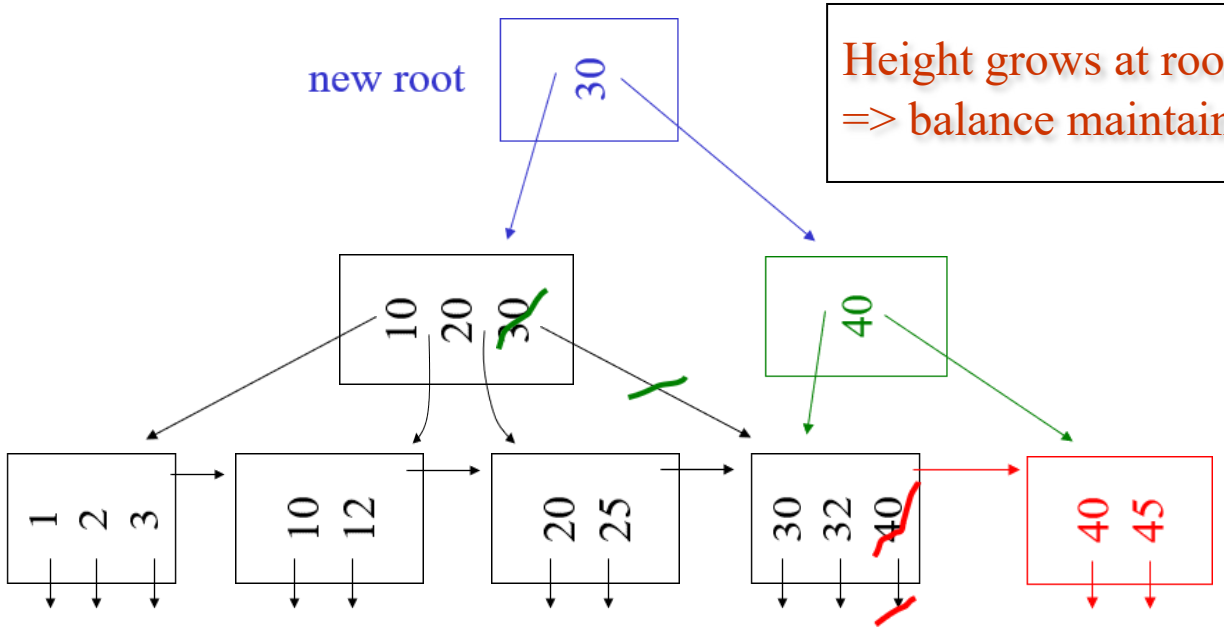
n=3




Database System - Nankai

# (d) New root, insert 45

n=3




Height grows at root
=> balance maintained

Database System - Nankai

# Deletion from B+tree

Again, first lookup the proper leaf;

(a): Simple case: no underflow; Otherwise ...

(b): Borrow keys from an adjacent sibling
(if it doesn't become too empty); Else ...

(c): Coalesce with a sibling node

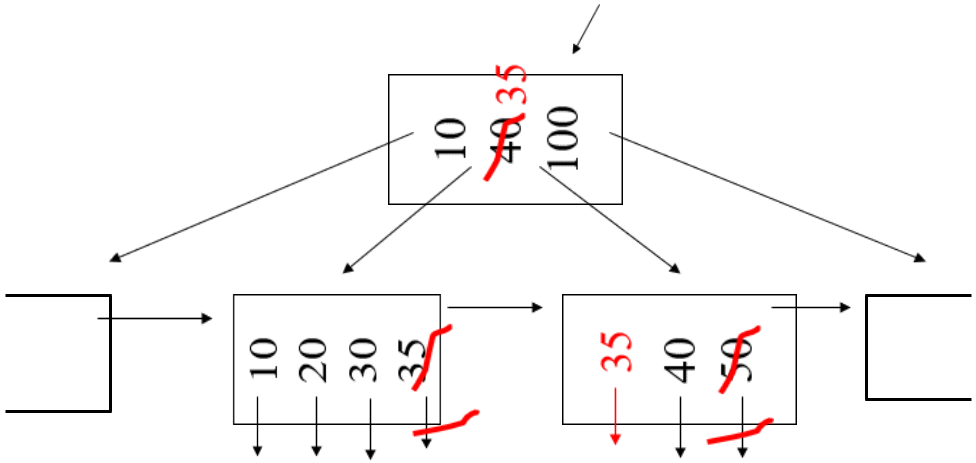
(d): Cases (a), (b) or (c) at non-leaf

## (b) Borrow keys

– Delete 50

n=4

=> min # of keys in a leaf = $\lfloor 5/2 \rfloor = 2$

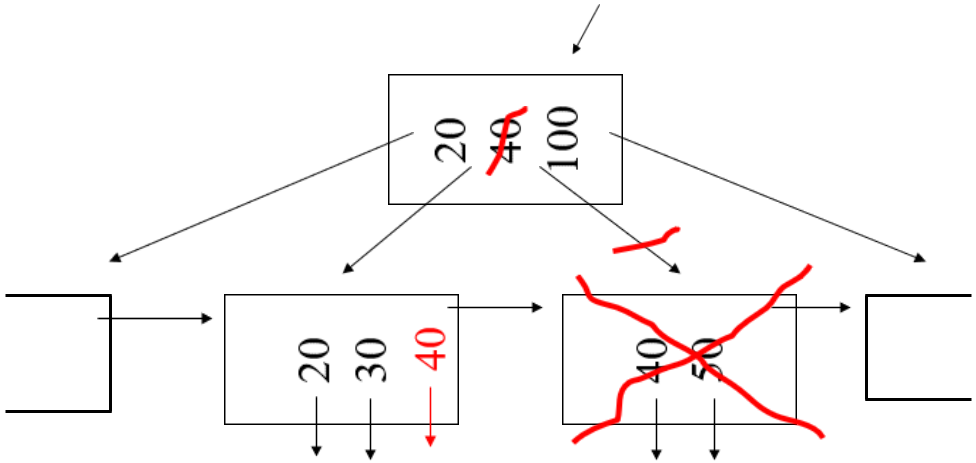



Database System - Nankai

## (c) Coalesce with a sibling

n=4

- Delete 50



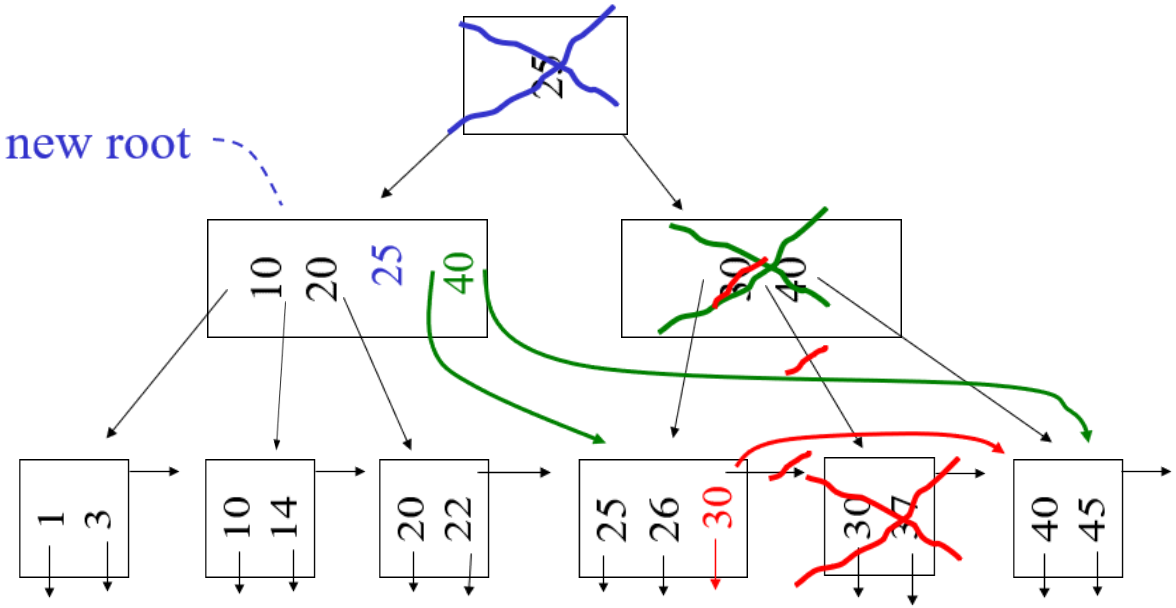

Database System - Nankai

# (d) Non-leaf coalesce

– Delete 37

n=4

=> min # of keys in a non-leaf = ⌈(n+1)/2⌉ - 1=3-1= 2




Database System - Nankai

# B+tree deletions in practice

- Often, coalescing is not implemented
  - Too hard and not worth it!
  - later insertions may return the node back to its required minimum size
  - Compromise: Try redistributing keys with a sibling;
    If not possible, leave it there
  - if all accesses to the records go through the B-tree, can place a "tombstone" for the deleted record at the leaf

Database System - Nankai

# Why B-trees Are Good?

- B-tree adapts well to insertions and deletions, maintaining balance
  - DBA does not need to care about reorganizing
  - split/merge operations rather rare
    (How often would nodes with 200 keys be split?)

–> Access times dominated by key-lookup
(i.e., traversal from root to a leaf)

Database System - Nankai

# Efficiency of B-trees

- For example, assume 4 KB blocks, 4 byte keys and 8 byte pointers
- How many keys and pointers fit in a node (= index block)?
  - Max n s.t. (4*n + 8*(n+1)) B ≤ 4096 B ?

    -> n=340; 340 keys and 341 pointers fit in a node

    -> 171 ... 341 pointers in a non-leaf node

Database System - Nankai

# Efficiency of B-trees (cont.)

- Assume an average node has 255 pointer

-> a three-level B-tree has $255^2 = 65025$ leaves with total of $255^3$ or about 16.6 million pointers to records

-> if root block kept in main memory, each record can be accessed with 2+1 disk I/Os;

If all 256 internal nodes are in main memory, record access requires 1+1 disk I/Os

(256 x 4 KB = 1 MB; quite feasible!)

Database System - Nankai

# B+ Tree Index - Summary

B+树是DBMS在存储层采用的核心索引技术，同学们要深刻体会B+树的动态调整特性，掌握B+树的机理。

Database System - Nankai

填空题 1分

# 互动交流一

Relation R has 4000 records. Suppose a B+ tree index is built on R’s key attribute. A block can hold a B+ tree node (interior or leaf node) with 20 keys and 21 pointers. In this case, at least [填空1] blocks are needed to hold the entire B+ tree index?

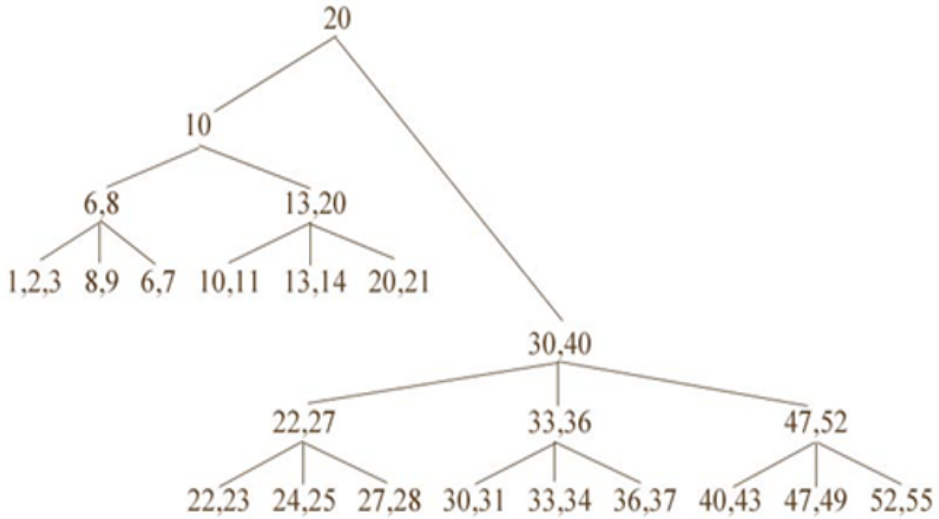
提交

se System - Nankai

主观题 10分

# 互动交流二

Find any/all violations of a B+ tree structure in the following diagram. Circle each bad node and give a brief explanation of each error. Assume the order of the tree is 4 (n=4; 4 keys)




提交

Database System - Nankai

# Week3_Course_part6

# Principles of Data Layout and Index

## B+ Tree Indexes

Database System - Nankai

# Principles of Data Layout and Index

How to find a record quickly?

本周课

磁盘的结构与访问特征

Conventional indexes

B+Trees

NEXT ⇒ Hashing schemes

Database System - Nankai

# Week3_Course_part7

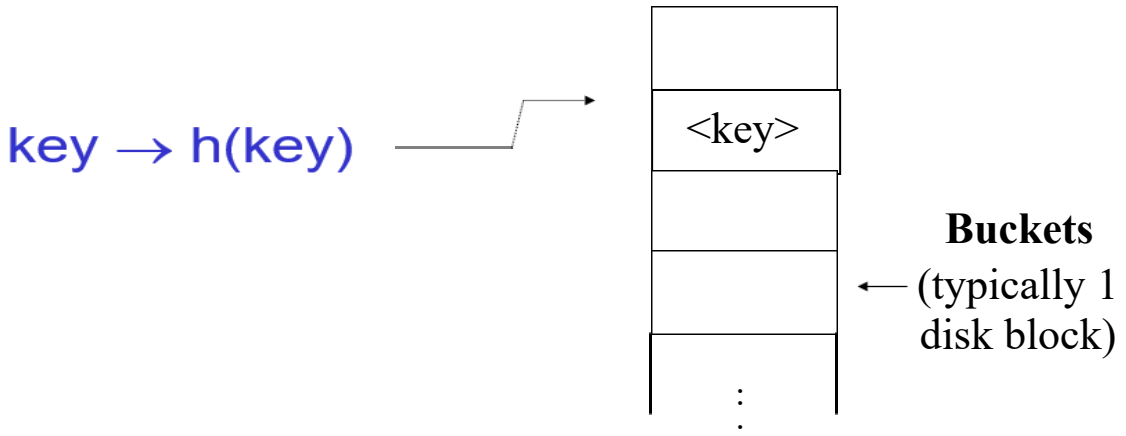
# Principles of Data Layout and Index

## Hashing

Database System - Nankai

# Hashing?

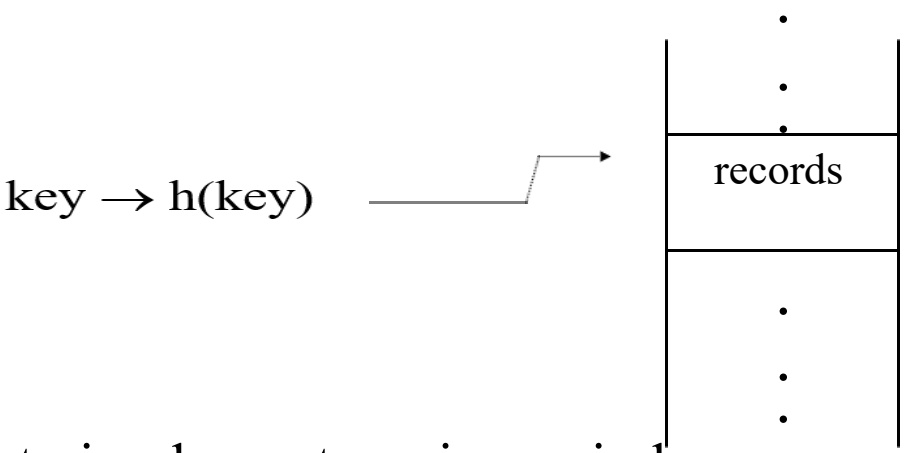
- Locating the storage block of a record by the hash value *h(k)* of its key *k*
- Normally really fast
  - records (often) located by a single disk access




Database System - Nankai

# Two alternatives

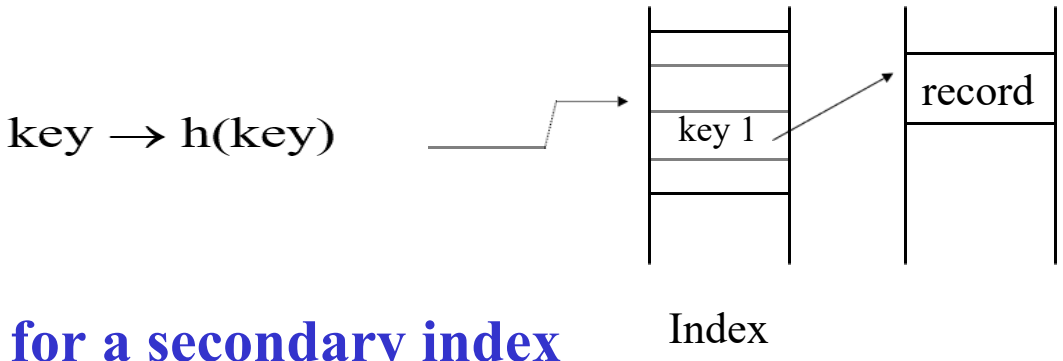
(1) Hash value determines the storage block directly




- to implement a primary index

# Two alternatives

(2) Records located indirectly via index buckets




- **for a secondary index**

# Example hash function

- Key = '$x_1$ $x_2$ … $x_n$'  *n* byte character string
- Have *b* buckets
- h = ($x_1$ + $x_2$ + … + $x_n$) mod b ∈ {0, 1, …, b-1}

Good hash function

Expected number  of function:

keys/bucket is the same for all buckets

# 一个实用的hash函数

J.D.Ullman提出了一个较为实用的Hash函数:

（1）把关键字值化为二进制位串；

（2）把二进制位串分成等长的几组，若最后一组位数不足则补0凑齐；

（3）将各组二进制位串相加，得到一个整数；

（4）将该整数除以桶的个数，其余数作为相应记录的桶地址

# EXAMPLE 2 records/bucket

INSERT:

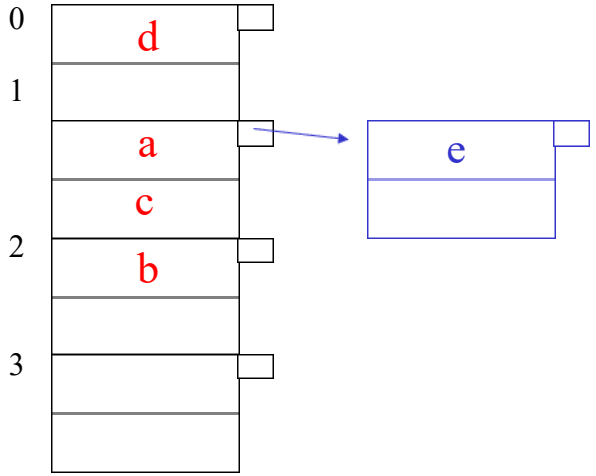
h(a) = 1

h(b) = 2

h(c) = 1

h(d) = 0

**h(e) = 1**




Database System - Nankai

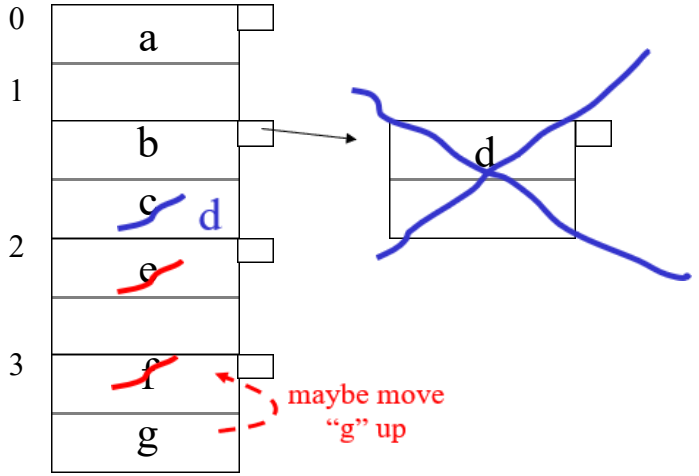
# EXAMPLE: deletion

Delete:

e

f

c




Database System - Nankai

# Rule of thumb:

- Try to keep space utilization between 50% and 80%

$$\text{Utilization} = \frac{\text{\# keys used}}{\text{total \# keys that fit}}$$

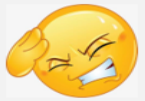
- **If < 50%, wasting space**
- **If > 80%, overflows significant**
  - **depends on how good hash function is & on # keys/bucket**

# 数据库中Hash方法的使用困难

一共有多少个学号记录(key)?

一共有多少个盒子(block)?

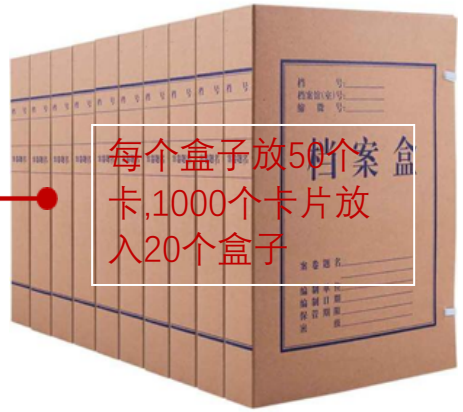
希望一个盒子放多少记录(block)?



key会增长
block随key增长

查找20190675
学生记录

20190675



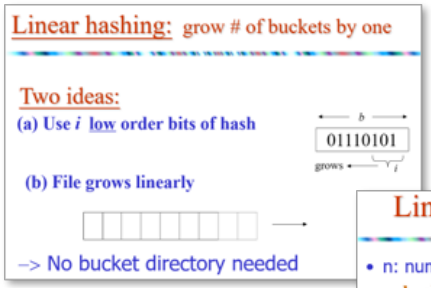
每个盒子放50个卡,1000个卡片放入20个盒子

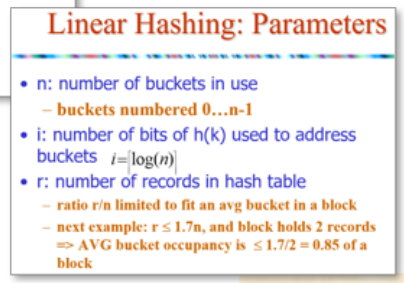
Database System - Nankai

# How do we cope with growth?

- Overflows and reorganizations
- Dynamic hashing: # of buckets may vary

- **Extensible**
- **Linear**
- **also others ...**




Database System - Nankai

# 本周课程总结

How to find a record quickly?

本周课

- 磁盘的结构与访问特征
- Conventional indexes（sparse, dense, Primary index, Secondary index）
- B+Trees（B+tree rules, Insertion, Deletion ）
- Hashing schemes