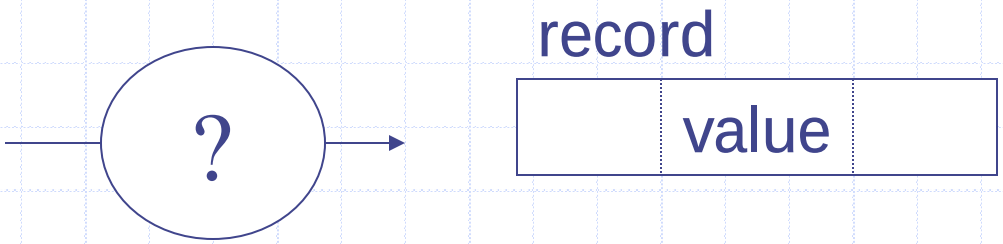
Database System Principles

chapter 4: Indexing

Chapter 4

Indexing

value



Topics

- Conventional indexes
- B+trees
- Hashing

简介

- *对于任何索引，拥有搜索码值***k** *的数据目录项***k***有三种:**
  K包含数据记录
  <k, 拥有搜索码**k的数据记录号id**>
  <k,拥有搜索码**k的数据记录号id** 的串>
- 树结构索引支持范围查找和对等值查找*range searches* and *equality searches*.
- *索引顺序存取方法ISAM*（Indexed Sequential Access Method）：是一个静态索引结构;
- *B+ tree*: 是动态的，能进行动态地插入和删除.

Sequential File

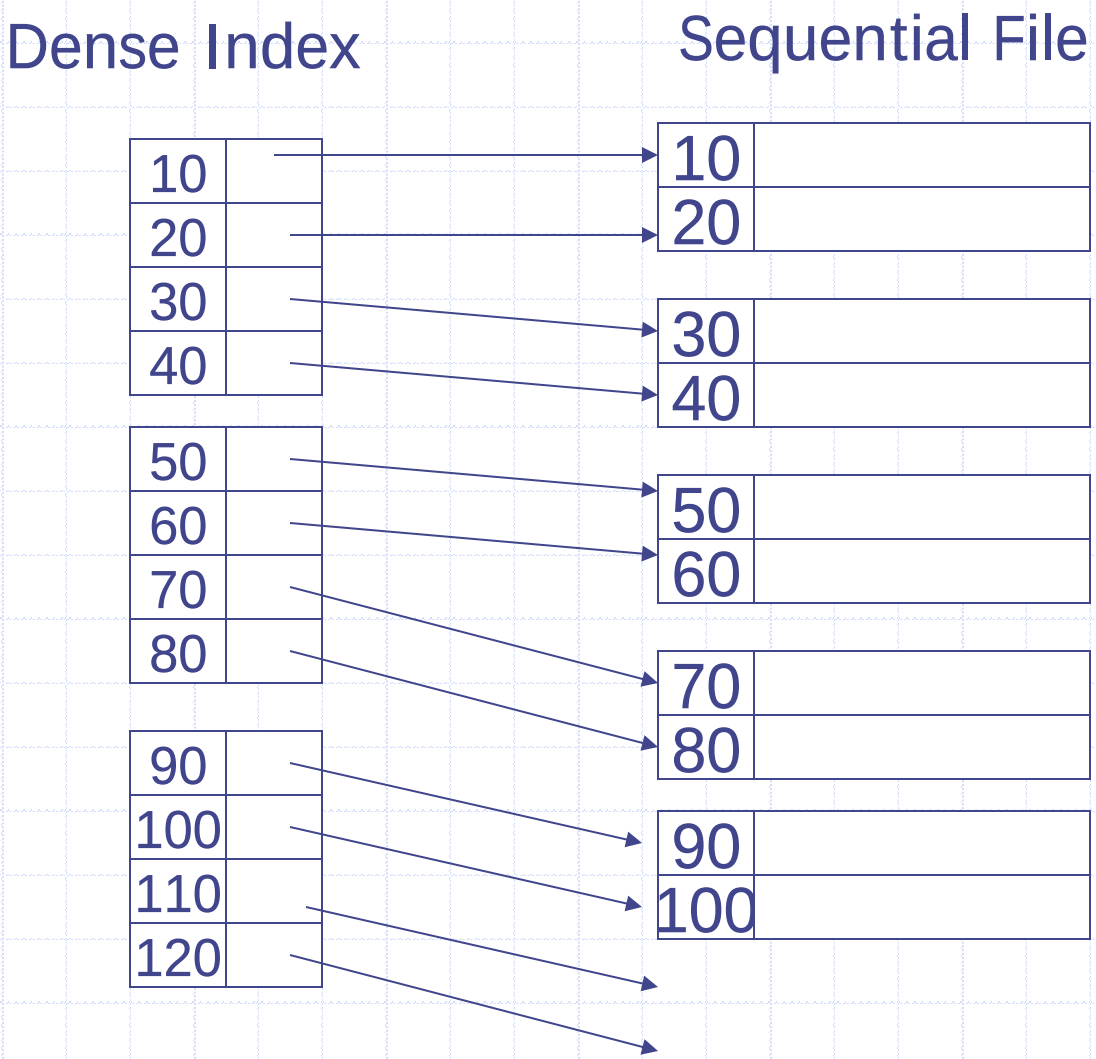
10	
20	

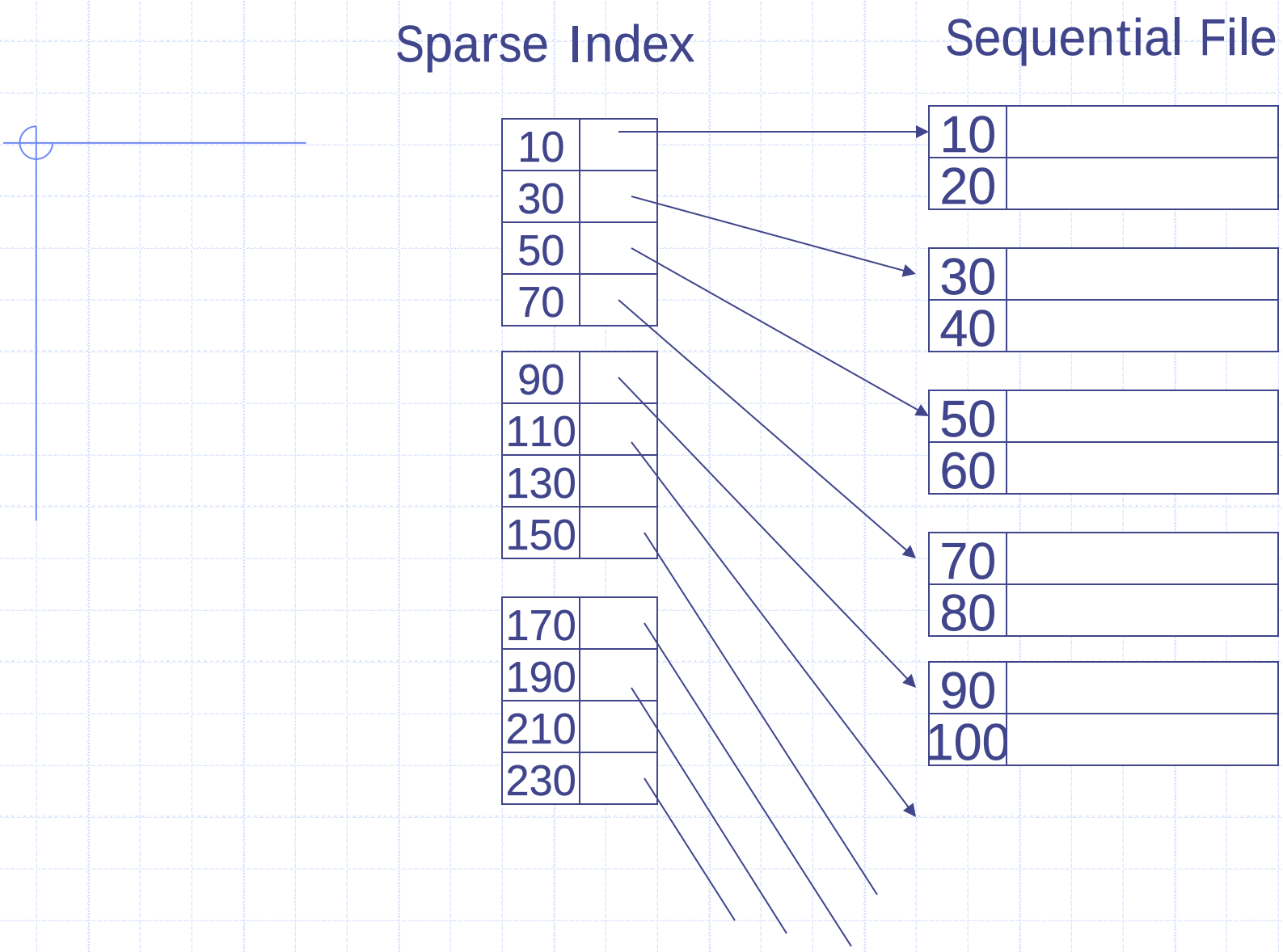
30	
40	

50	
60	

70	
80	

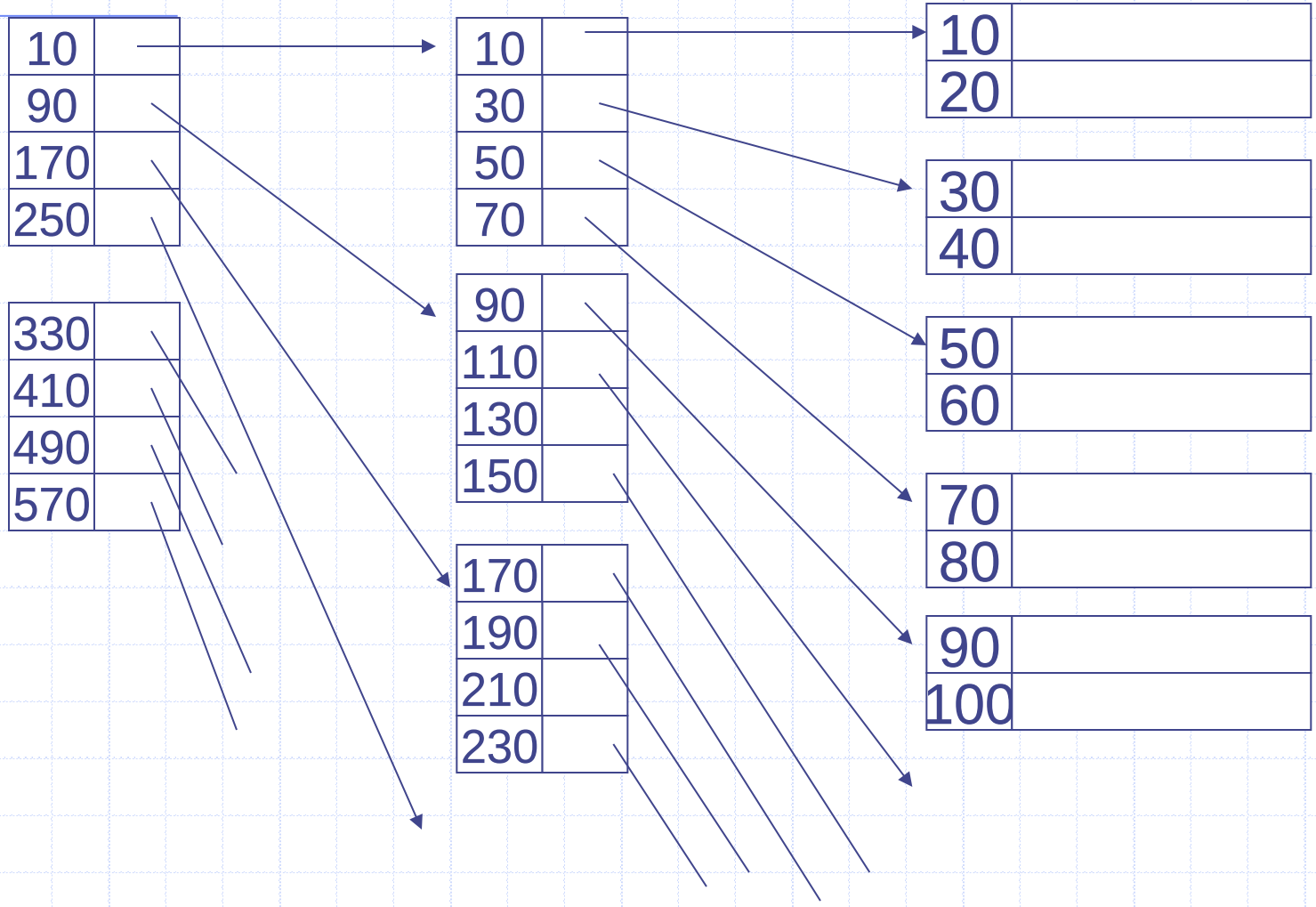
90	
100	


Dense Index
Sequential File
10
20
30
40
50
60
70
80
90
100
110
120
10
20
30
40
50
60
70
80
90
100


Sparse Index
Sequential File
10
30
50
70
90
110
130
150
170
190
210
230
10
20
30
40
50
60
70
80
90
100

Sparse 2nd level

Sequential File

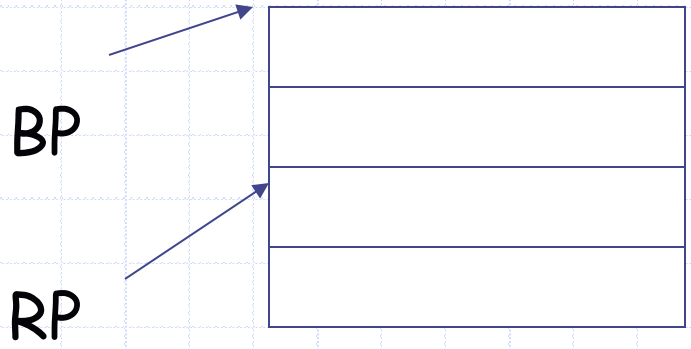


Question:

- Can we build a dense, 2nd level index for a dense index?

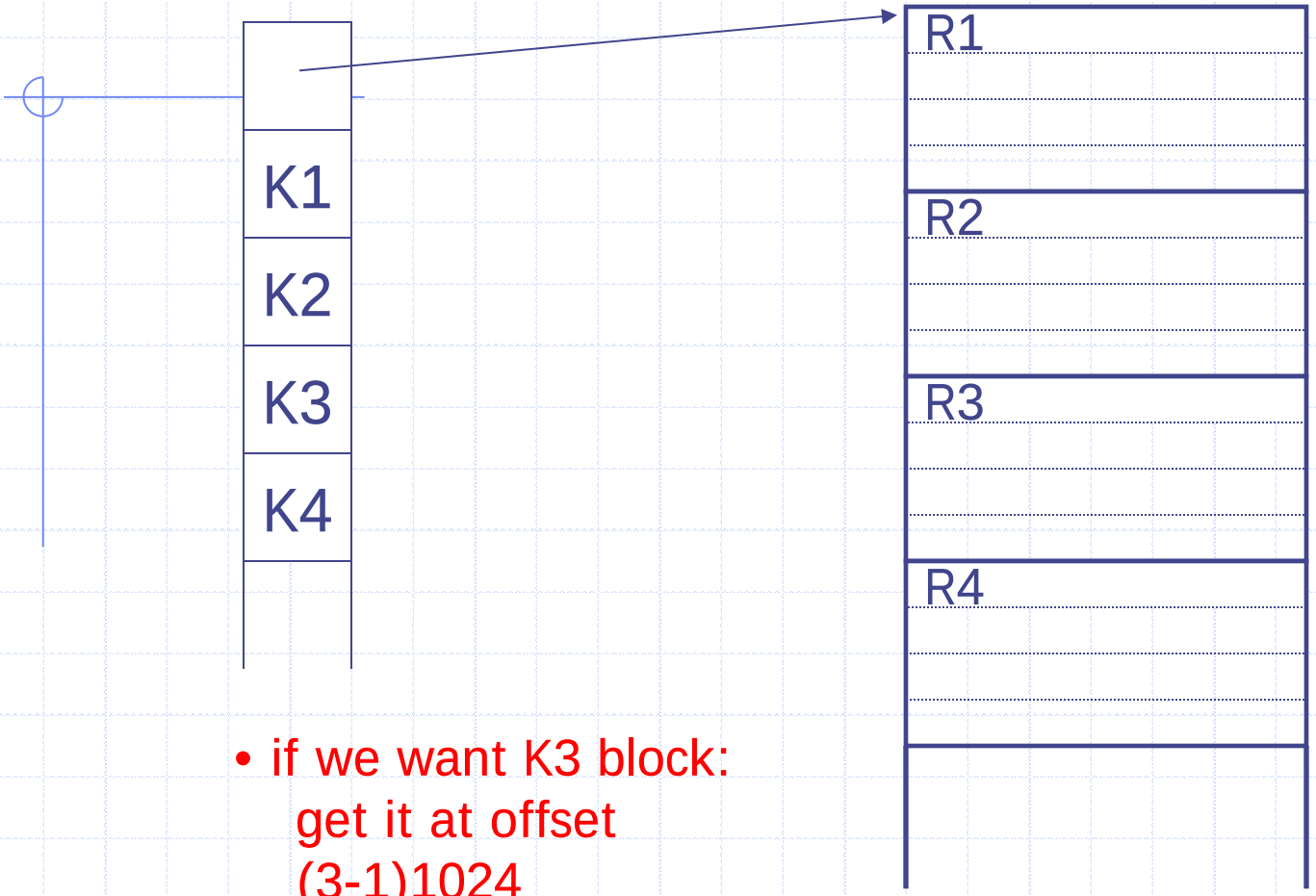
Notes on pointers:

(1) Block pointer (sparse index) can be smaller than record pointer



Notes on pointers:

(2) If file is contiguous, then we can omit pointers (i.e., compute them)



say:
1024 B
per block

- if we want K3 block:
  get it at offset
  (3-1)1024
  = 2048 bytes

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

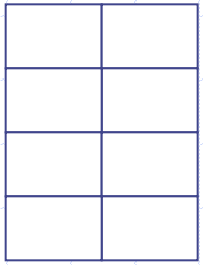
Terms

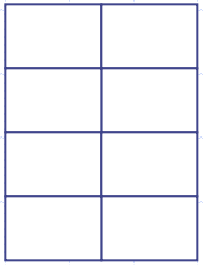
- Index sequential file
- Search key (≠ primary key)
- Primary index (on Sequencing field)主索引
- Secondary index次索引
- Dense index (all Search Key values in)
- Sparse index
- Multi-level index

Next:

- Duplicate keys
- Deletion/Insertion
- Secondary indexes

Duplicate keys





10	
10	

10	
20	

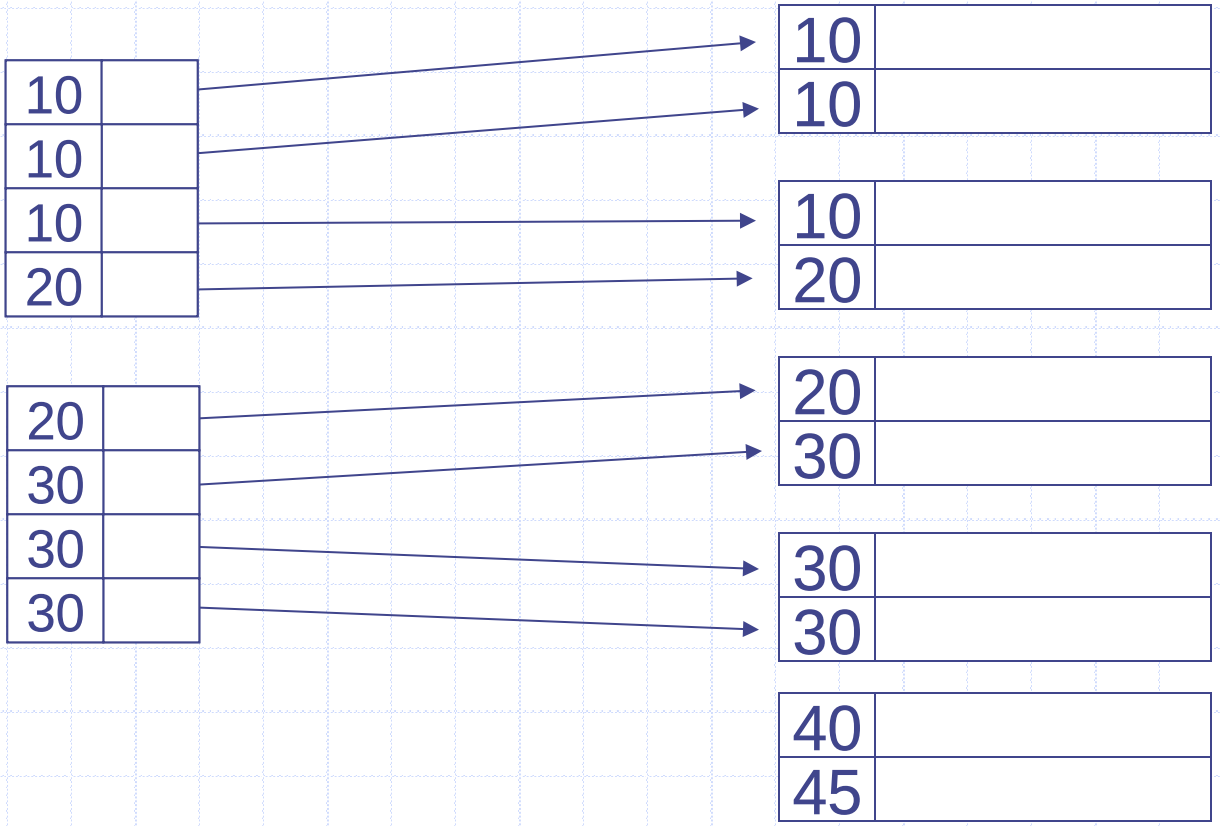
20	
30	

30	
30	

40	
45	

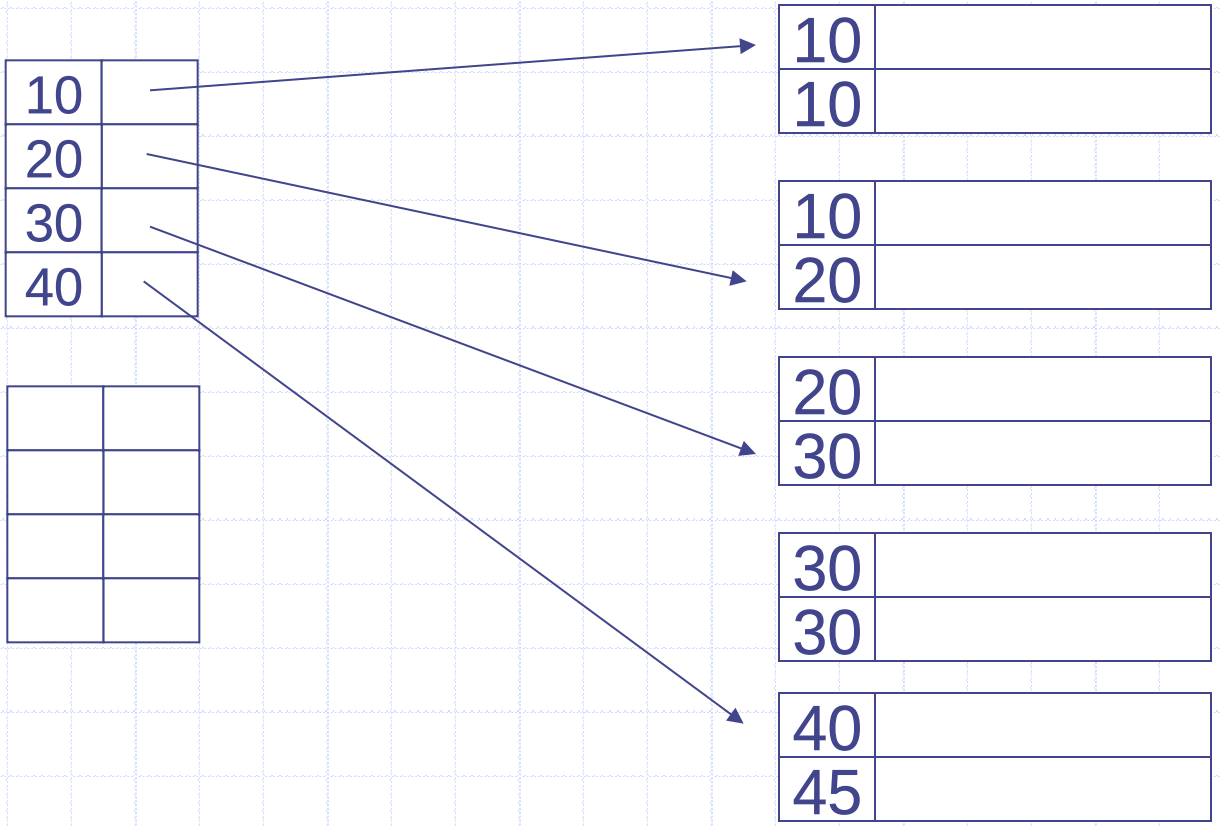
Duplicate keys

Dense index, one way to implement?



Duplicate keys

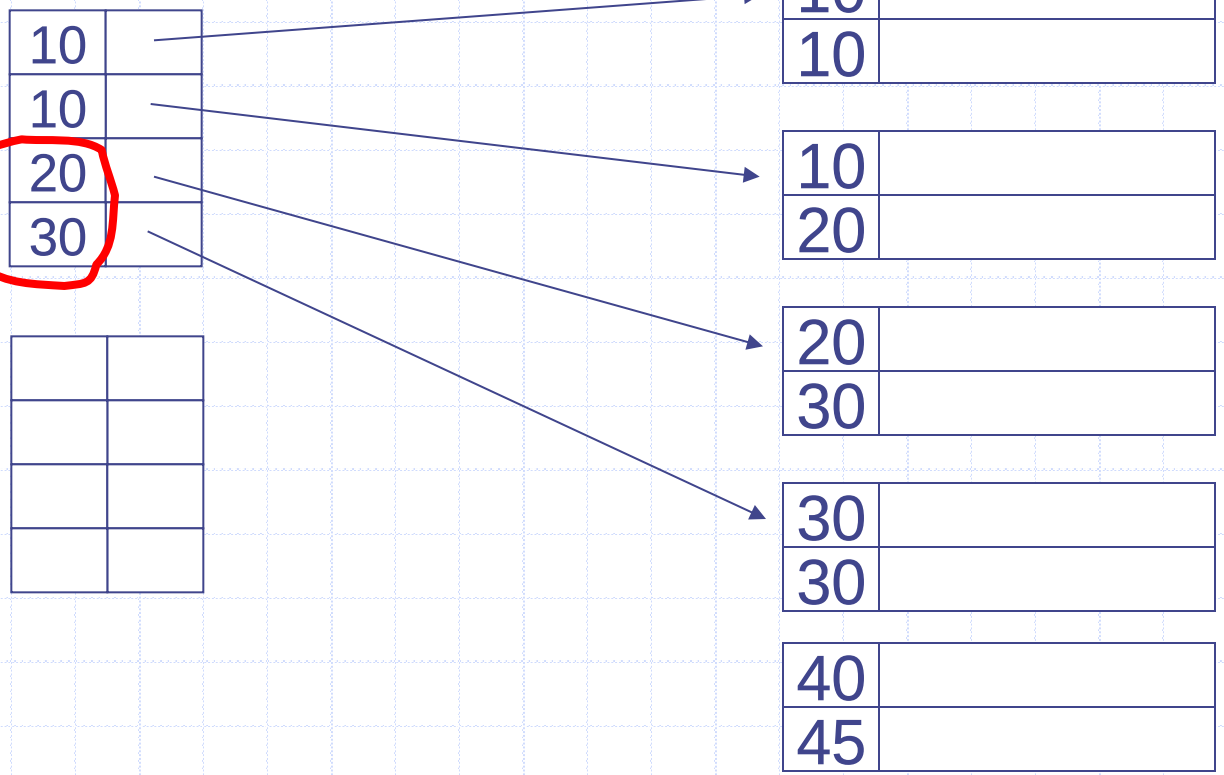
Dense index, better way?



Duplicate keys

20? Sparse index, one way?

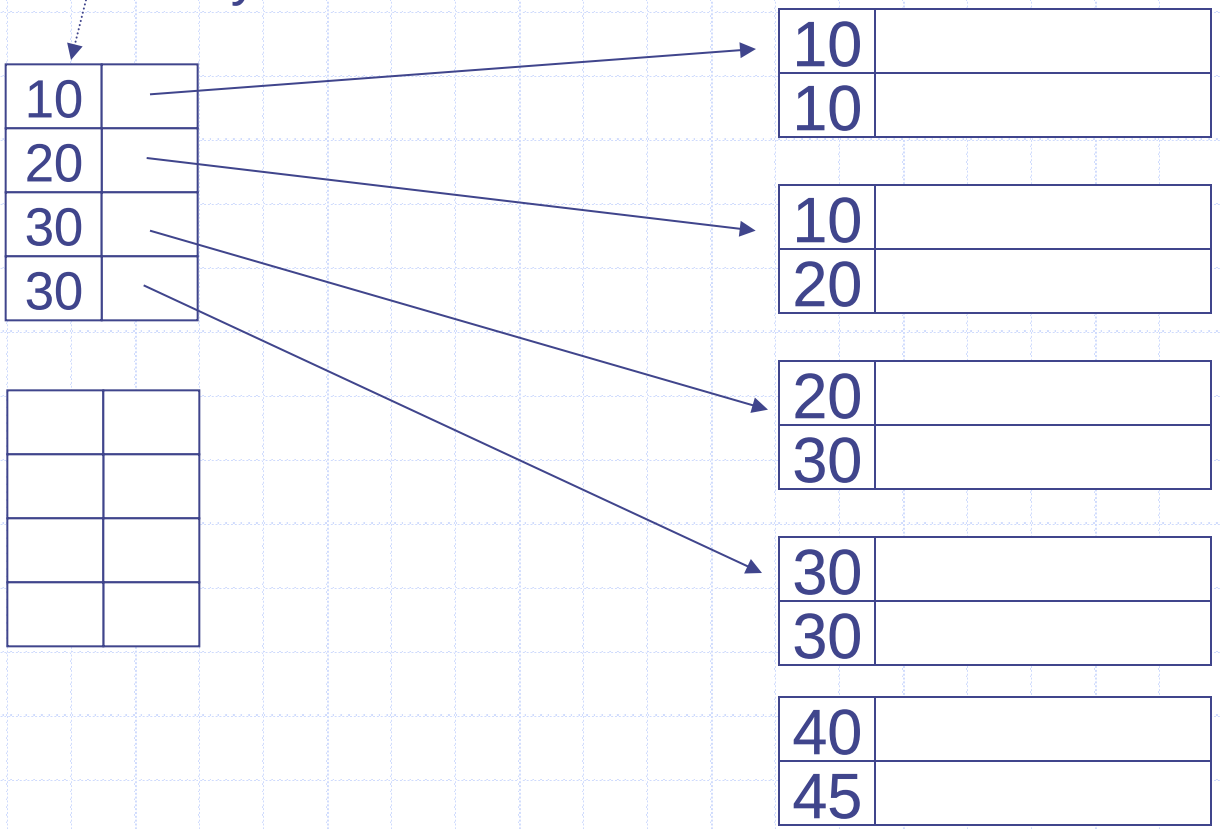
careful if looking for 20 or 30!



Duplicate keys

Sparse index, another way?

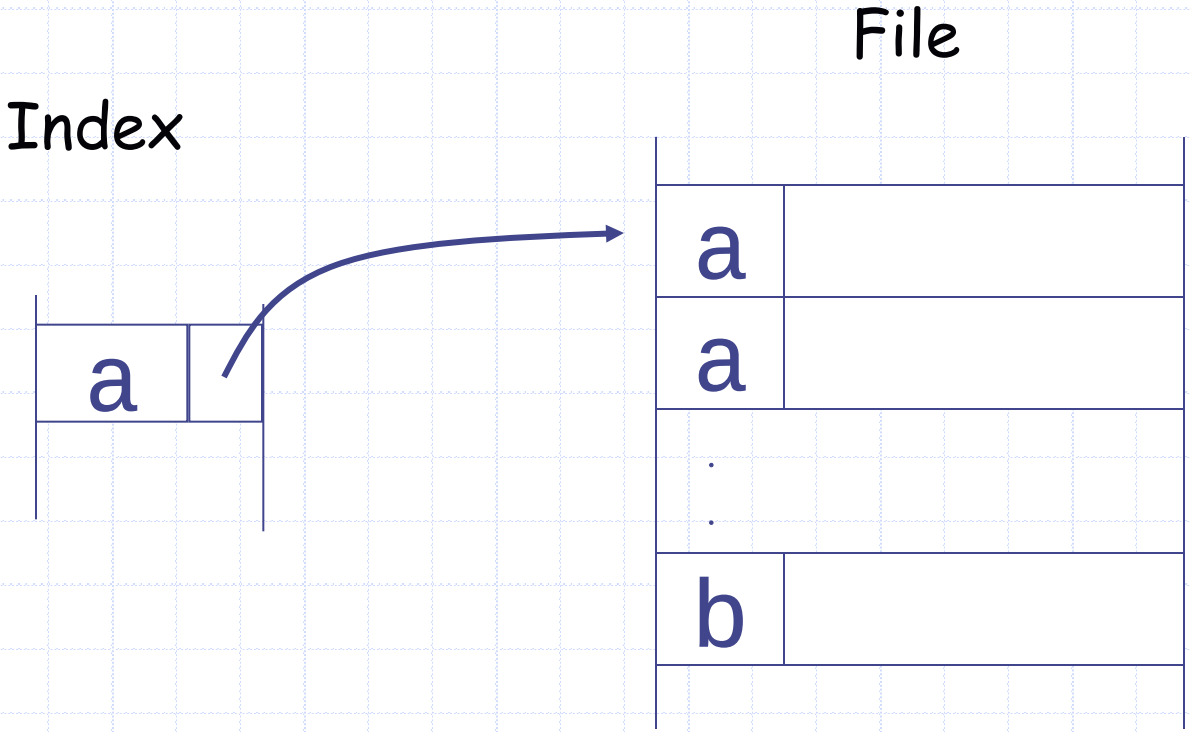
– place first new key from block



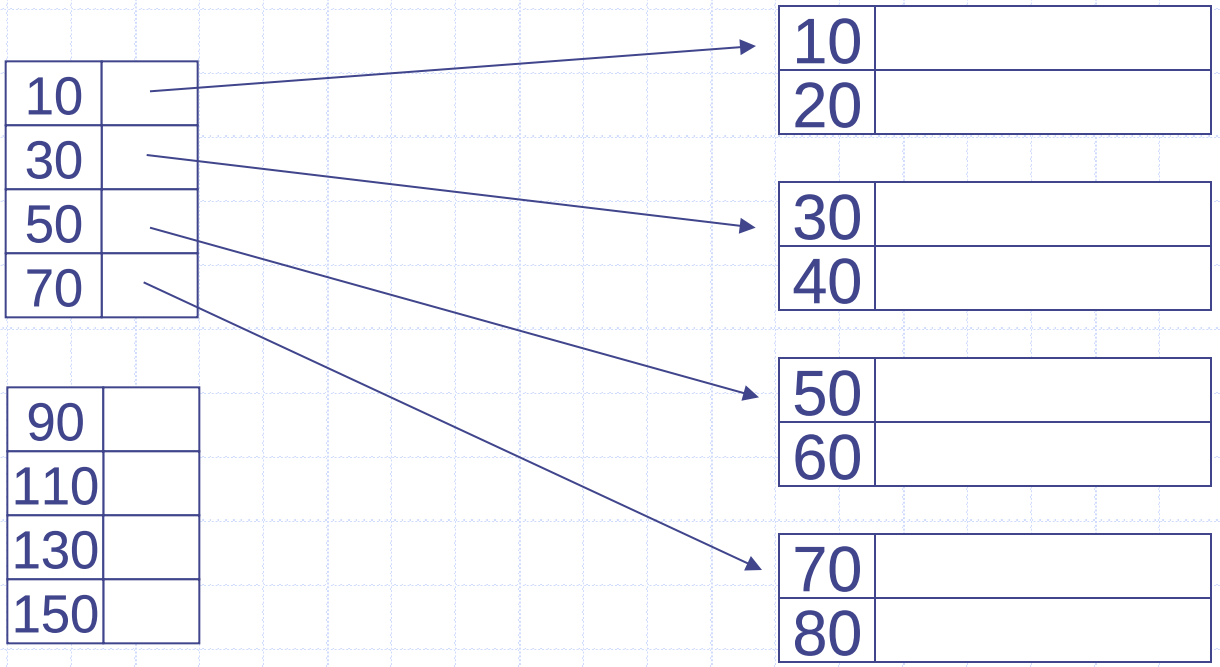
Summary

Duplicate values, primary index

- Index may point to first instance of each value only

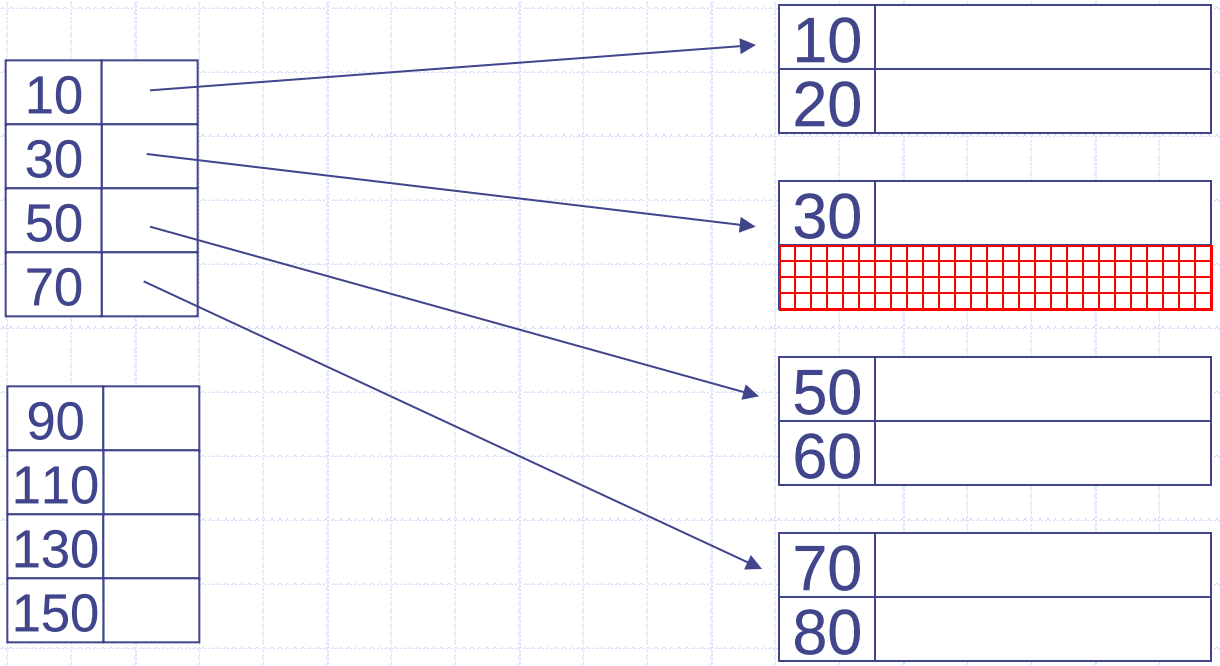


Deletion from sparse index



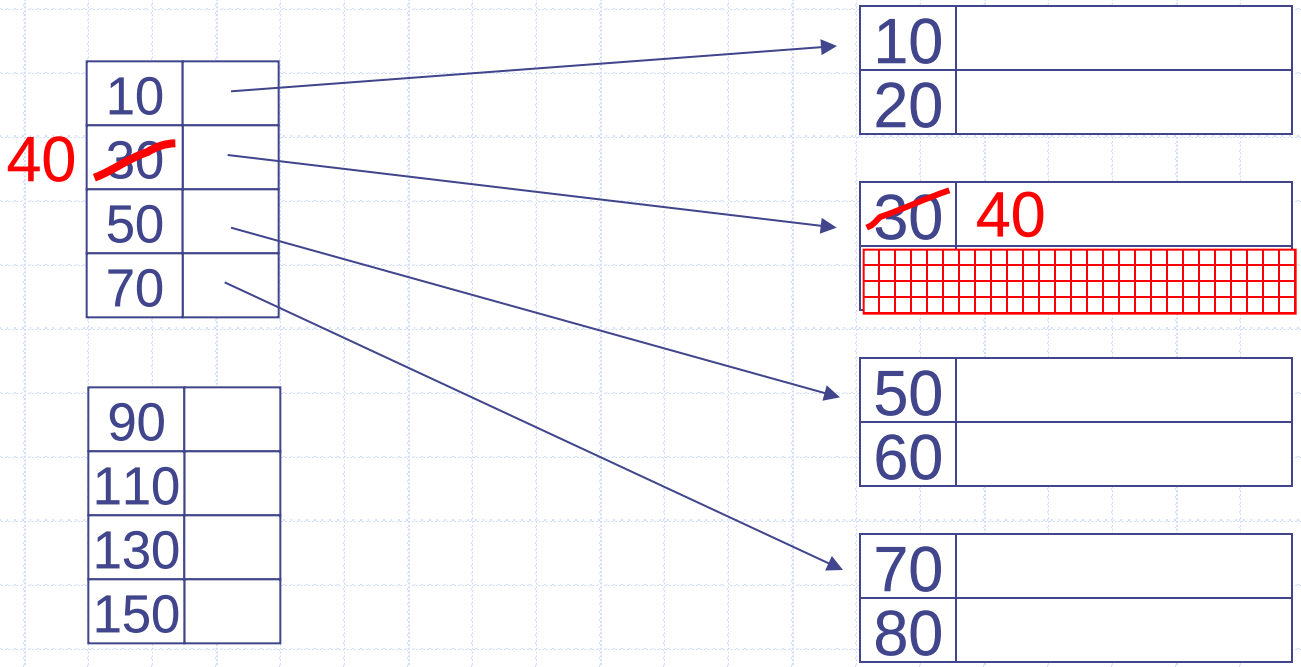
Deletion from sparse index

– delete record 40



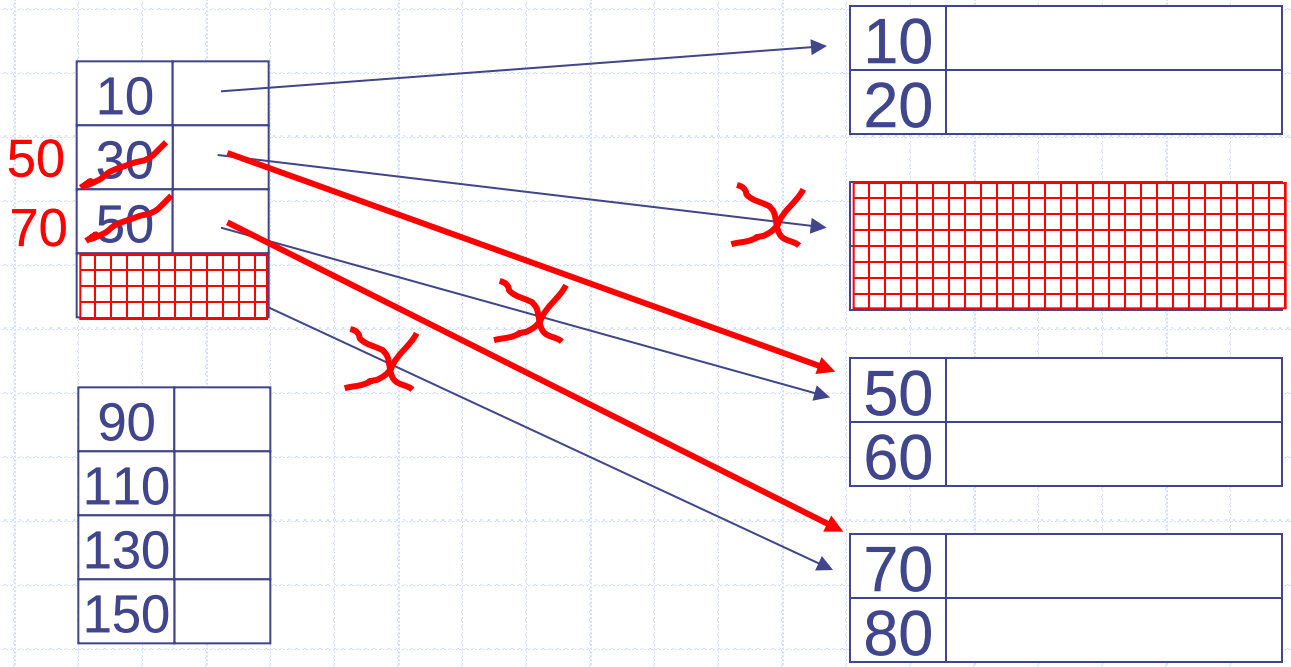
Deletion from sparse index

– delete record 30

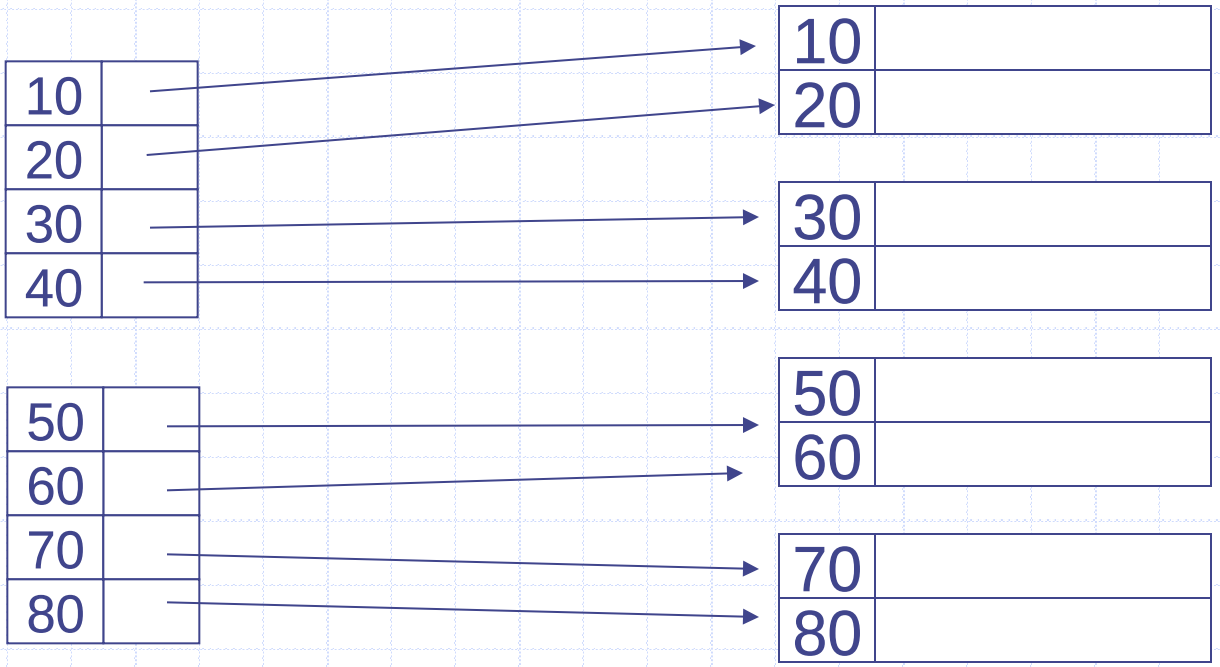


Deletion from sparse index

- delete records 30 & 40

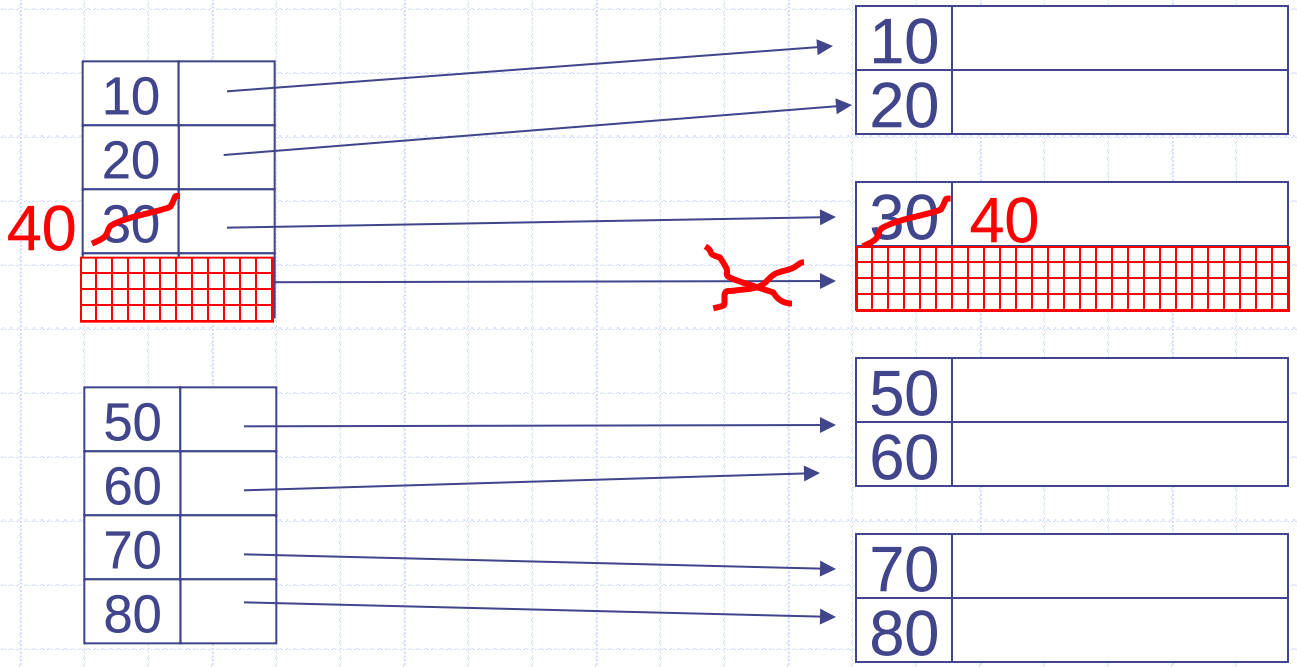


Deletion from dense index

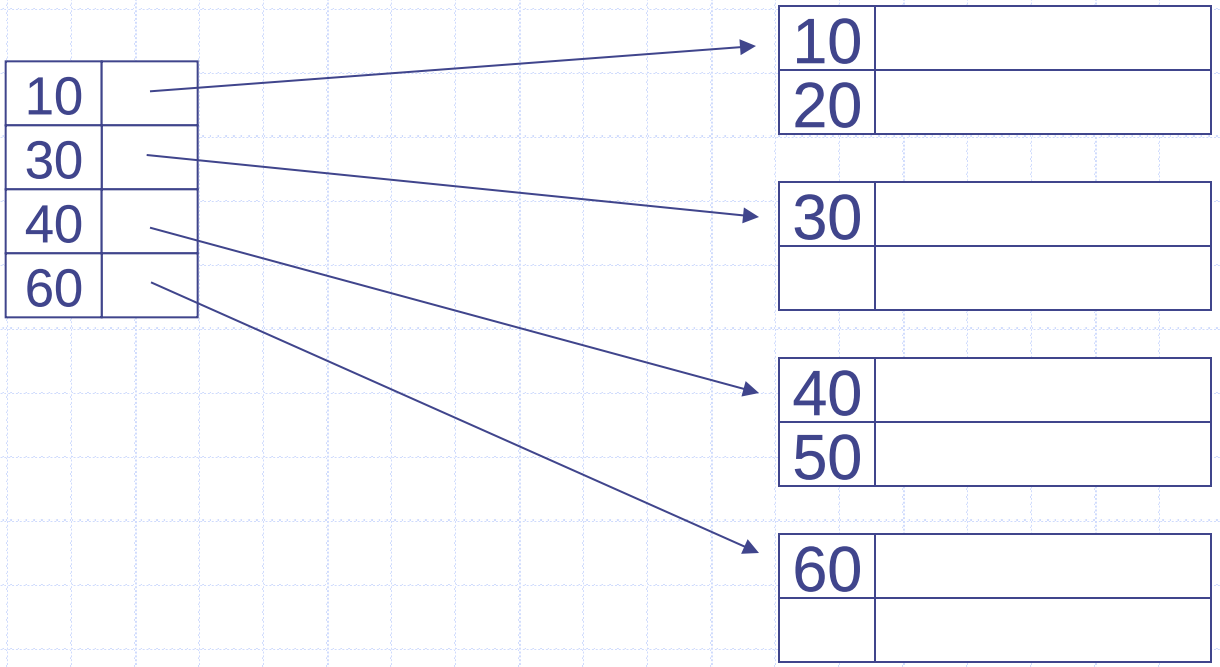


Deletion from dense index

– delete record 30

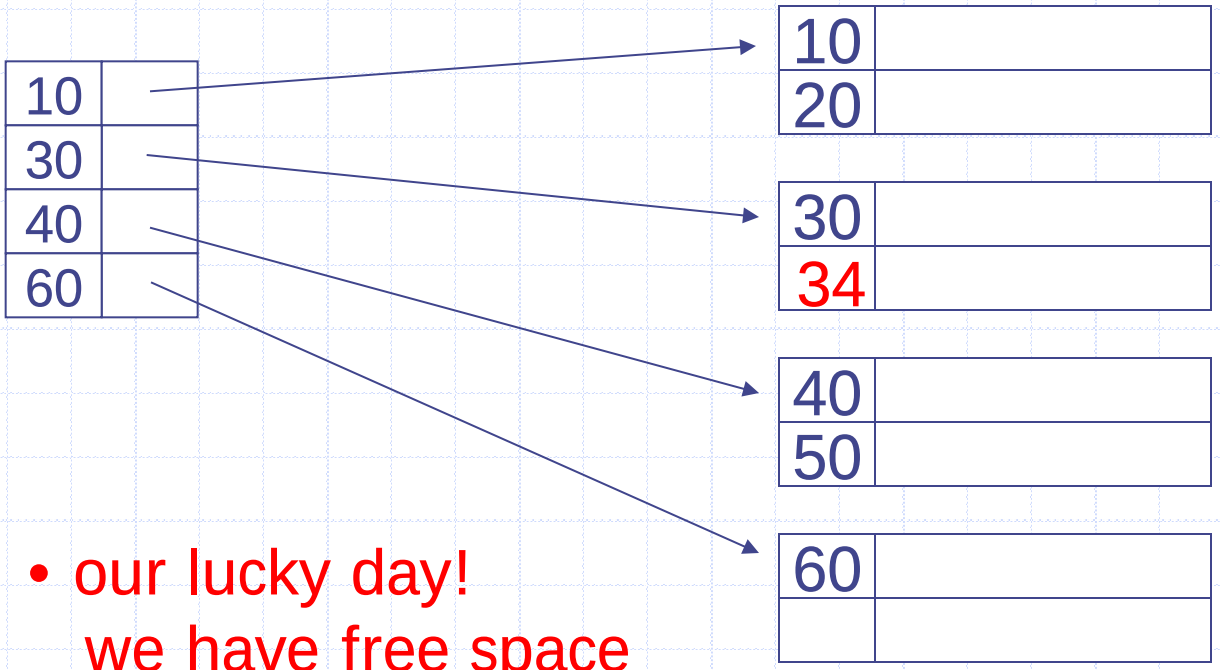


Insertion, sparse index case



Insertion, sparse index case

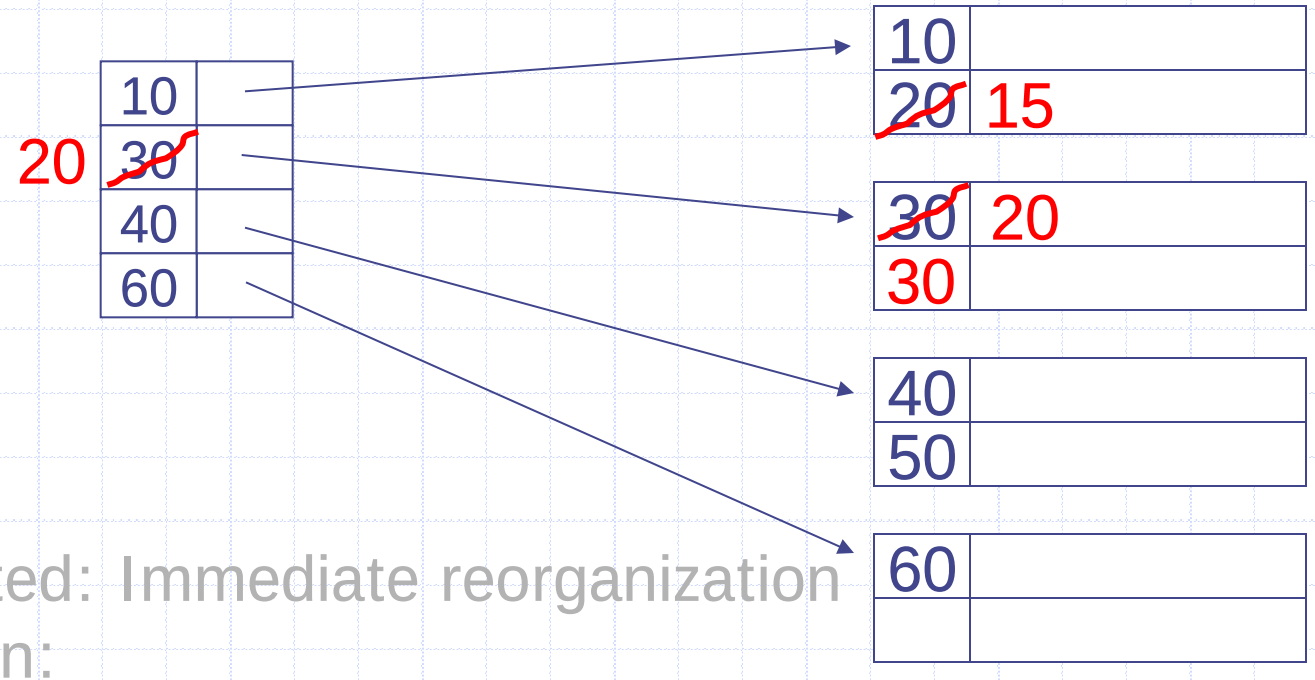
– insert record 34



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

Insertion, sparse index case

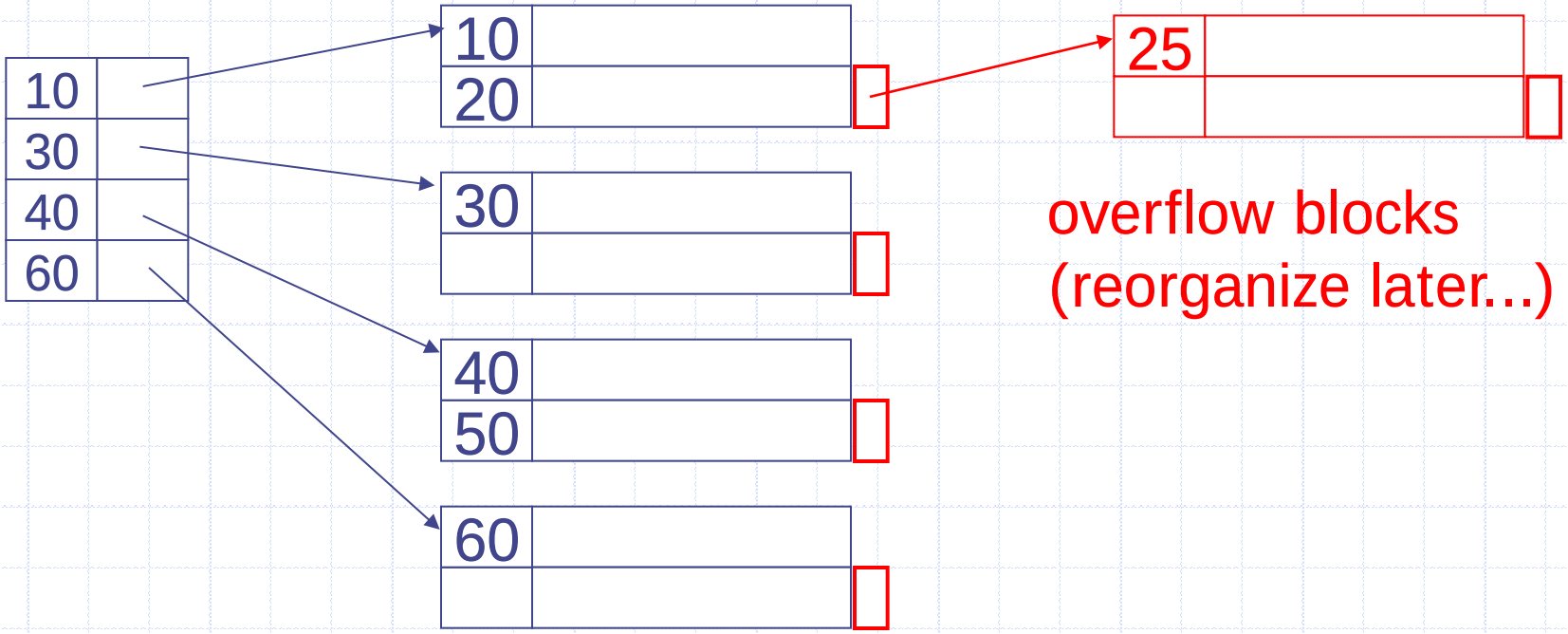
– insert record 15



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

Insertion, sparse index case

– insert record 25

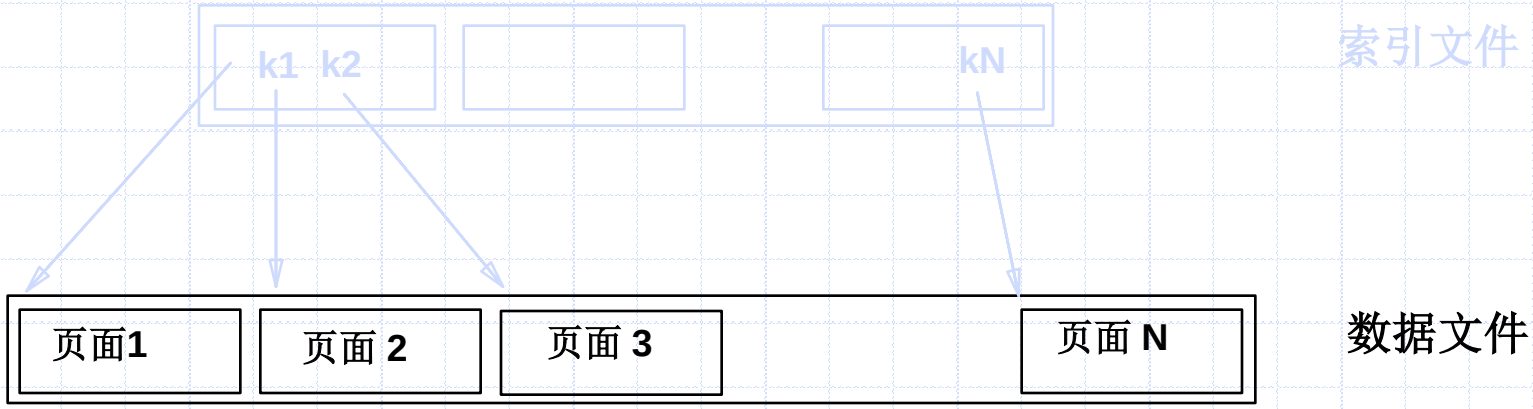


Insertion, dense index case

- Similar
- Often more expensive . . .

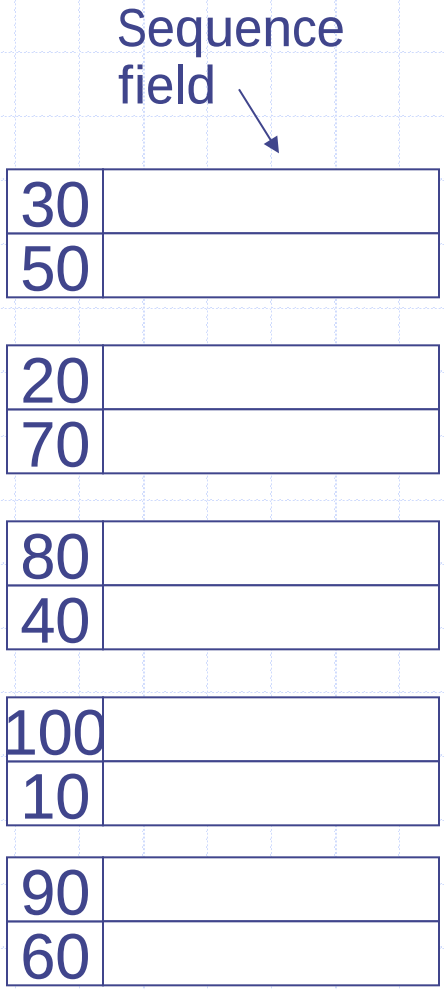
总结:
ISAM范围查找（example）

- ``查找所有成绩为优秀的学生:成绩> 90”
  - 如果数据是排好顺序的，可以进行二分查找，知道排第一的学生，进而可以扫描顺序文件，找到其它成绩优秀的学生.
  - 二分查找的代价是比较高的.如何改善？
- 简单的方法: 产生一个索引文件.



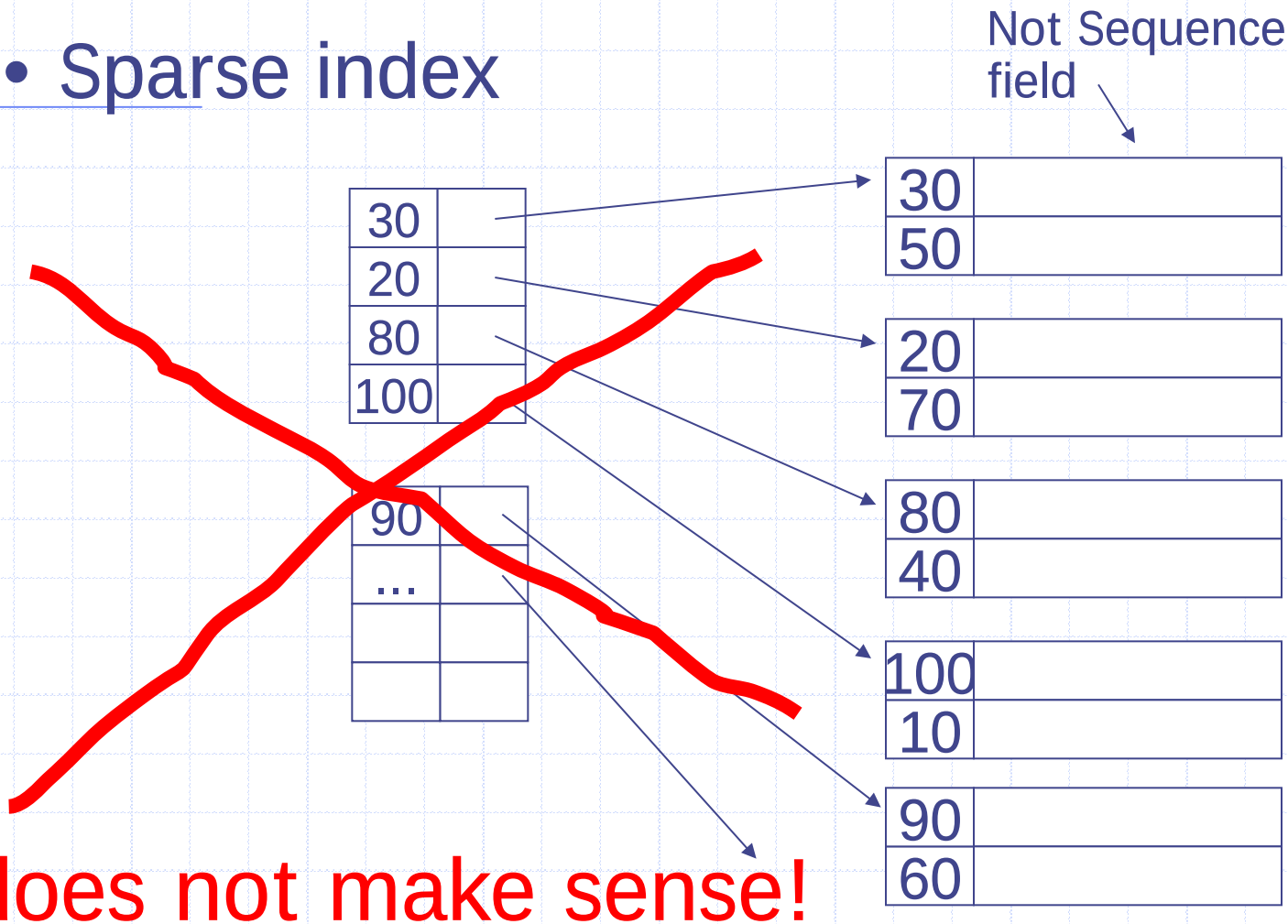
*可以在索引文件（文件较小）上进行二分查找!

Secondary indexes



Secondary indexes

- Sparse index

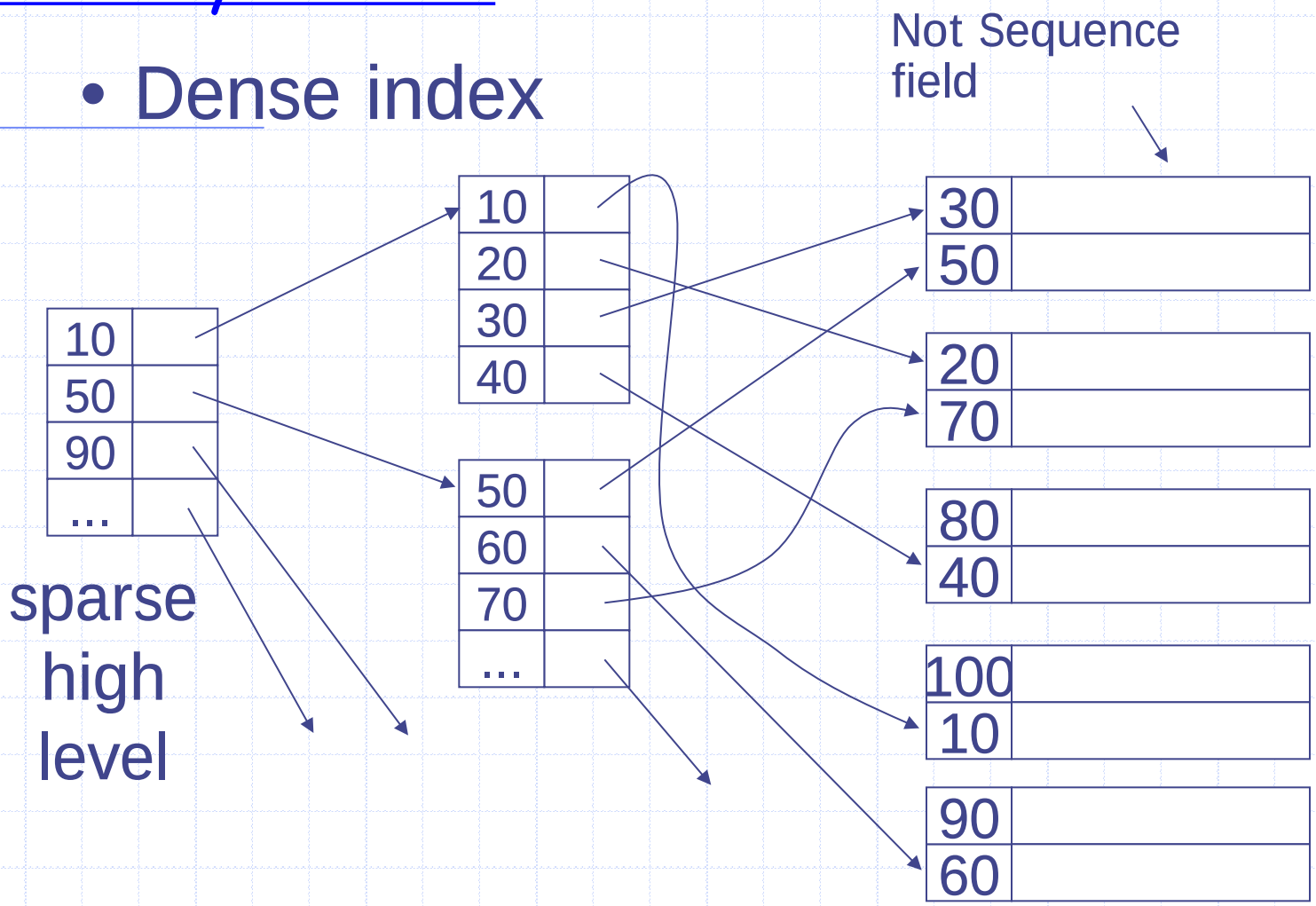


does not make sense!

How to do?

Secondary indexes

- Dense index



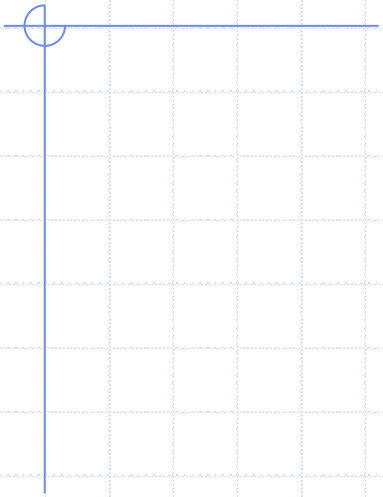
With secondary indexes:

- Lowest level is dense
- Other levels are sparse

Also: Pointers are record pointers

(not block pointers;)

Duplicate values & secondary indexes



20	
10	

20	
40	

10	
40	

10	
40	

30	
40	

Duplicate values & secondary indexes

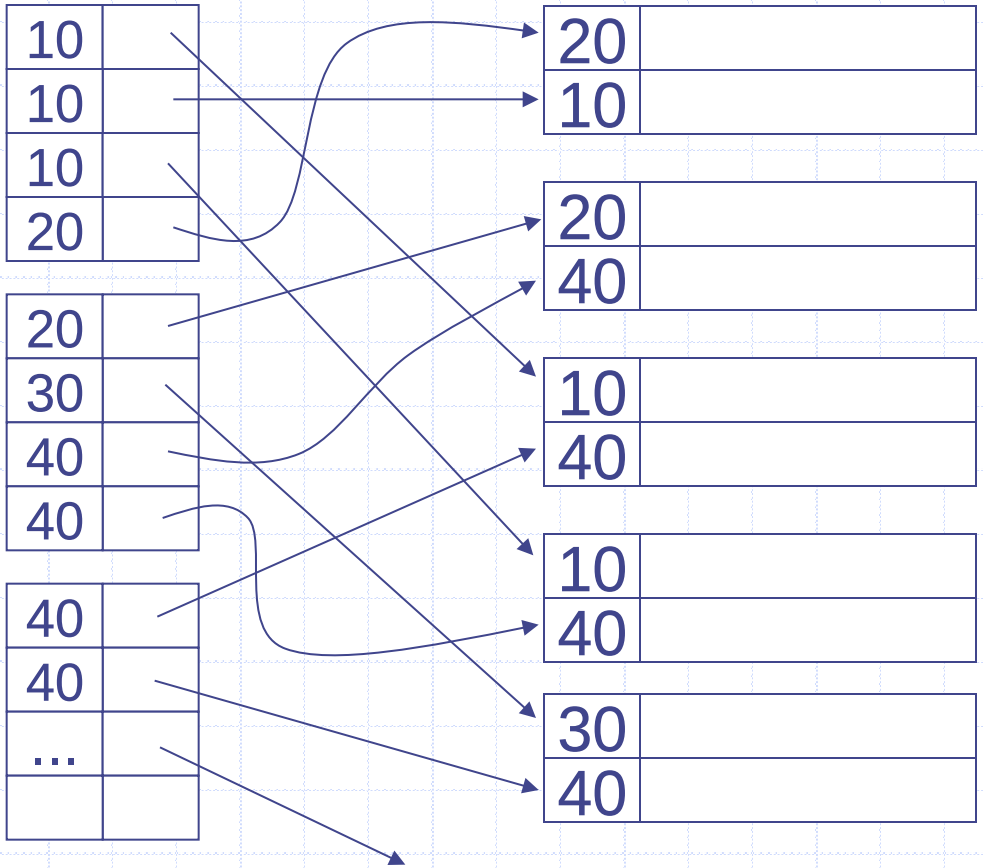
one option...

Problem:

excess overhead!

- disk space
- search time

- Another methods?

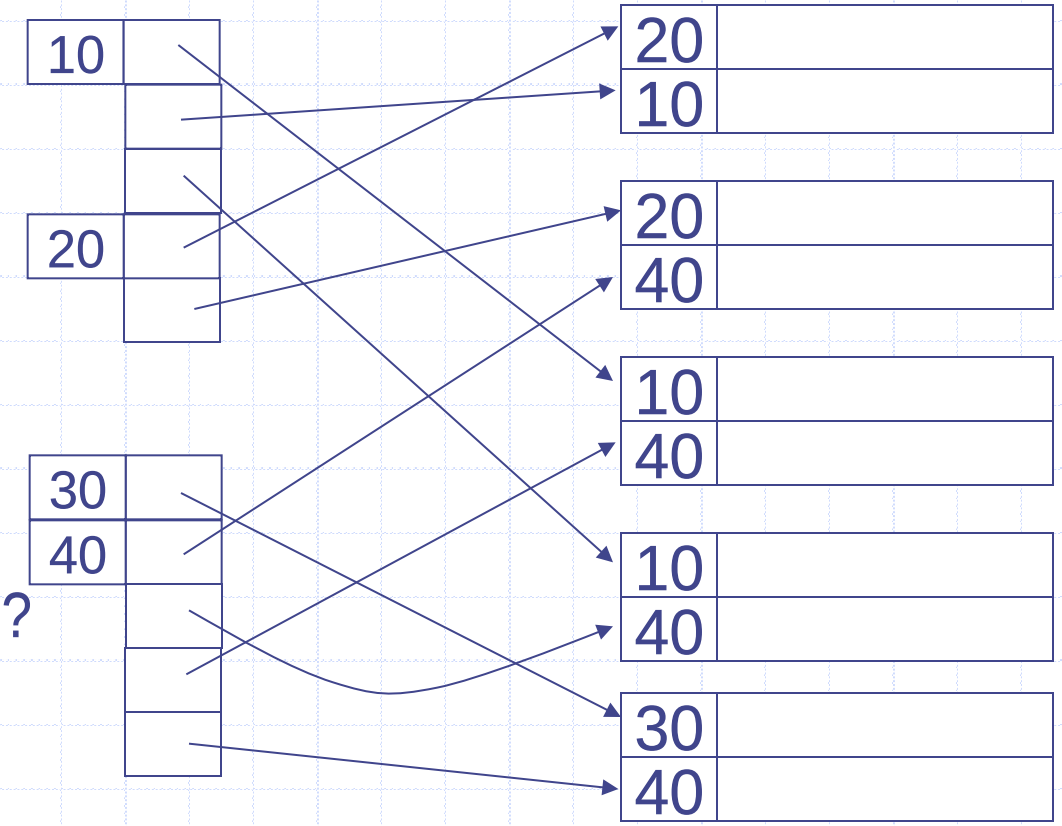


Duplicate values & secondary indexes

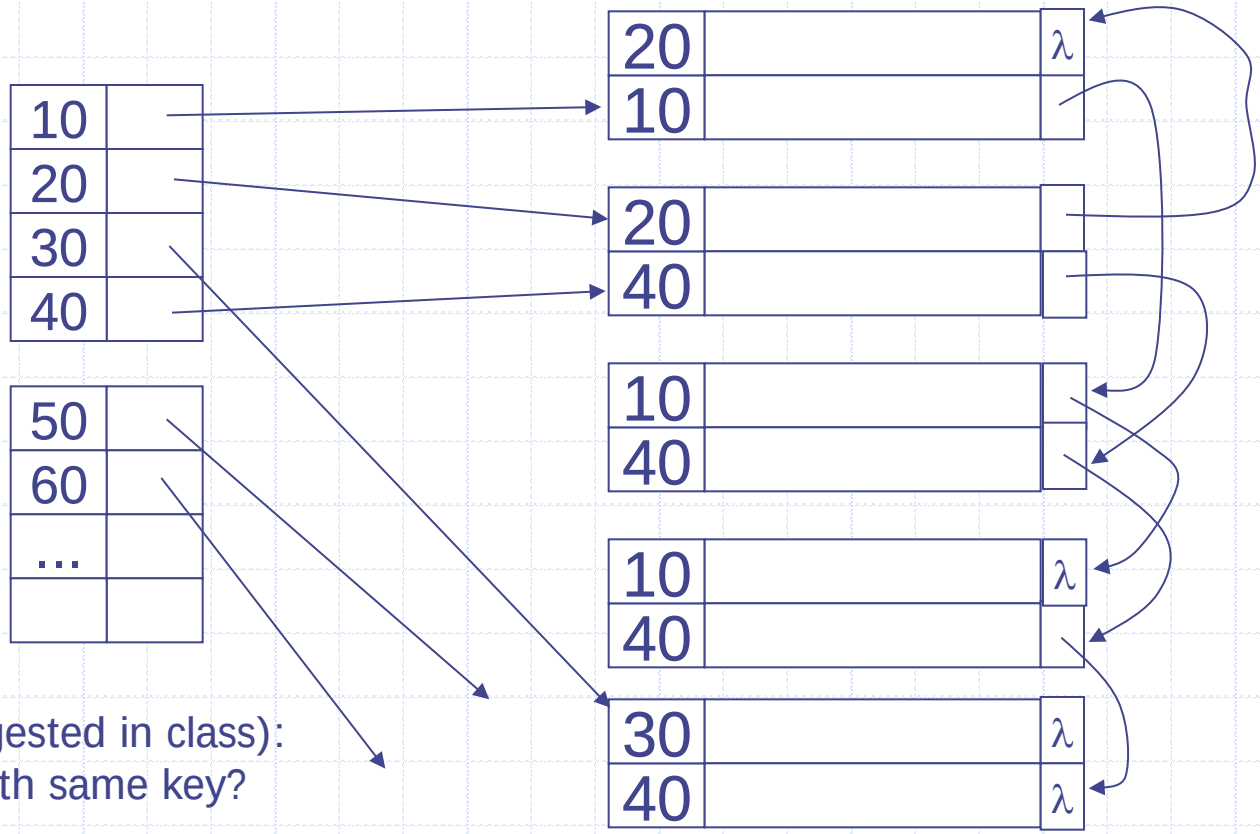
another option...

Problem:
variable size records in index!

- Another methods?



Duplicate values & secondary indexes

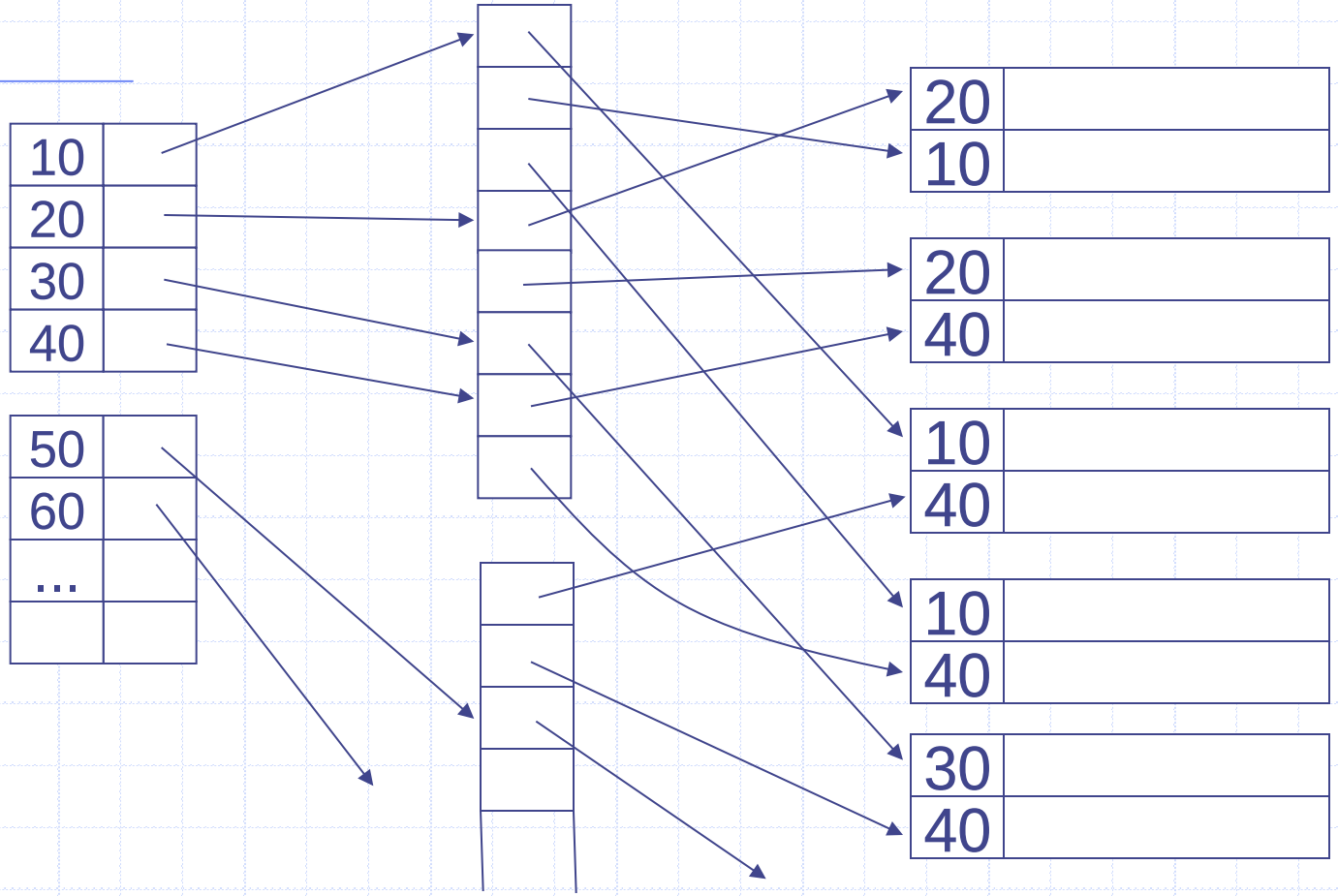


Another idea (suggested in class):
Chain records with same key?

Problems:

- Need to add fields to records
- Need to follow chain to know records
- Another methods?

Duplicate values & secondary indexes



Buckets

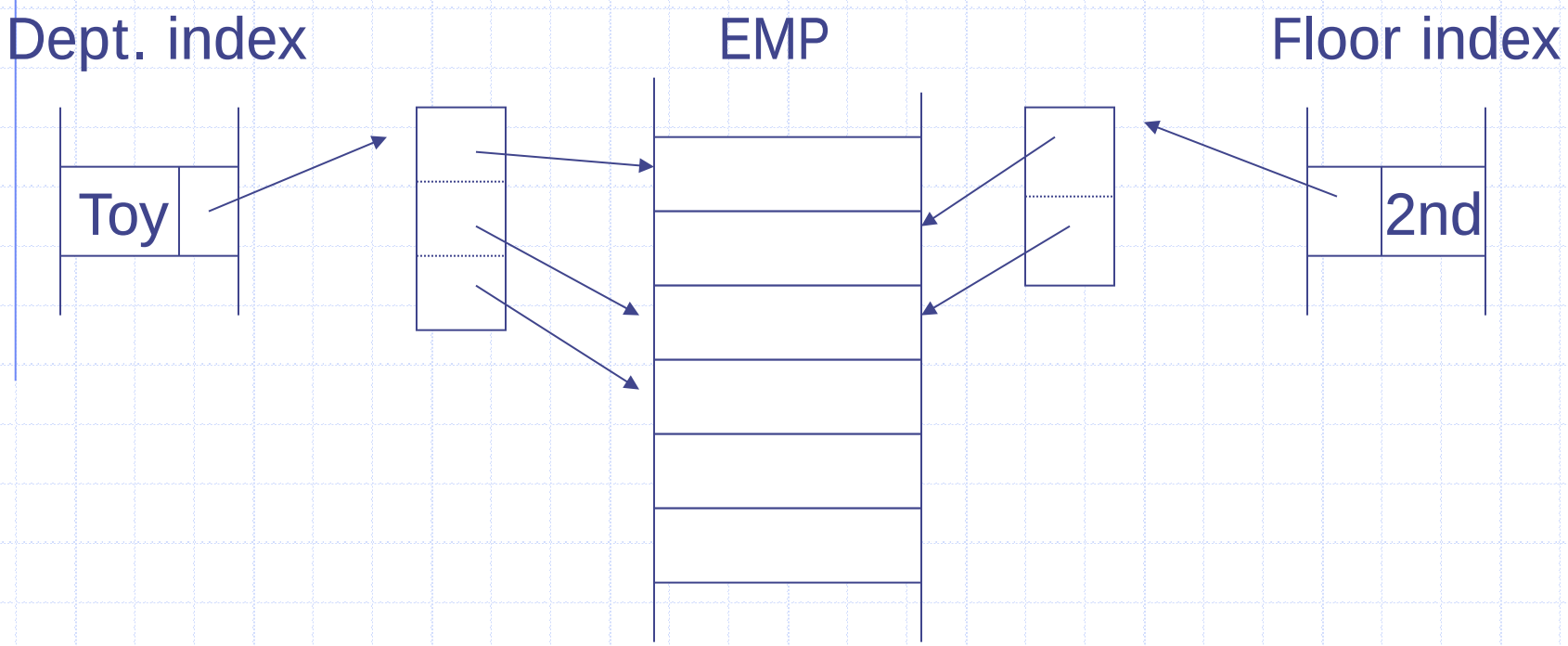
(similar to hash)

Advantage?

Why "bucket" idea is useful

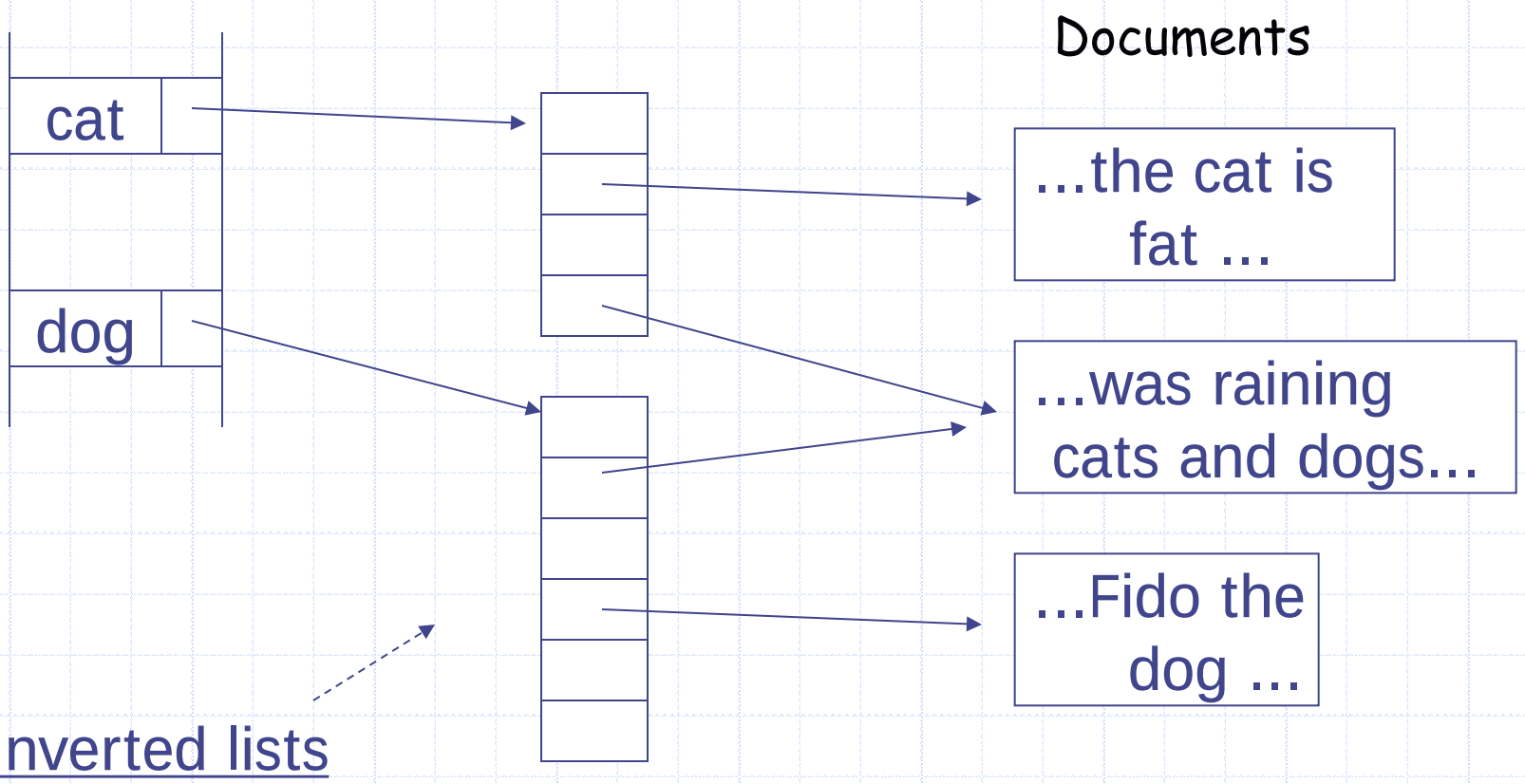
Indexes	Records
Name: primary	EMP (name,dept,floor,...)
Dept: secondary	
Floor: secondary	

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



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

This idea used in text information retrieval



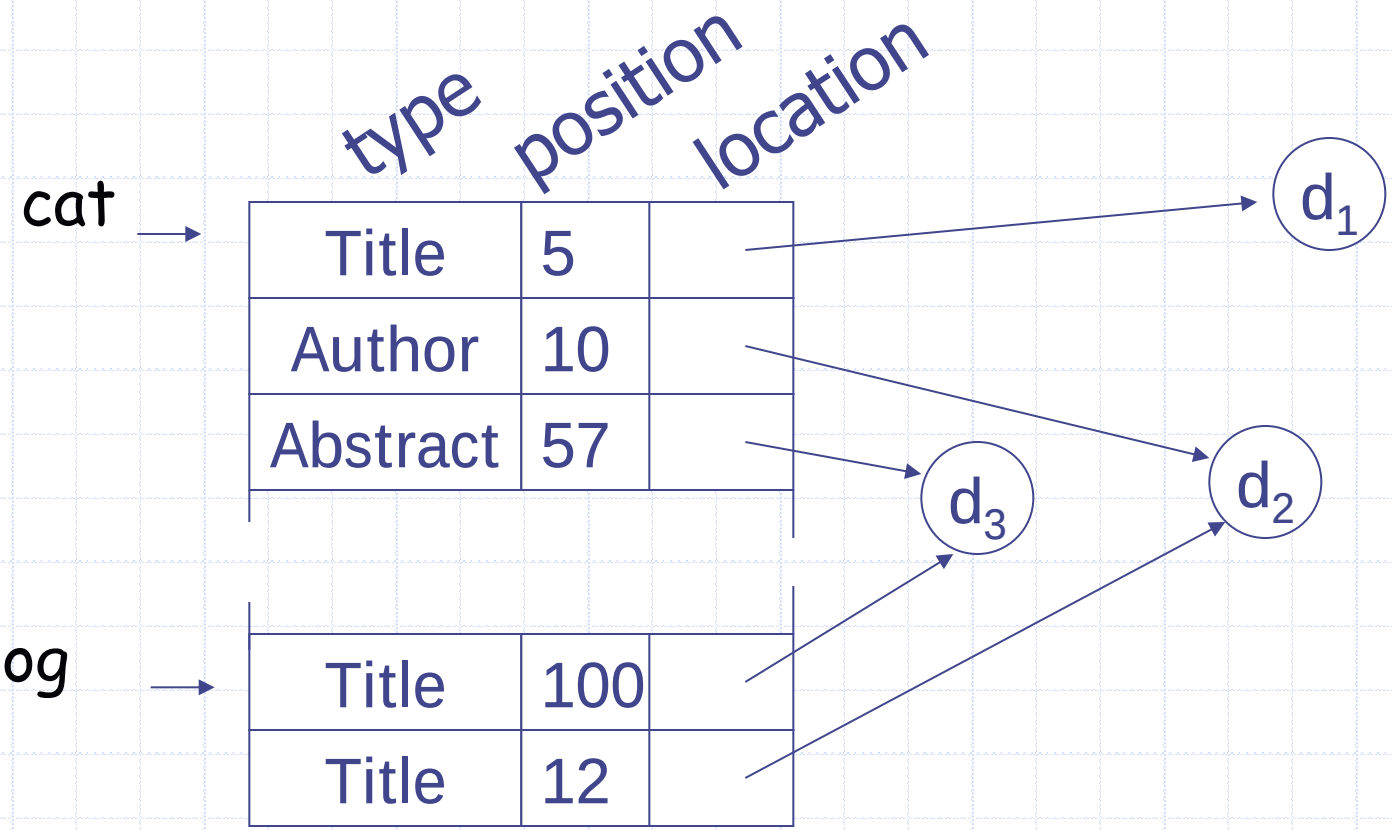
IR QUERIES

- 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

Common technique:

more info in inverted list



IR DISCUSSION

- Stop words
- Truncation
- Thesaurus
- Full text vs. Abstracts
- Vector model

Vector space model

	w1	w2	w3	w4	w5	w6	w7 ...
DOC =	<1	0	0	1	1	0	0 ...>
Query=	<0	0	1	1	0	0	0 ...>

PRODUCT = 1 + ... = score

- Tricks to weigh scores + normalize

e.g.: Match on common word not as useful as match on rare words...

why?

(rare words:具有区分能力的词)

(common word: this, that, it, is, the, an,...)

- Try google
- Try Altavista, Excite, Infoseek, Lycos...

Summary so far

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

Conventional indexes

Advantage:

- Simple
- Index is sequential file
  good for scans

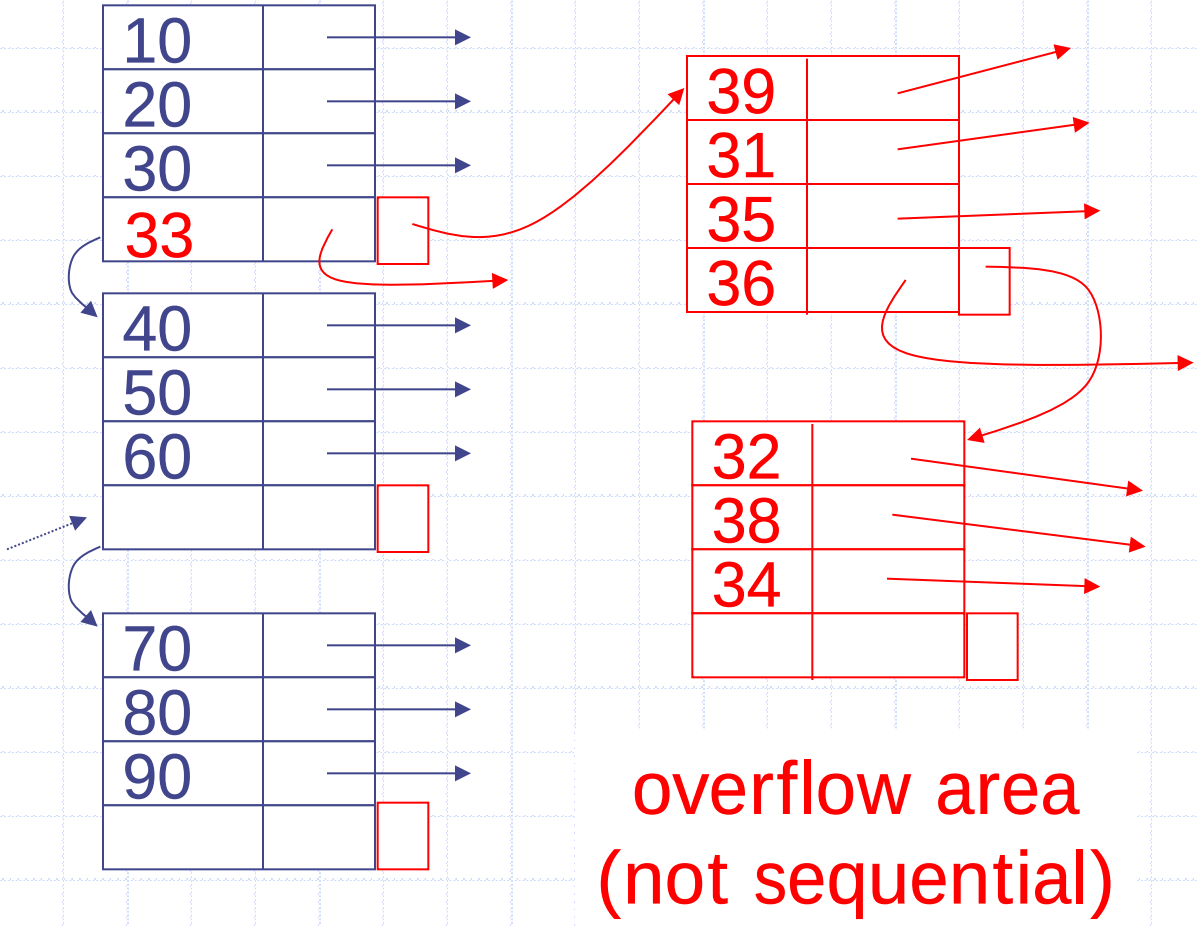
Disadvantage:

- Inserts expensive and/or
- Lose sequentiality & balance
  (why?)

Example

Index (sequential)

continuous



New index?

Outline:

- Conventional indexes
- B-Trees (balanced multiway tree) ⇒ NEXT
- Hashing & Multi-dimentional Indexes

- NEXT: Another type of index
  - Give up on sequentiality of index
  - Try to get "balance"

Comparison: B-trees vs. static indexed sequential file

Ref #1: Held & Stonebraker

"B-Trees Re-examined"

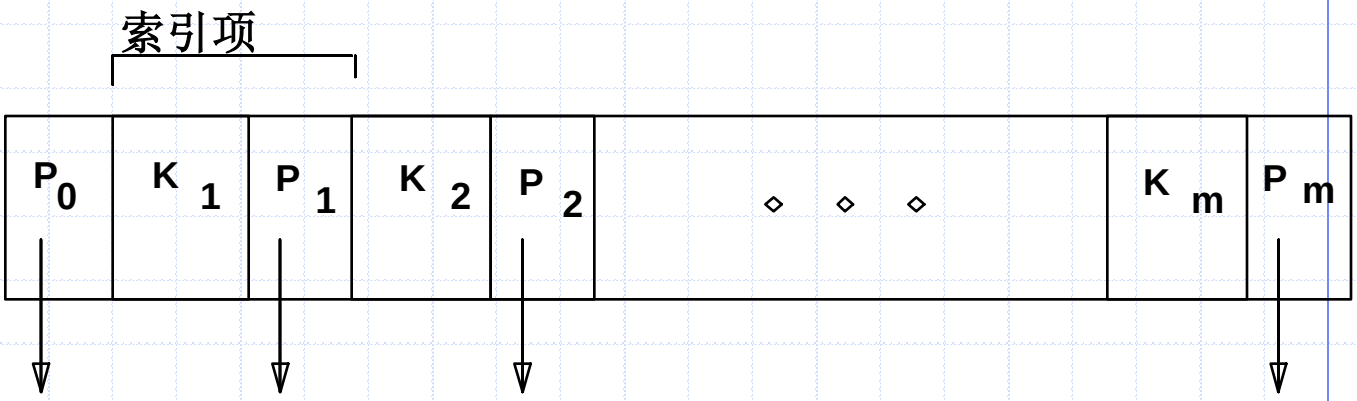
CACM, Feb. 1978

Ref #2: M. Stonebraker,

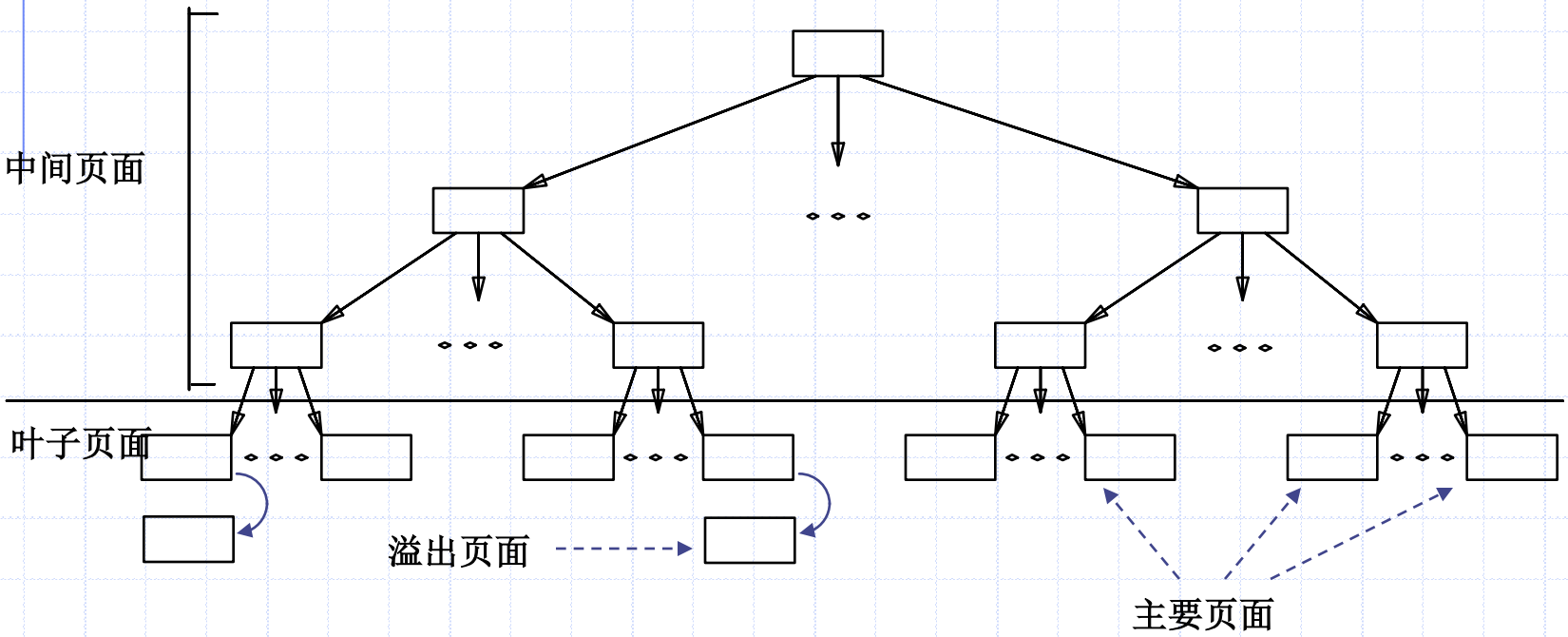
"Retrospective on a database system," TODS, June 1980

B-trees better!!

ISAM

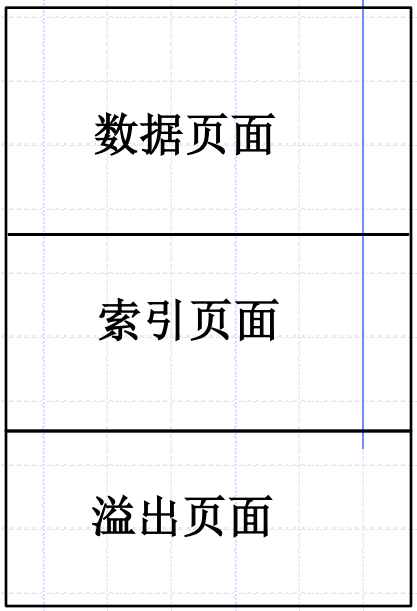


- 索引文件可以仍然很大，可以重复这种方法!



** 叶子页面中包括查找数据的入口*

注意问题

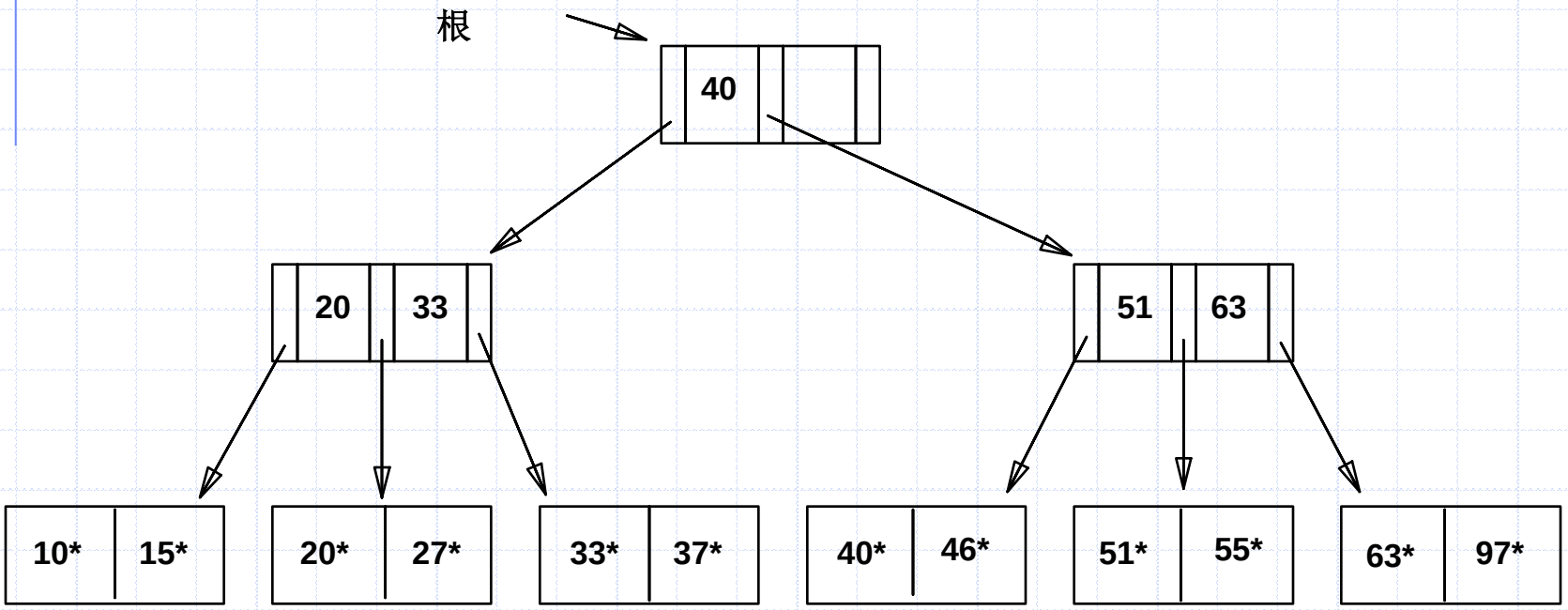


- *创建文件*：叶子页（存放数据）根据查找关键字顺序分配，然后分配索引页，最后分配溢出页.
- *索引入口*：<搜索码值, 页面表示符 id>；指向数据页
- *查找*：从根节点开始，通过匹配，找到叶子。代价是$\log_F N$；F = # 入口数/每个页面, N = # 叶子页的数量
- *插入*：找到叶子页面应该在的位置，插入进索引文件.
- *删除*：首先找到需要移走的叶子，再删除

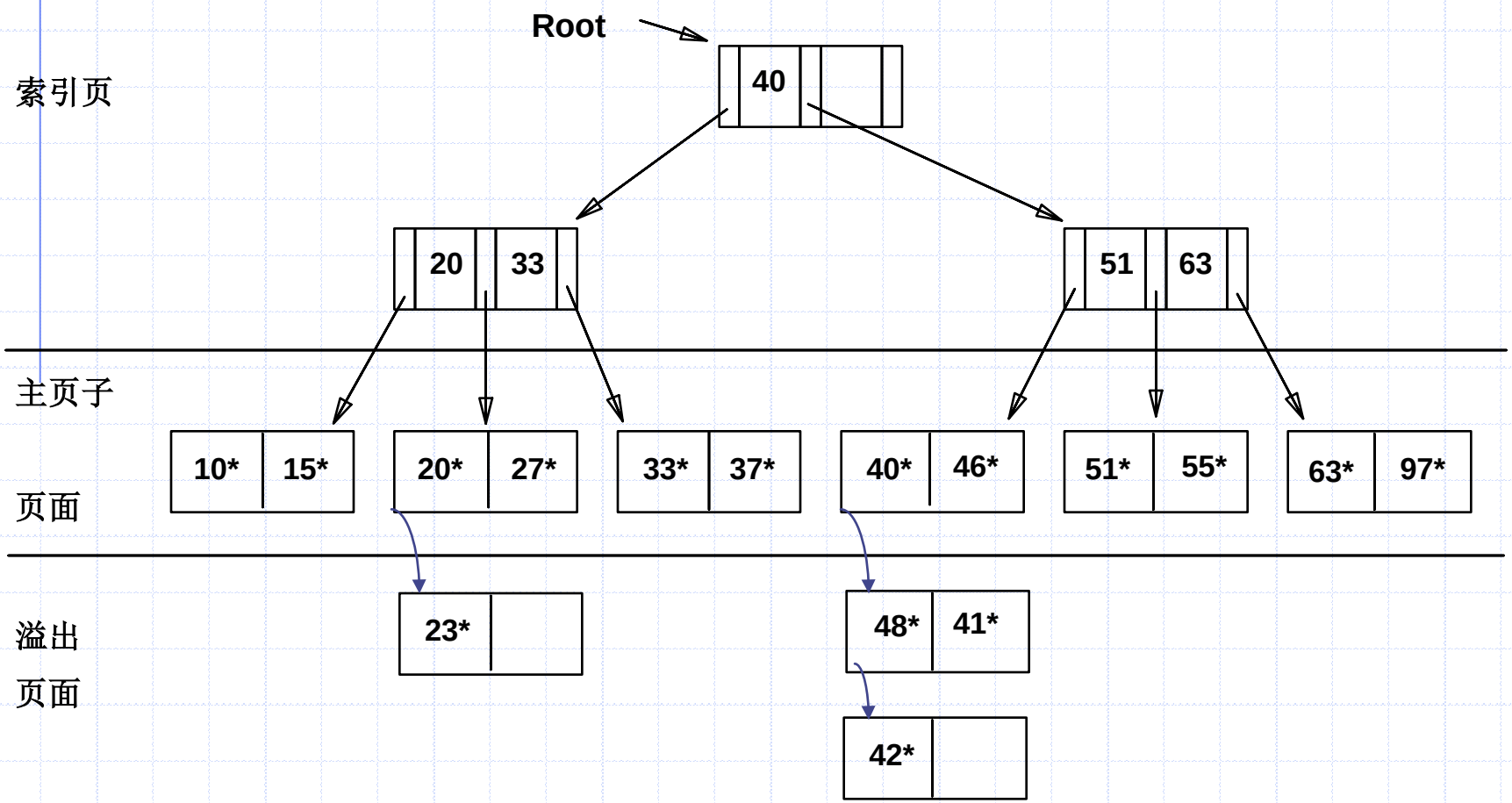
*静态树结构：插入和删除只影响叶子页.

ISAM 索引树举例

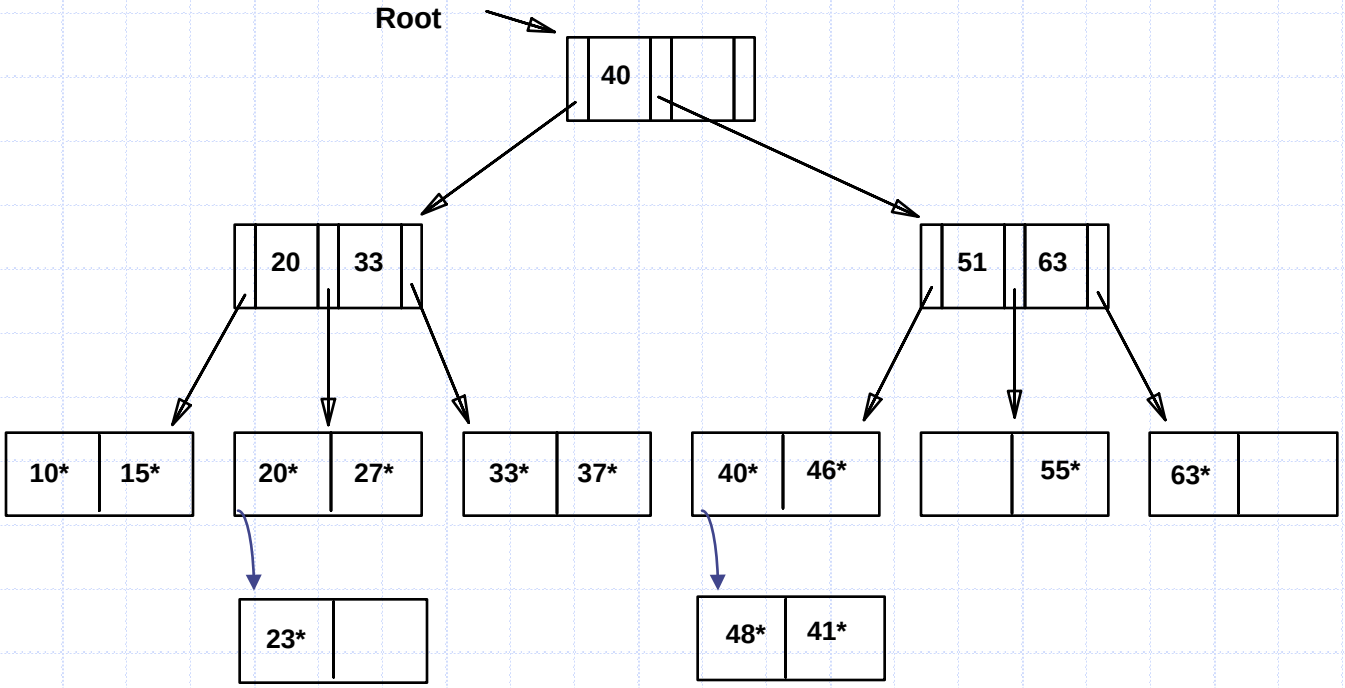
- 每个节点包含两个入口



插入23*, 48*, 41*, 42* 之后...



… 删除 42*, 51*, 97*之后

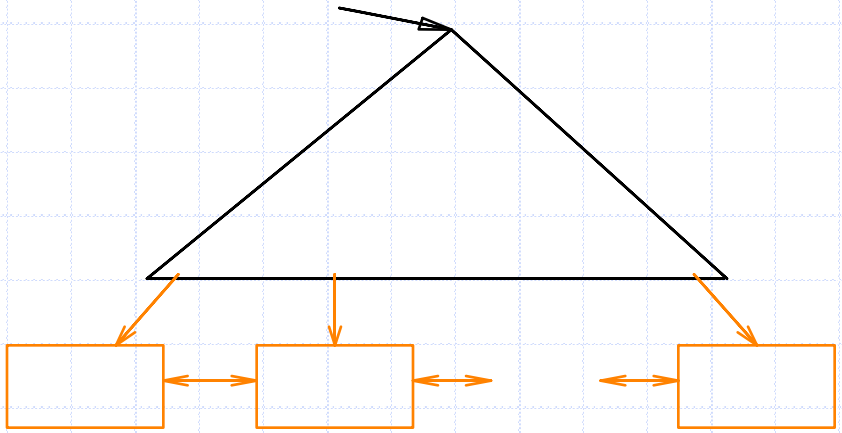


** 注意：虽然51* 出现在索引树中，但是，叶子页面却被删除了!*

动态索引的动机

B+ 树：使用最广泛的索引

- 插入/删除 代价为$\log_F N$; 能保持高度平衡 (F = 扇出数, N = #叶子页面数量)
- 除了根节点，每个节点都能保证最小 50%的占有率．每个节点包含**d** <= *m* <= **2d** 目录项．参数**d** 称为树的秩.
- 有效地支持相等值的查找和范围查找.

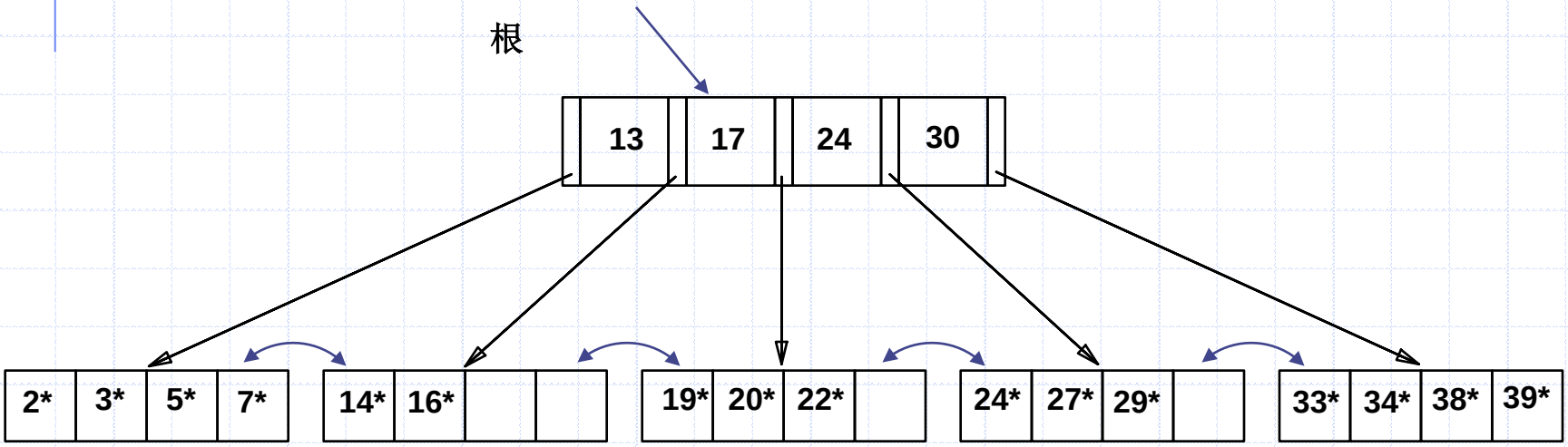


索引目录
(直接查找)

数据目录
("顺序查找")

B+ 树举例

- 从根开始查找，通过比较可以定位到叶子 (与ISAM方法技术相同).
- 查找 5*, 15*,...



实际使用中的B+ 树

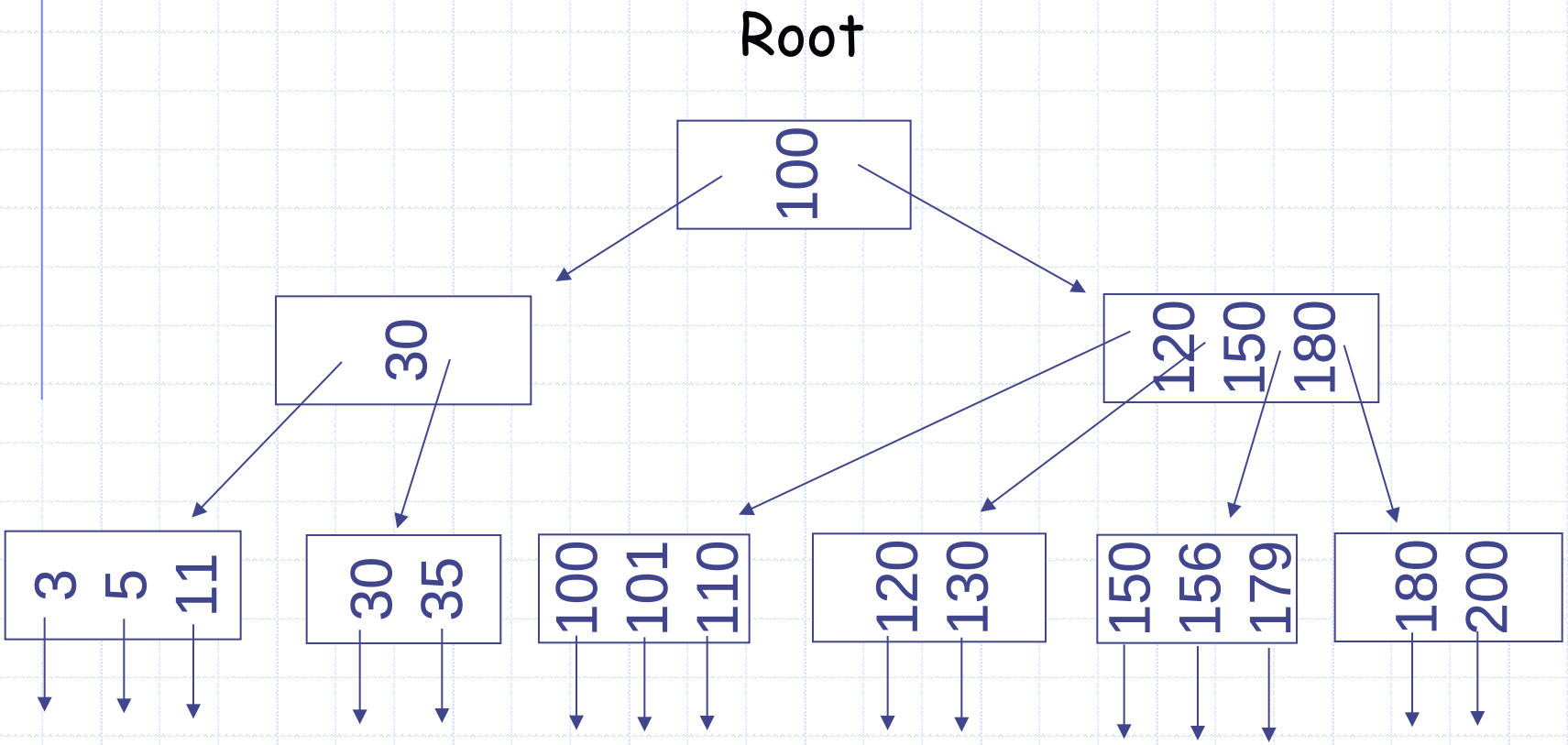
- 典型的秩为: 100. 每个节点典型的占有率为: 67%.
  - 平均扇出数为 = 133
- 典型的容量:
  - 高度为 4时: 133^4 = 312,900,700个记录
  - 高度为 3时: 133^3 = 2,352,637个记录

插入一个数据目录到 B+ 树

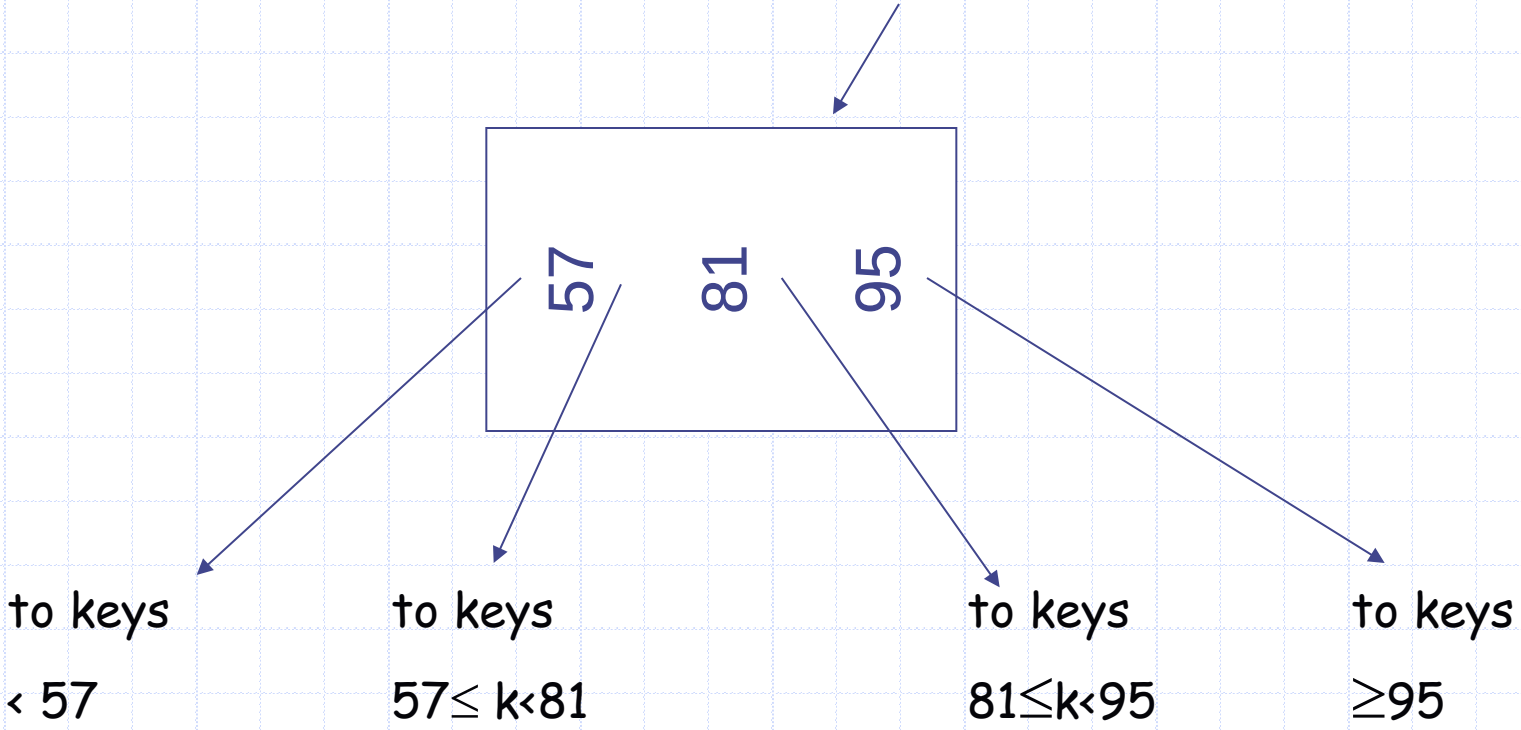
- 找到正确的叶子*L*.
- 把数据目录插入到*L*中.
  - 如果 *L* 有足够的空间，成功!
  - 否则, 必须把*L* 进行 分裂 (成为两个节点 *L*和新节点 *L2*)
    - 重新分配目录.
    - 把*L2*的作为*L*父节点的孩子节点.
- 这个过程可以重复进行
- 这种分使树的高度增高.
  - 树增长: 变wider宽 或者 高度增加1.

B+Tree Example

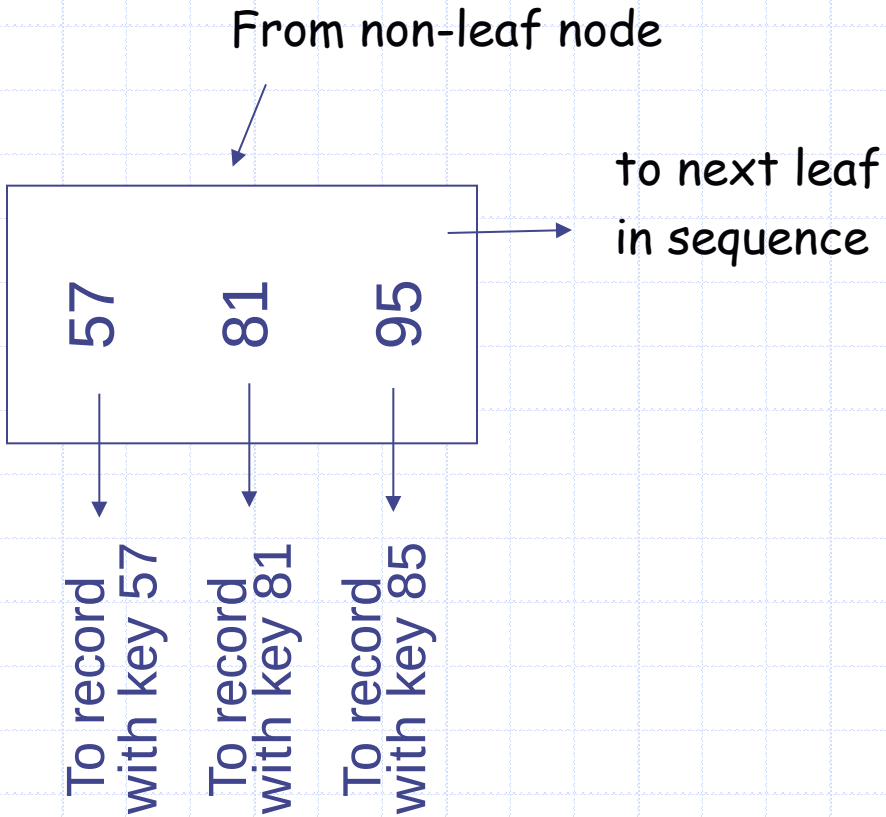
n=3



Sample non-leaf



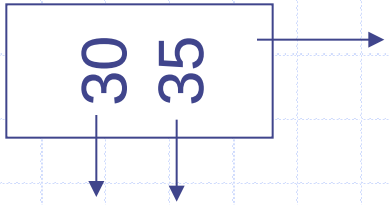
Sample leaf node:

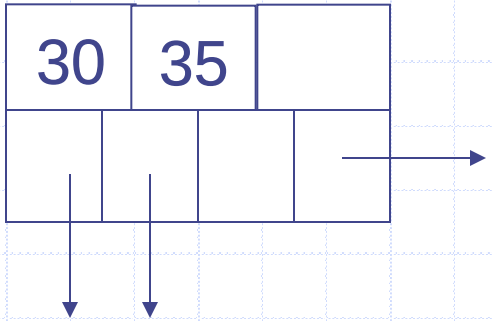


In textbook's notation

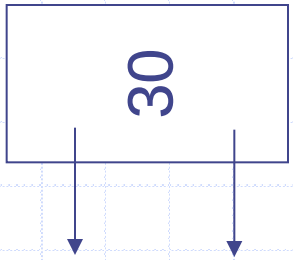
n=3

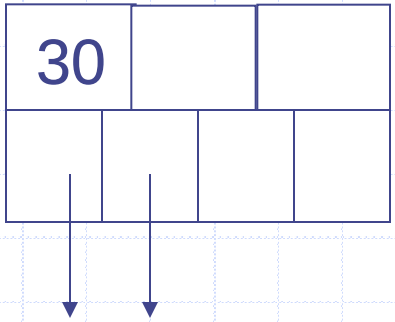
Leaf:





Non-leaf:





Size of nodes:

- n+1 pointers
- n keys

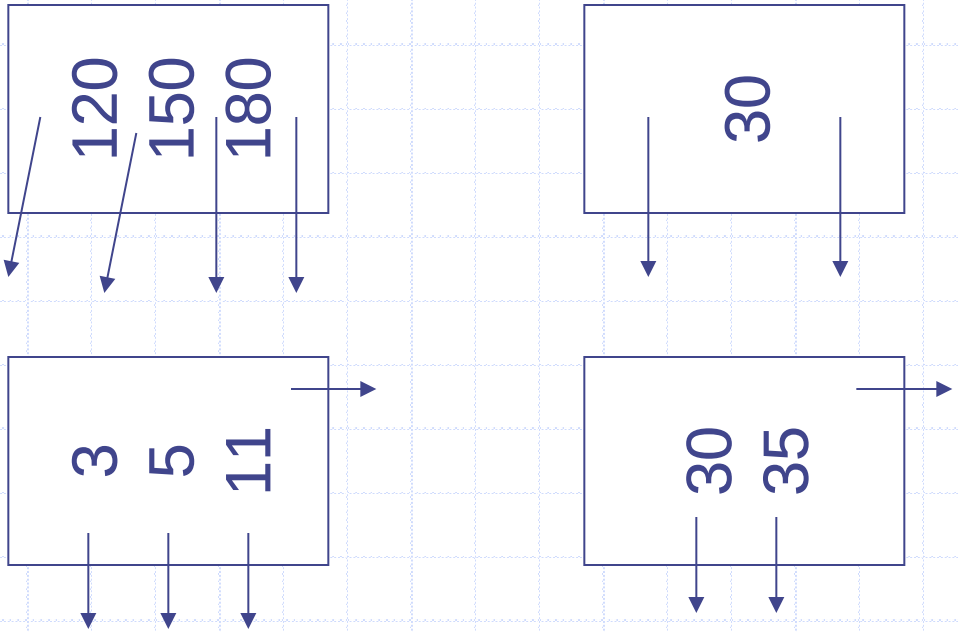
n=3

Full node

min. node

Non-leaf

Leaf



B+tree rules

(1) All leaves at 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	1	1

Insert into B+tree

(a) simple case
- space available in leaf

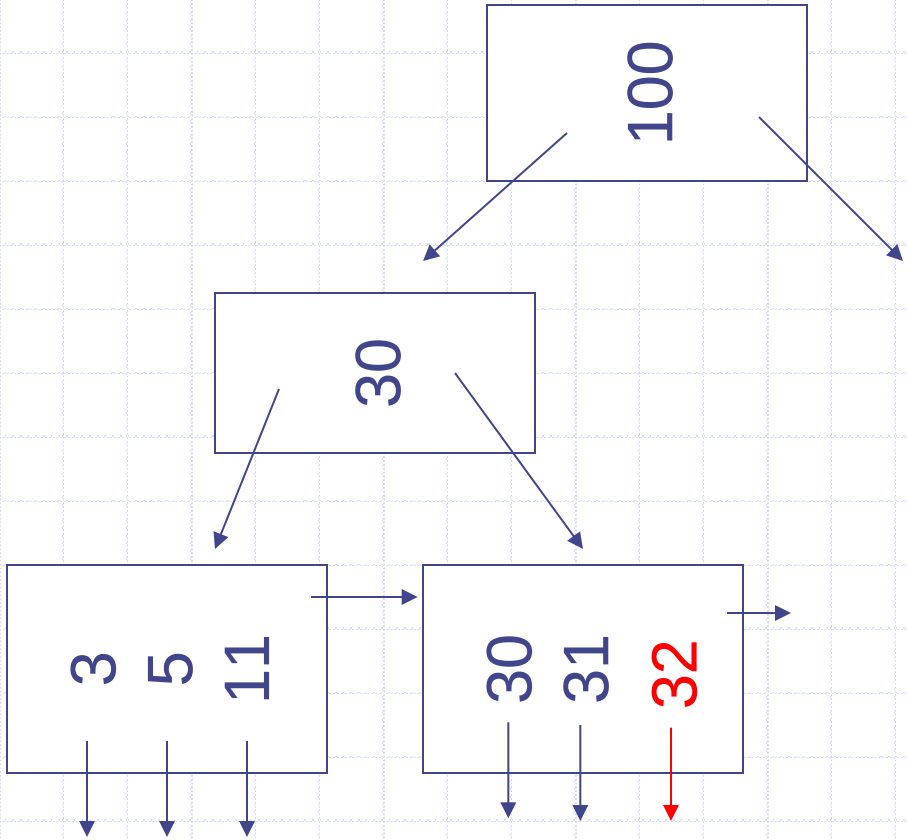
(b) leaf overflow

(c) non-leaf overflow

(d) new root

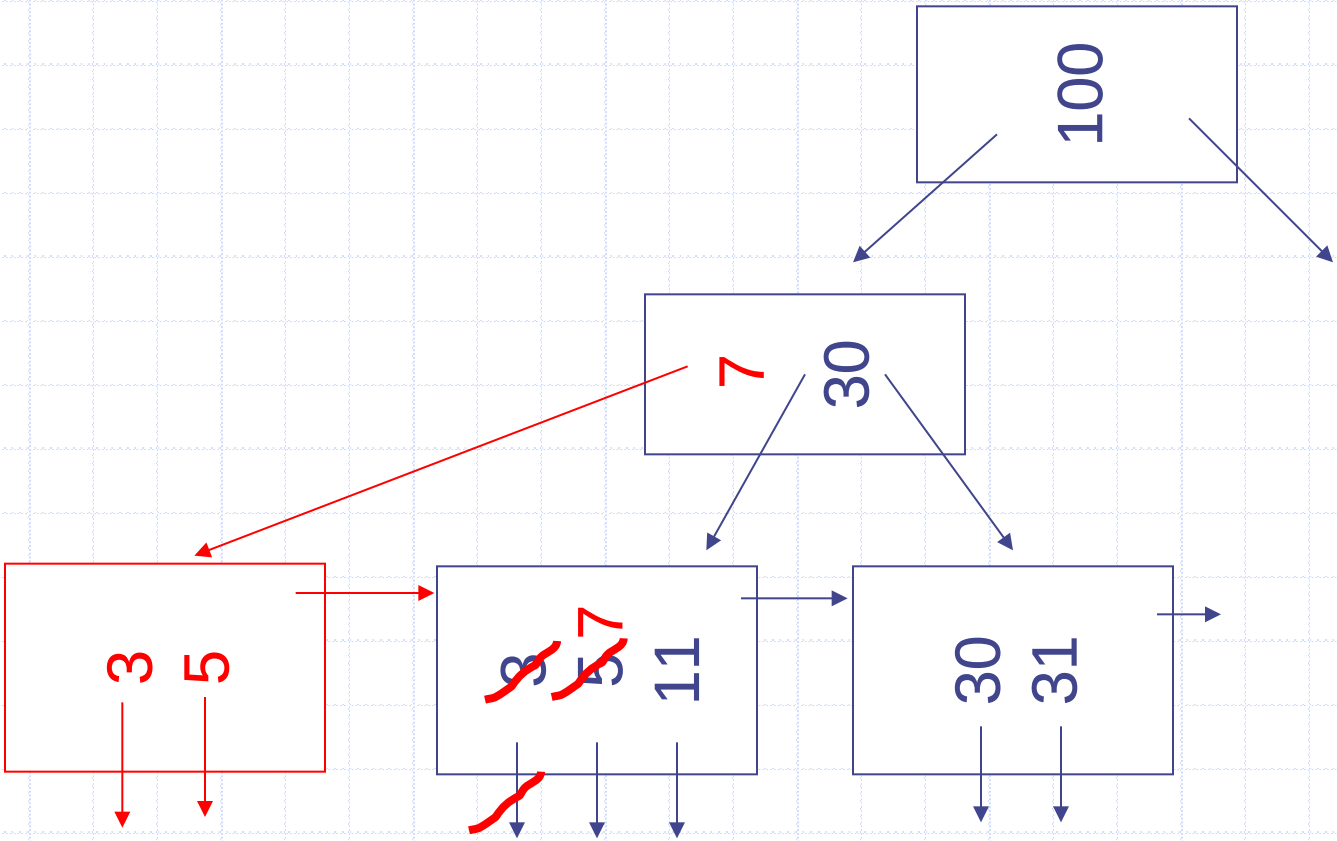
(a) Insert key = 32

n=3



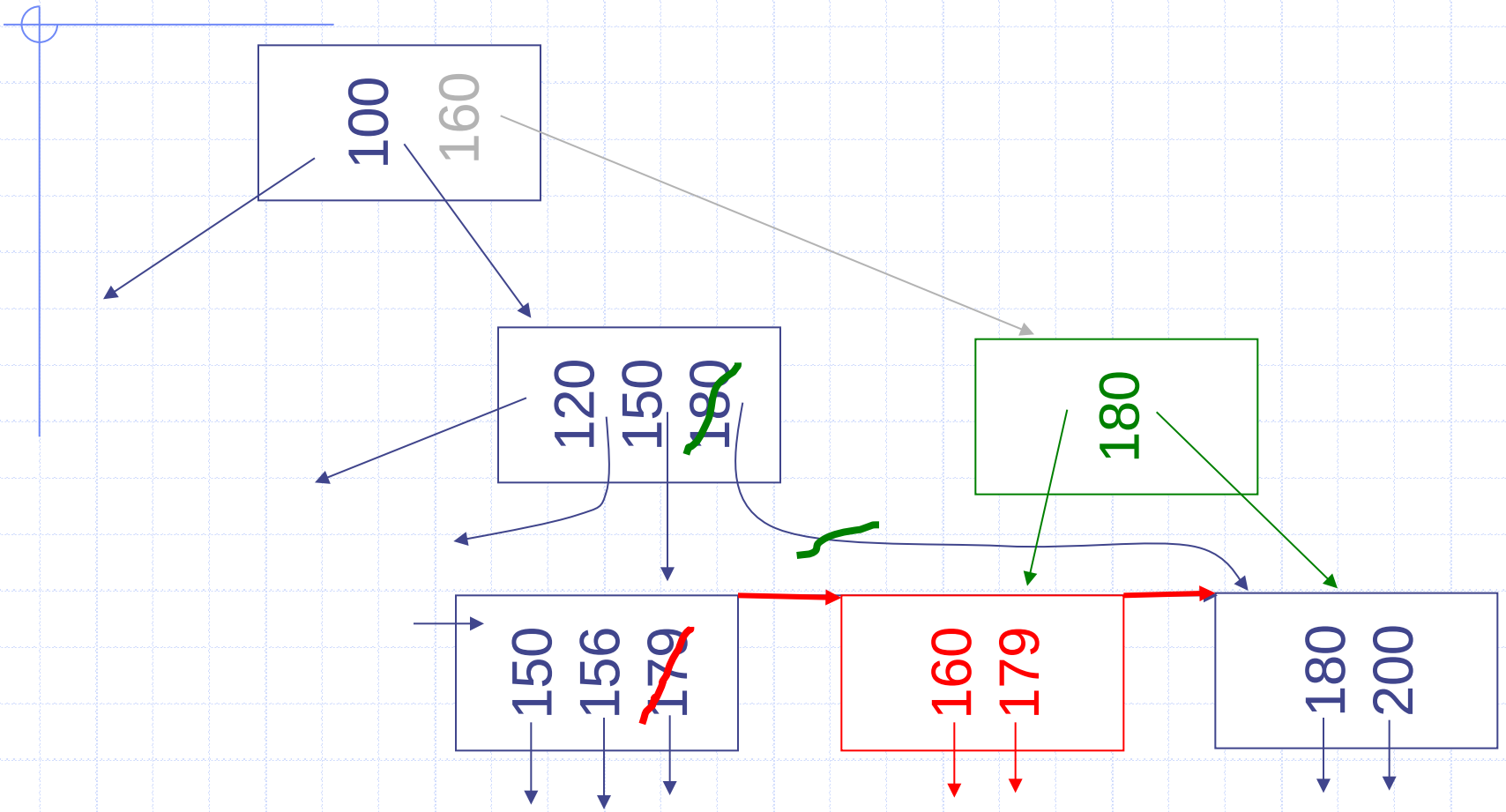
(b) Insert key = 7

n=3



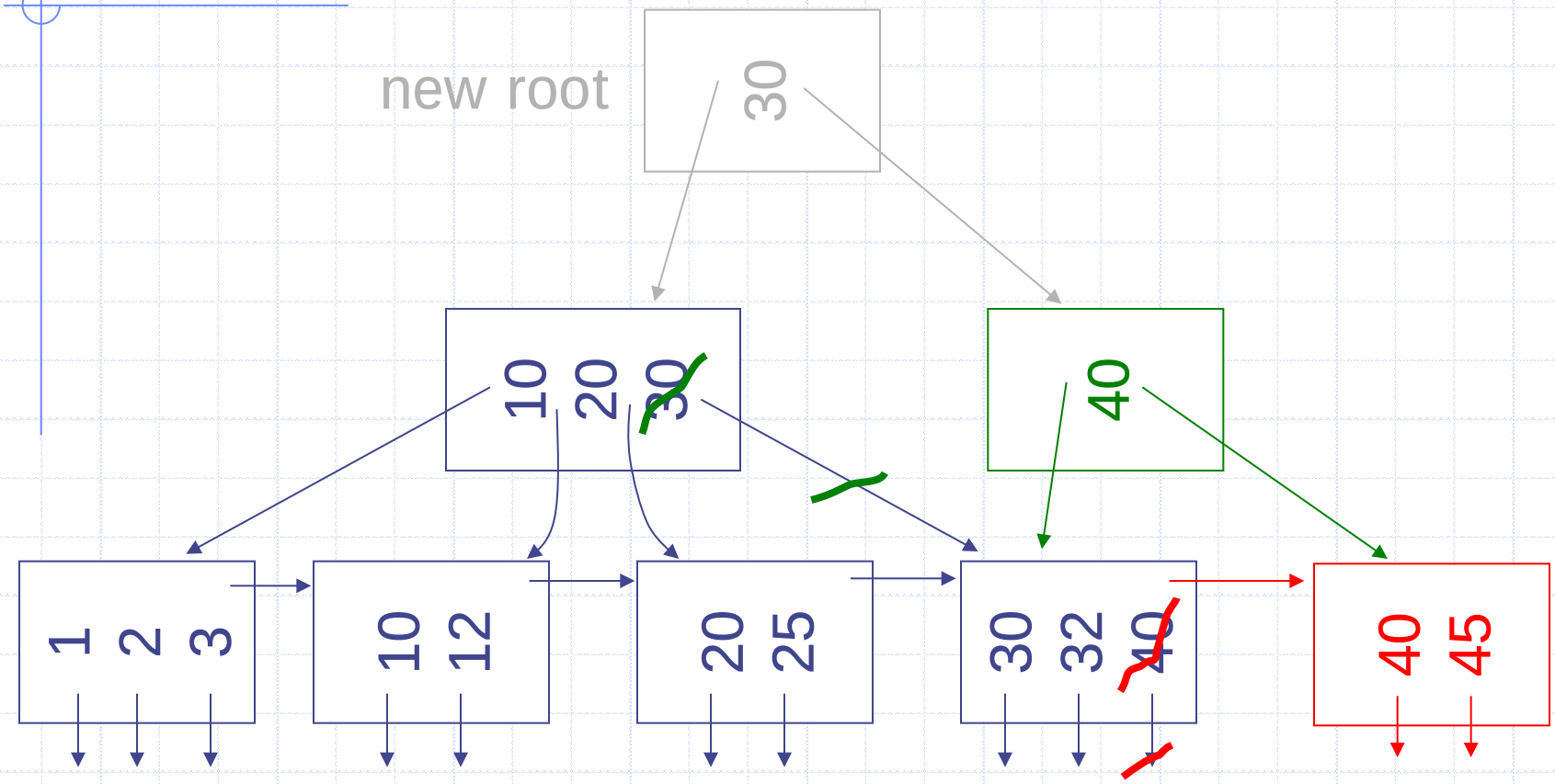
(c) Insert key = 160

n=3



(d) New root, insert 45

n=3



Deletion from B+tree

(a) Simple case - no example

(b) Coalesce with neighbor (sibling)

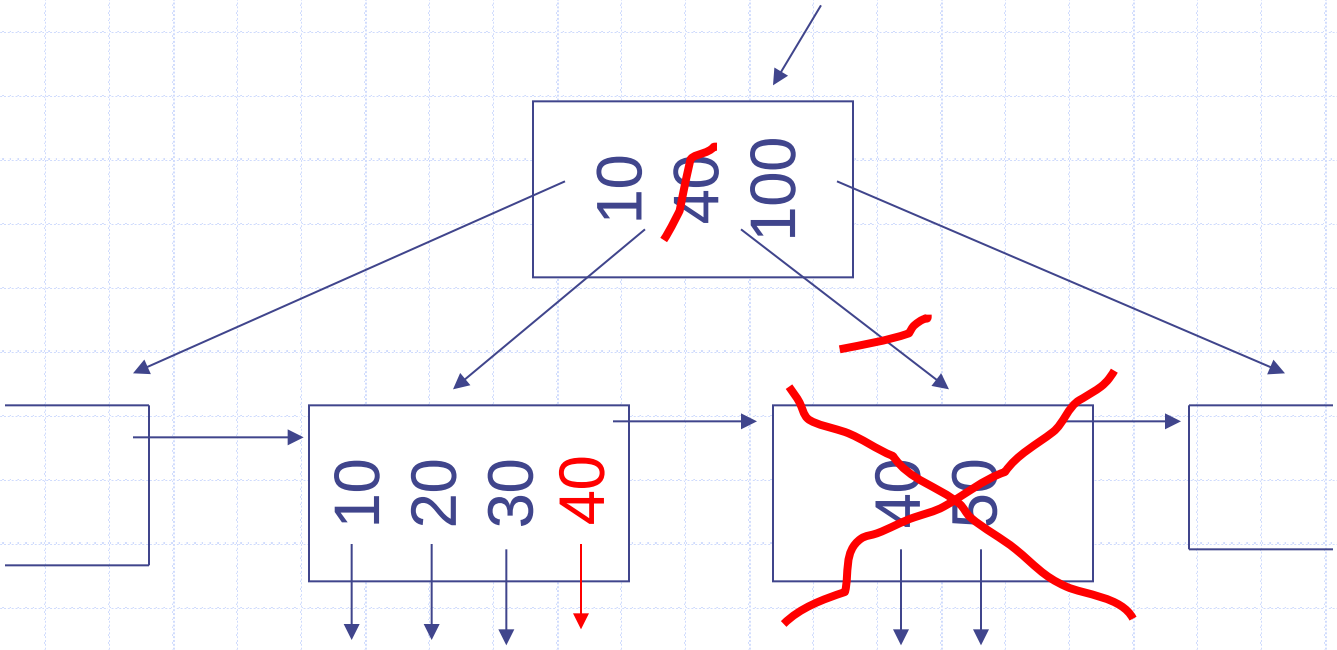
(c) Re-distribute keys

(d) Cases (b) or (c) at non-leaf

(b) Coalesce with sibling

- Delete 50

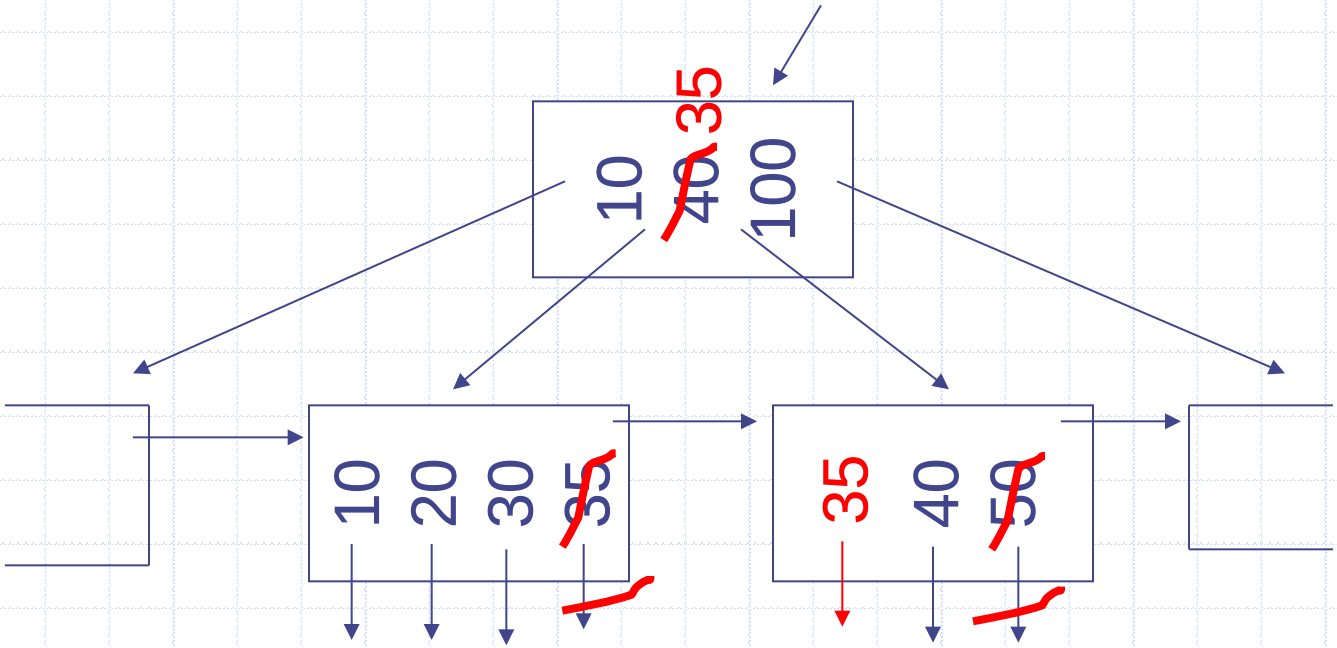
n=4



(c) Redistribute keys

- Delete 50

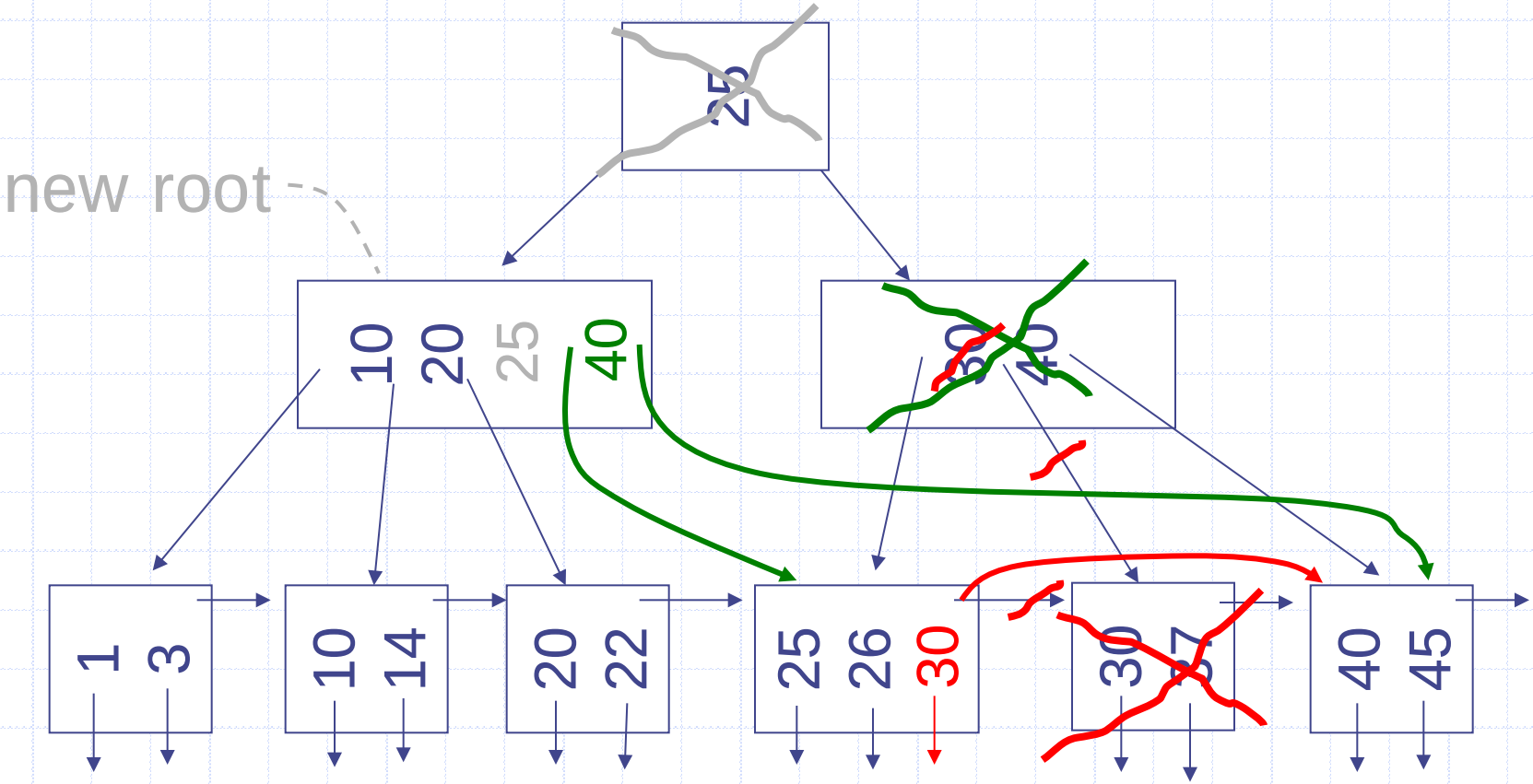
n=4



(d) Non-leaf coalese

- Delete 37

n=4

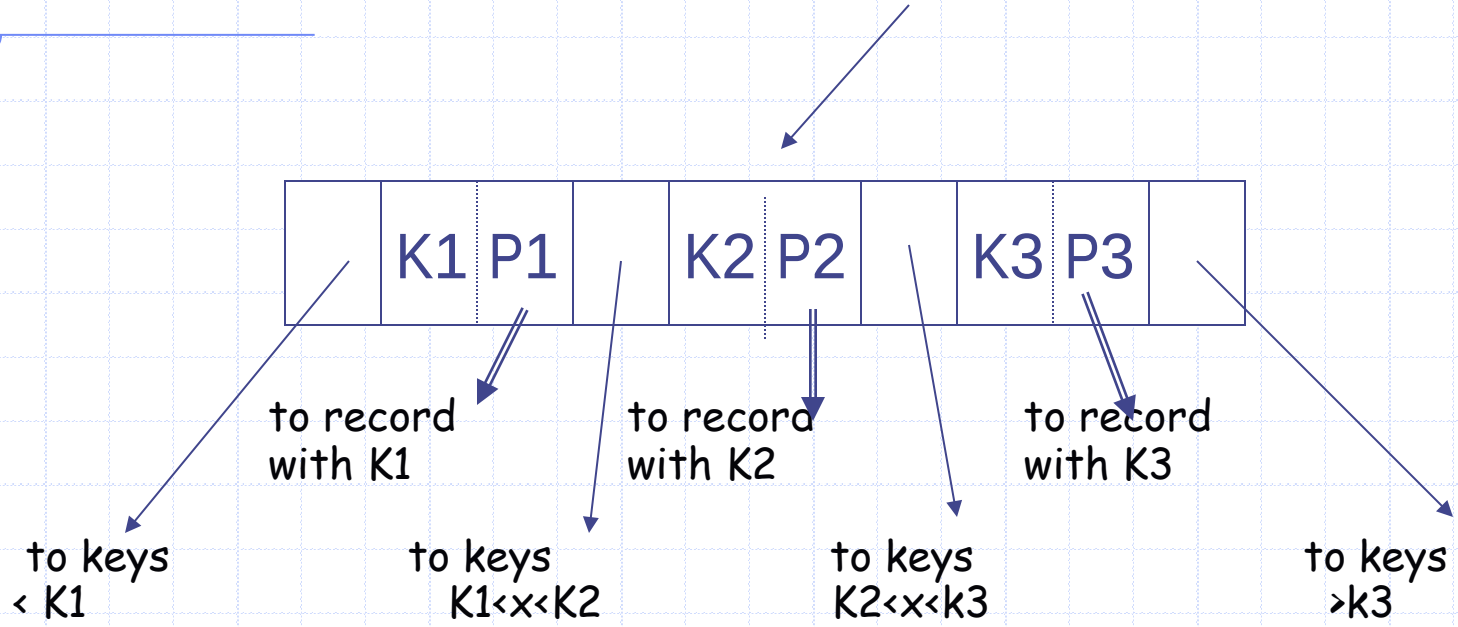


B+tree deletions in practice

- Often, coalescing(合并) is not implemented
  - Too hard and not worth it!

Variation on B+tree: B-tree (no +)

- Idea:
  - Avoid duplicate keys
  - Have record pointers in non-leaf nodes
  - B+ & B- difference?


K1
P1
K2
P2
K3
P3
to record with K1
to record with K2
to record with K3
to keys < K1
to keys K1<x<K2
to keys K2<x<k3
to keys >k3

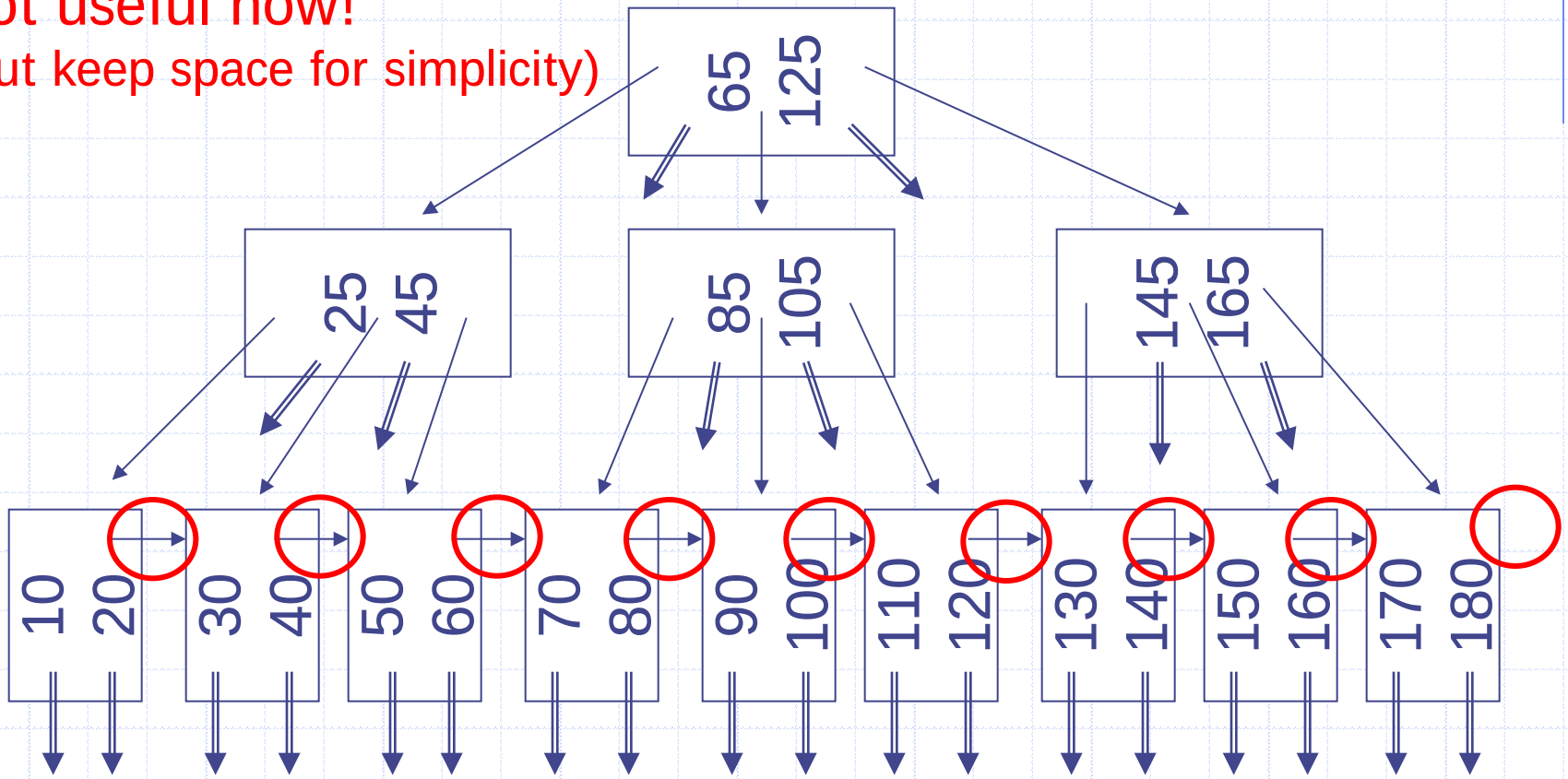
B-tree example

n=2

- sequence pointers

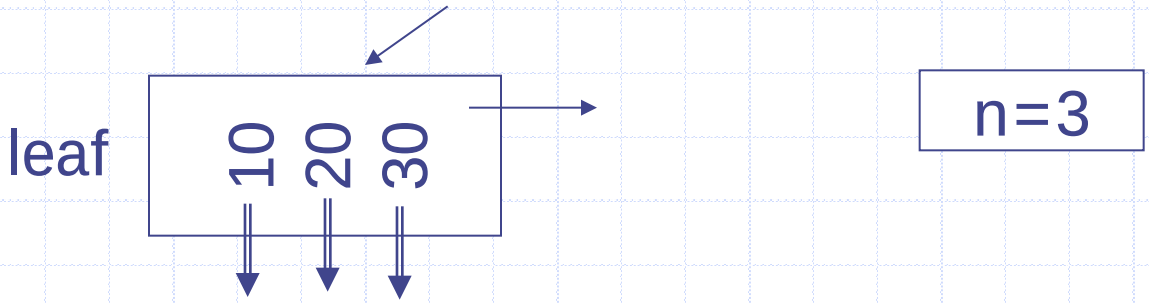
not useful now!

(but keep space for simplicity)

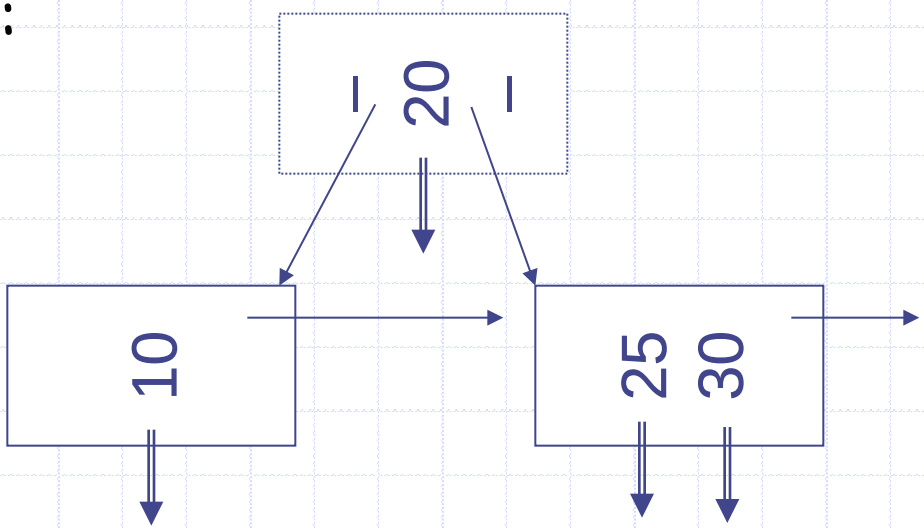


Note on inserts

- Say we insert record with key = 25



- Afterwards:



So, for B-trees:

	MAX			MIN		
	Tree Ptrs	Rec Ptrs	Keys	Tree Ptrs	Rec Ptrs	Keys
Non-leaf non-root	n+1	n	n	$\lceil (n+1)/2 \rceil$	$\lceil (n+1)/2 \rceil - 1$	$\lceil (n+1)/2 \rceil - 1$
Leaf non-root	1	n	n	1	$\lfloor (n+1)/2 \rfloor$	$\lfloor (n+1)/2 \rfloor$
Root non-leaf	n+1	n	n	2	1	1
Root Leaf	1	n	n	1	1	1

Tradeoffs:

- ☺ B-trees have faster lookup than B+trees
- ☹ in B-tree, non-leaf & leaf different sizes (the result?)
- ☹ in B-tree, deletion more complicated

➔ B+trees preferred!

Outline/summary

- Conventional Indexes
  - Sparse vs. dense
  - Primary vs. secondary
- B trees
  - B+trees vs. B-trees
  - B+trees vs. indexed sequential
- Hashing & Multidimentional Indexes --> Next