CSE 562
Database Systems

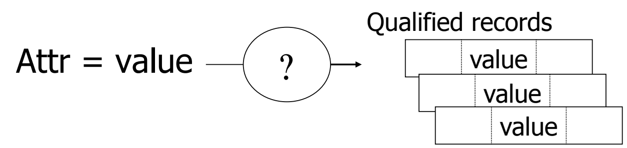
Indexing

Some slides are based or modified from originals by
Database Systems: The Complete Book,
Pearson Prentice Hall 2nd Edition

cse@buffalo

Goal of Indexing

Given condition(s) on attribute(s) find qualified records



Condition may also be

- Attr>value
- Attr>=value

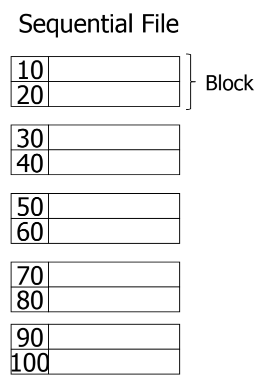
Indexes (or Indices)

- Data Structures used for quickly locating tuples that meet a specific type of condition
  - Equality condition: Find Movie tuples where Director=X
  - Range conditions: Find Employee tuples where Salary>40 AND Salary<50
- Many types of indexes. Evaluate them on:
  - Access time
  - Insertion/Deletion time
  - Condition types
  - Disk Space needed

Topics

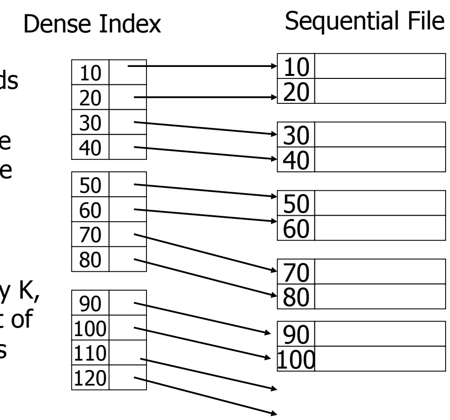
- Conventional indexes
- B-Trees
- Hashing schemes

Tuples are sorted by their primary key



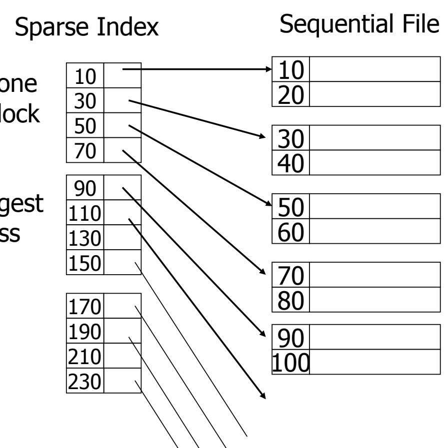
Index file needs much fewer blocks than the data file, hence easier to fit in memory

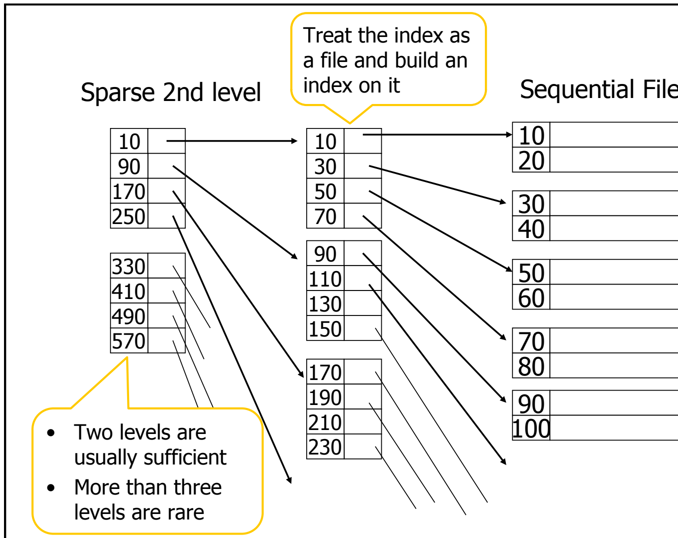
For a given key K, only $\log_2 n$, out of n, index blocks need to be accessed



Typically, only one key per data block

Find the index record with largest value that is less or equal to the value we are looking





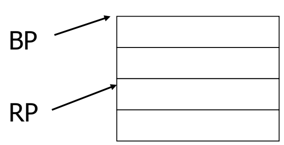
- Comment:
  {FILE,INDEX} may be contiguous
  or not (blocks chained)

Question:

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

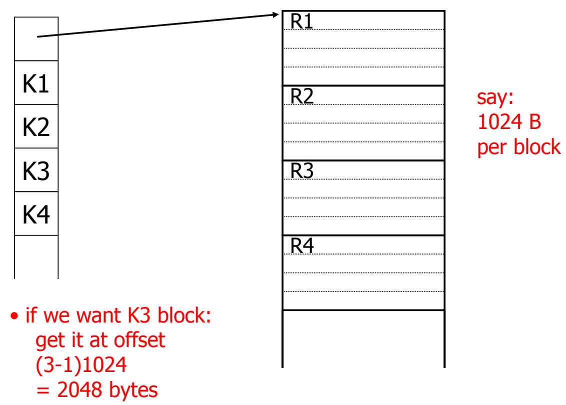
Notes on Pointers

- Record pointers consist of block pointer and position of record in the block
- Using the block pointer only saves space at no extra disk accesses cost
- Block pointer (sparse index) can be smaller than record pointer



Notes on Pointers

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



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

(Later:
  - sparse better for insertions
  - dense needed for secondary indexes)

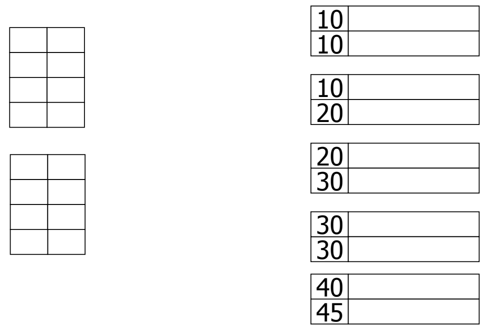
Terms

- Index sequential file
- Search key (≠ primary key)
- Primary index (on Sequencing field)
  - The index on the attribute (a.k.a. search key) that determines the sequencing of the table
- Secondary index
  - Index on any other attribute
- Dense index (all Search Key values in)
- Sparse index
- Multi-level index

Next

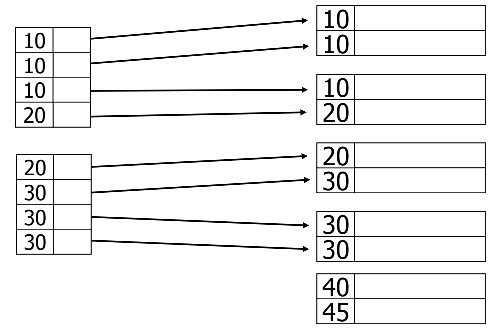
- Duplicate keys
- Deletion/Insertion
- Secondary indexes

Duplicate Keys



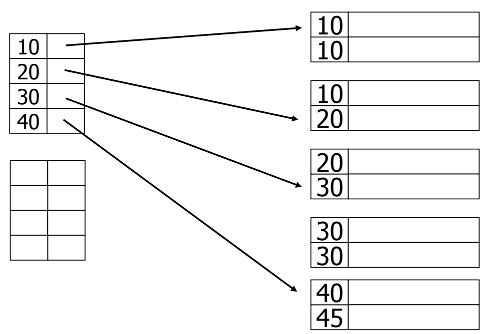
Duplicate Keys

Dense index, one way to implement?



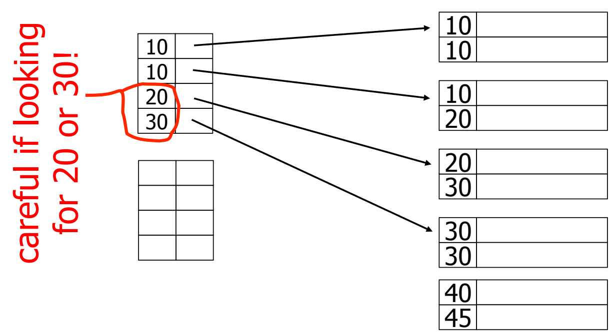
Duplicate Keys

Dense index, better way?



Duplicate Keys

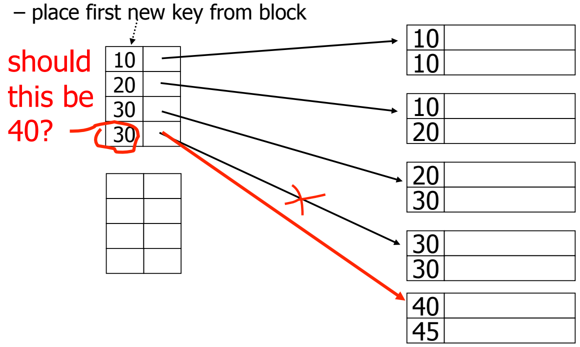
Sparse index, one way?



Duplicate Keys

Sparse index, another way?

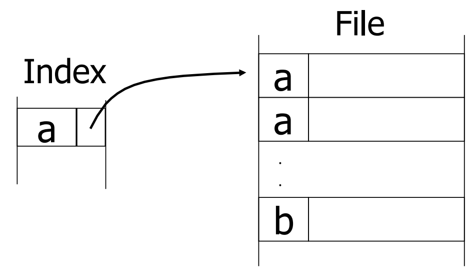
– place first new key from block



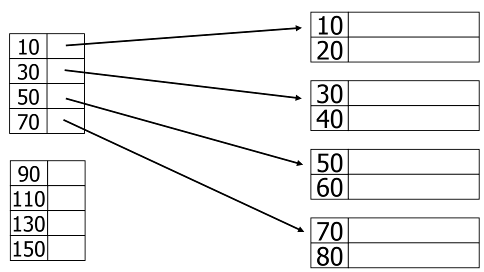
Summary

Duplicate values, primary index

- Index may point to first instance of each value only

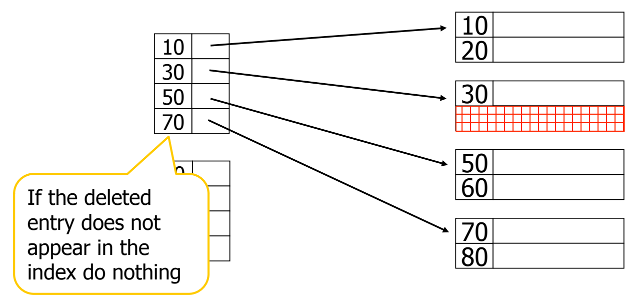


Deletion from sparse index



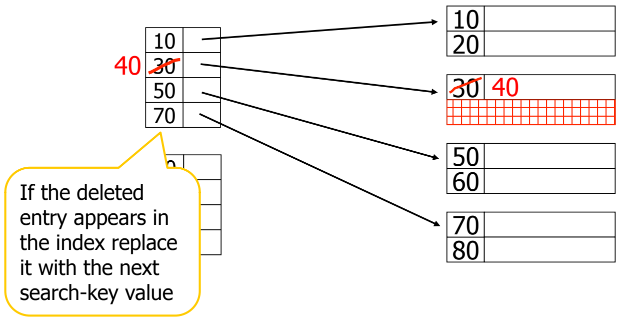
Deletion from sparse index

– delete record 40



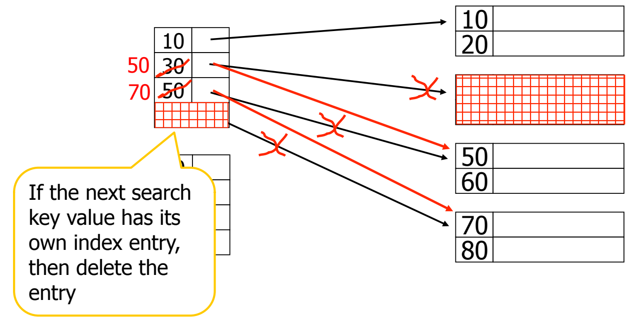
Deletion from sparse index

– delete record 30

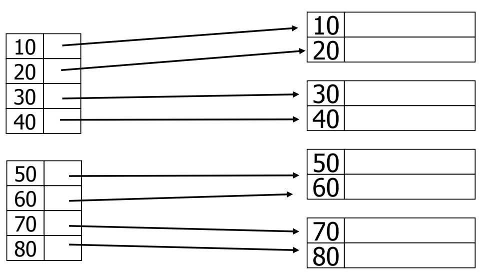


Deletion from sparse index

– delete records 30 & 40

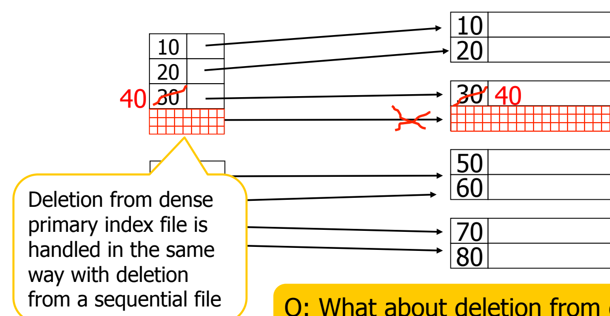


Deletion from dense index



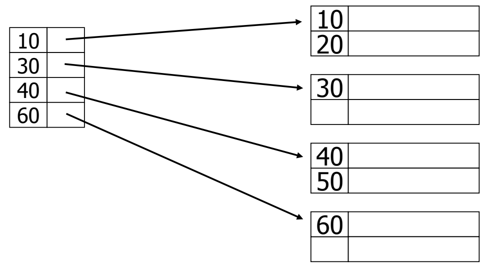
Deletion from dense index

– delete record 30



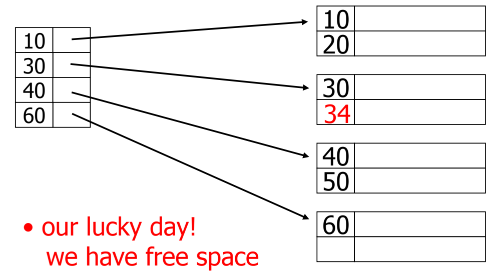
Q: What about deletion from dense primary index with duplicates?

Insertion, sparse index case



Insertion, sparse index case

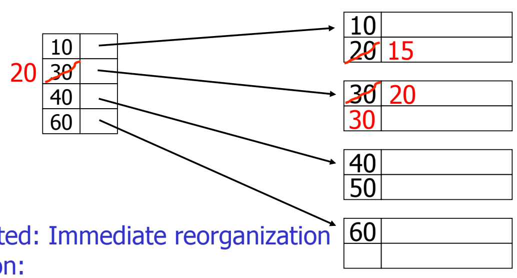
– insert record 34



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

Insertion, sparse index case

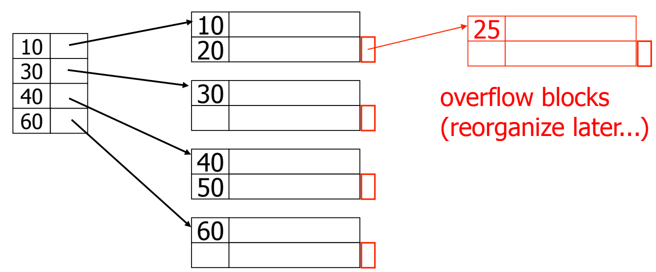
– insert record 15



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

Insertion, sparse index case

– insert record 25

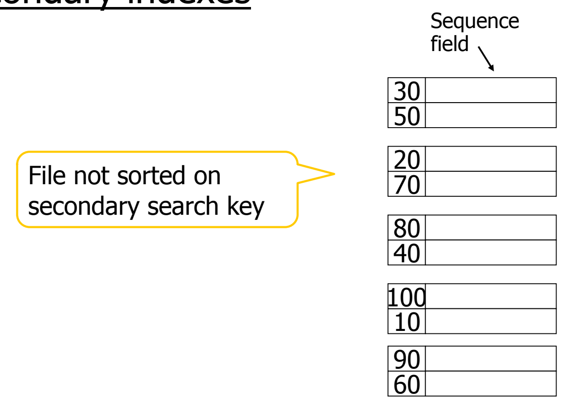


• How often do we reorganize and how expensive is it?
B-Trees offer convincing answers

Insertion, dense index case

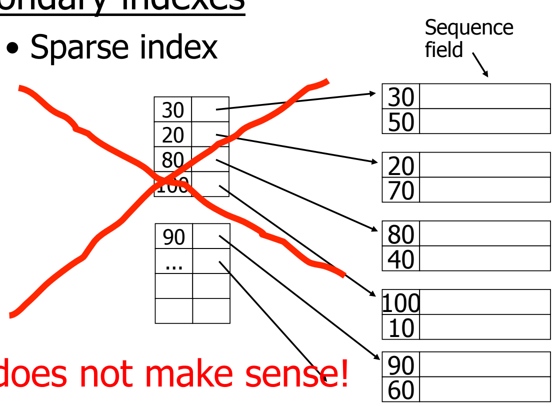
- Similar
- Often more expensive . . .

Secondary indexes



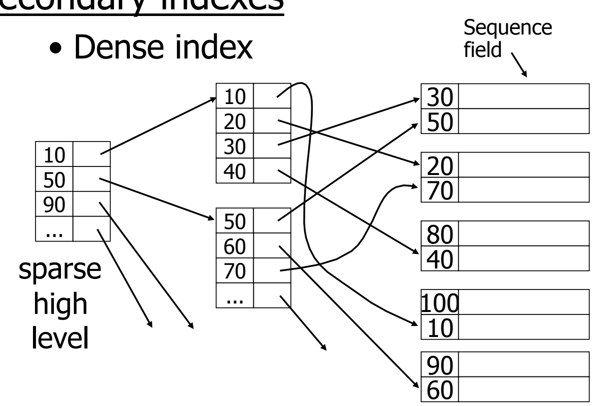
Secondary indexes

- Sparse index



Secondary indexes

- Dense index



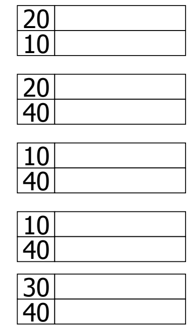
With secondary indexes:

- Lowest level is dense
- Other levels are sparse

Also: Pointers are record pointers

(not block pointers; not computed)

Duplicate values & secondary indexes

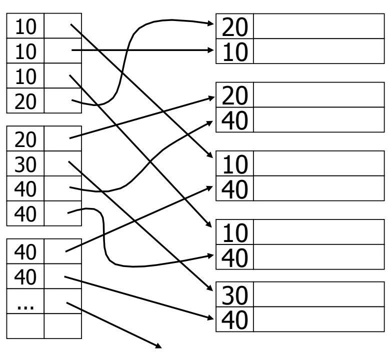


Duplicate values & secondary indexes

one option...

Problem:
excess overhead!

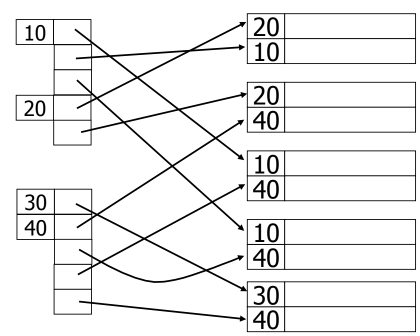
- disk space
- search time



Duplicate values & secondary indexes

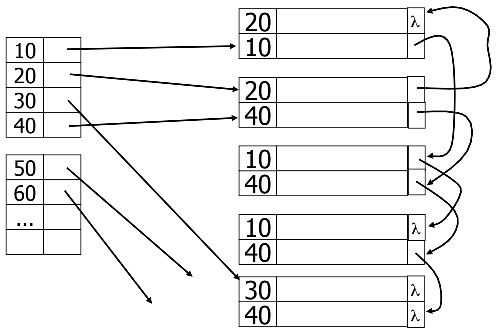
another option...

Problem:
variable size records in index!



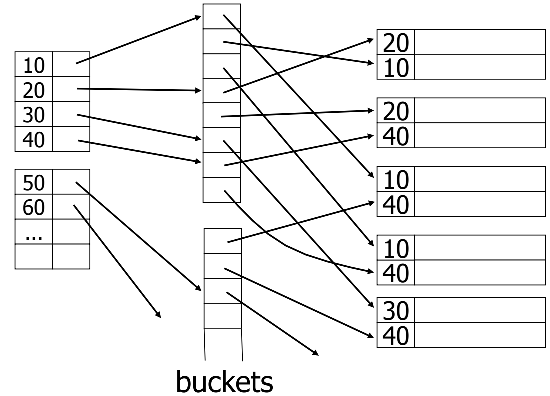
Duplicate values & secondary indexes

Another idea: Chain records with same key?



Problems:
- Need to add fields to records, messes up maintenance
- Need to follow chain to know records

Duplicate values & secondary indexes



Why "bucket" idea is useful

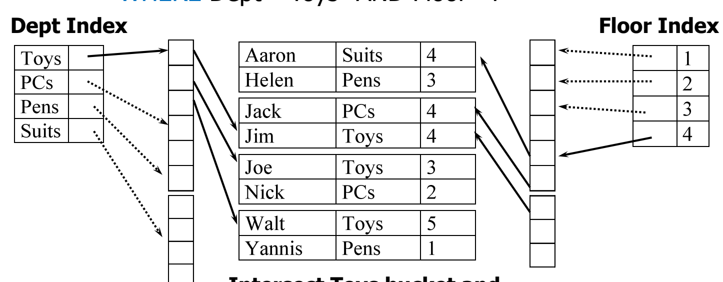
Indexes	Records
Name: primary	EMP (name, dept, floor, ...)
Dept: secondary	
Floor: secondary	

- **Enables the processing of queries working with pointers only**
- Very common technique in Information Retrieval

Advantage of Buckets: Process Queries Using Pointers Only

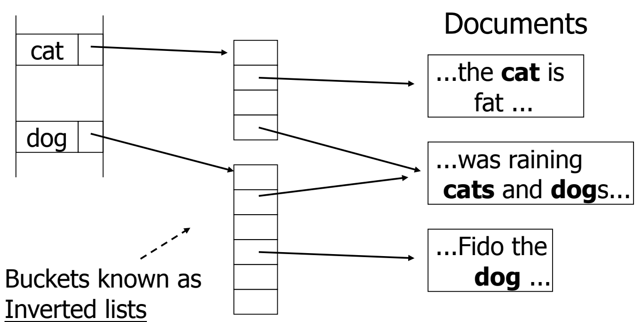
Find employees in Toys dept on the 4th floor:

SELECT Name FROM Employee
WHERE Dept="Toys" AND Floor=4



Intersect Toys bucket and 4^{th} floor bucket to get set of matching EMP's

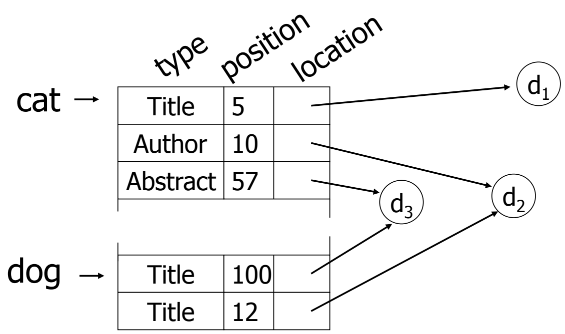
This idea used in
<u>text information retrieval</u>



<u>IR QUERIES</u>

- Find articles with "cat" and "dog"
  - Intersect inverted lists
- Find articles with "cat" or "dog"
  - Union inverted lists
- Find articles with "cat" and not "dog"
  - Subtract list of dog pointers from list of cat pointers
- Find articles with "cat" in title
- Find articles with "cat" and "dog" within 5 words

<u>Common technique</u>:
more info in inverted list



<u>Posting:</u> an entry in inverted list.
Represents occurrence of term in article

<u>Size of a list:</u> (in postings)

1 — Rare words or miss-spellings

↓

10^6 — Common words

<u>Size of a posting:</u> 10-15 bits (compressed)

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...

Summary of Indexing So Far

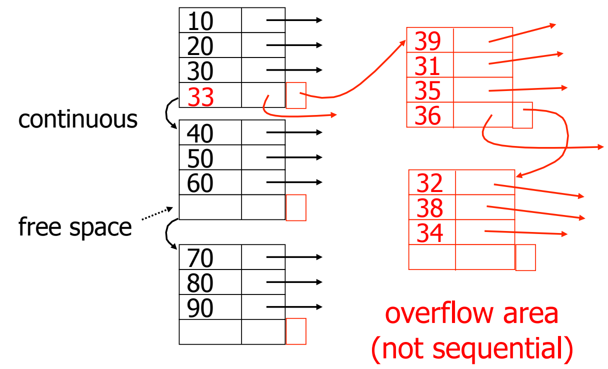
- Conventional index
  - Basic Ideas: sparse, dense, multi-level…
  - Duplicate Keys
  - Deletion/Insertion
  - Secondary indexes
    - Buckets of Postings List

Conventional Indexes

Advantage:
- Simple algorithms
- Index is sequential file
  good for scans

Disadvantage:
- Inserts expensive, and/or
- Lose sequentiality,
  reorganizations are needed

Example Index (sequential)

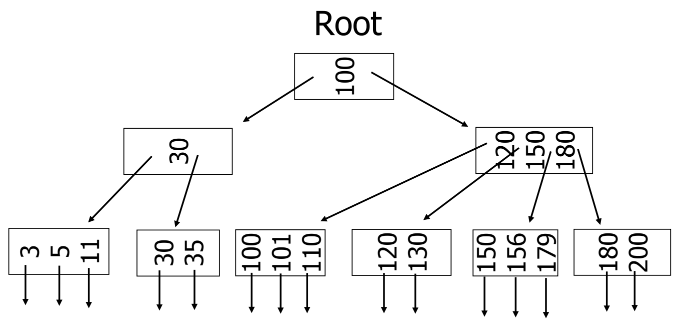


Topics

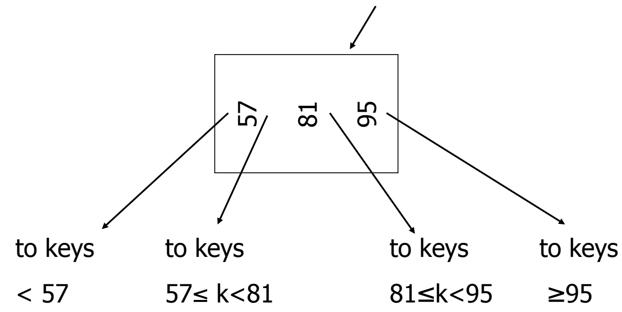
- Conventional indexes
- B-Trees ⇒ NEXT
- Hashing schemes

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

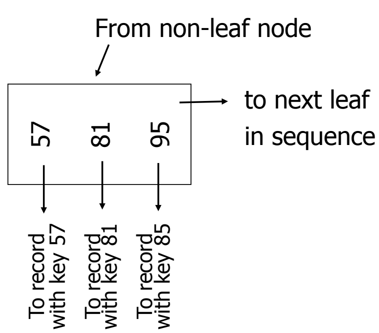
B+Tree Example n=3



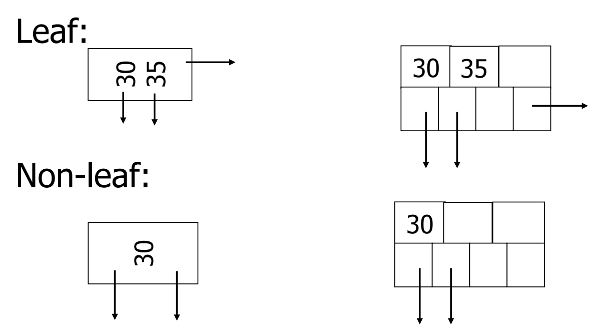
Sample non-leaf



Sample leaf node:



In textbook's notation n=3



Size of nodes: n+1 pointers, n keys (fixed)

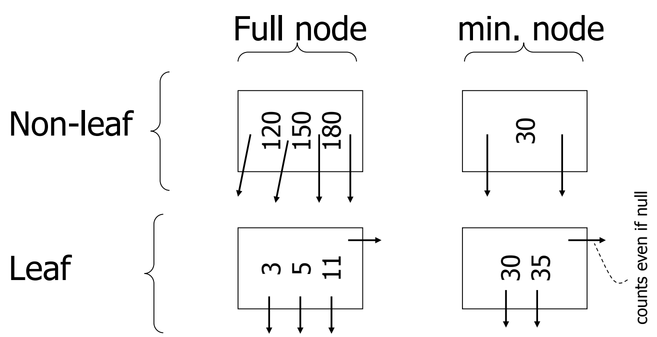
Don't want nodes to be too empty

- Non-root nodes have to be at least half-full
- Use at least

Non-leaf: $\lceil (n+1)/2 \rceil$ pointers

Leaf: $\lfloor (n+1)/2 \rfloor$ pointers to data

n=3



B+Tree rules tree of order *n*

(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

Counting sequence pointer also

Insert into B+Tree

(a) simple case
- space available in leaf

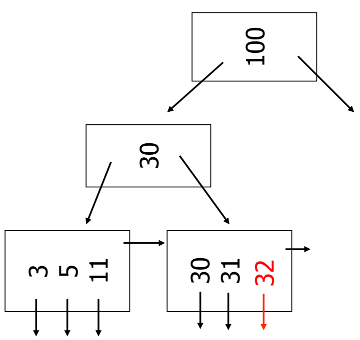
(b) leaf overflow

(c) non-leaf overflow

(d) new root

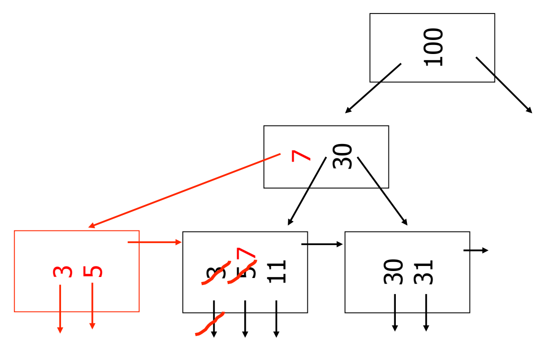
(a) Insert key = 32

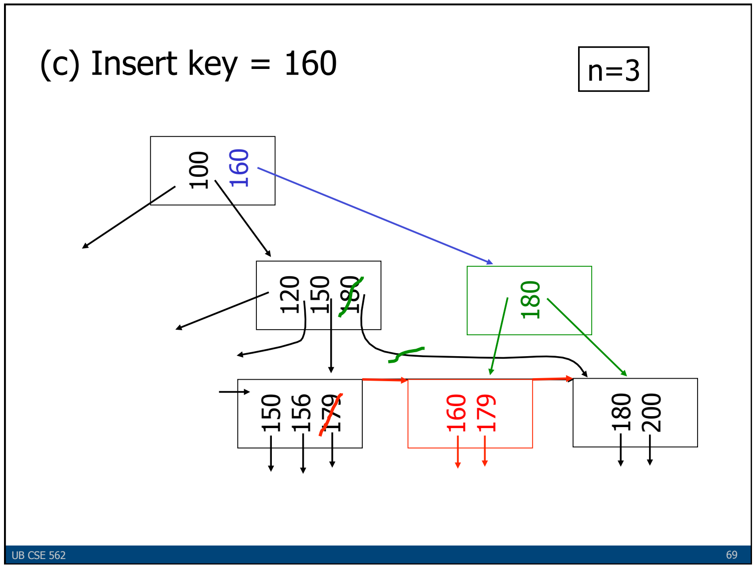
n=3

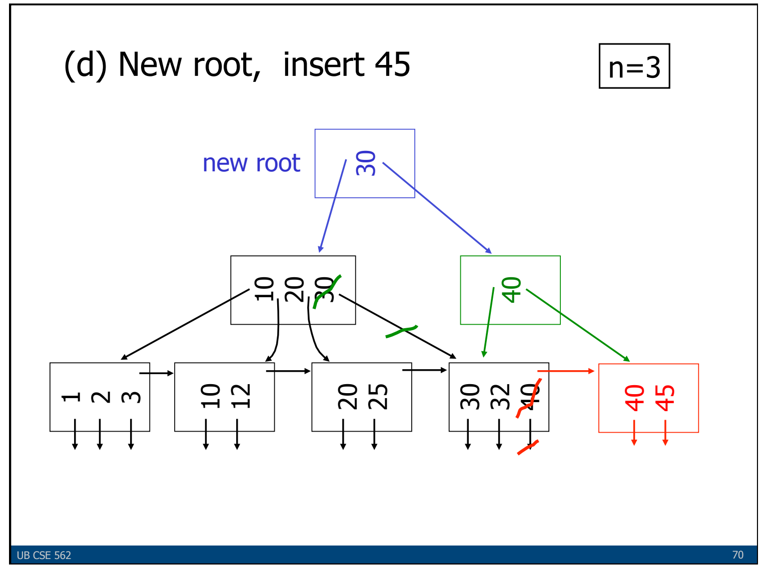


(a) Insert key = 7

n=3







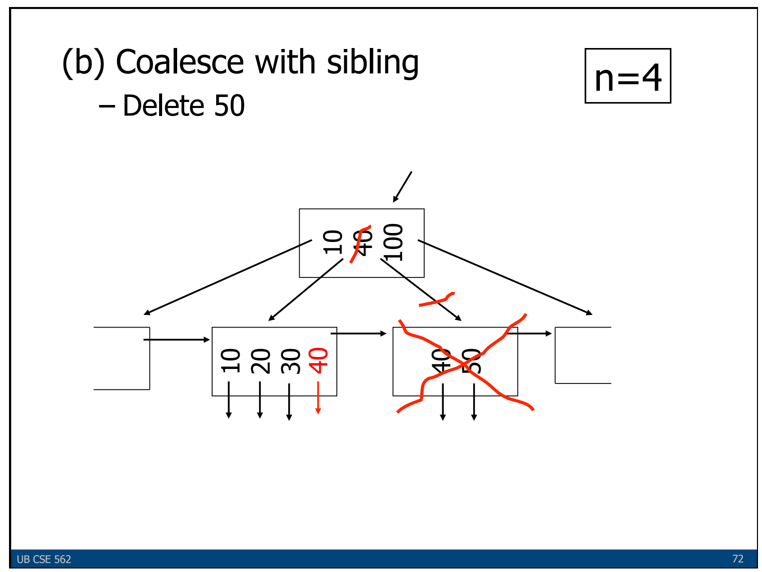
Deletion from B+Tree

(a) Simple case - no example

(b) Coalesce with neighbor (sibling)

(c) Re-distribute keys

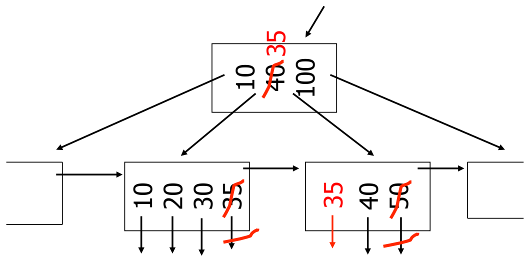
(d) Cases (b) or (c) at non-leaf



(c) Redistribute keys

n=4

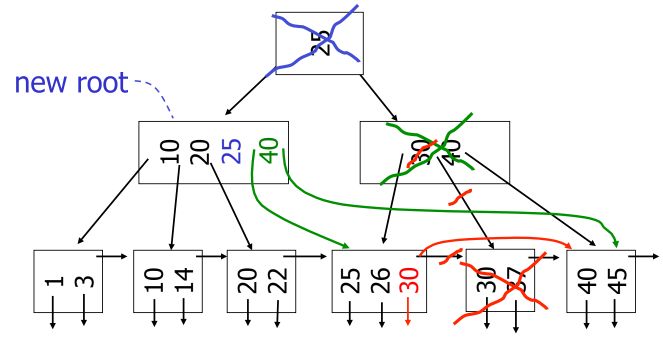
– Delete 50



(d) Non-leaf coalese

n=4

– Delete 37



B+Tree deletions in practice

– Often, coalescing is not implemented
  – Too hard and not worth it!

Comparison: B-Trees vs. static indexed sequential file

Ref #1: Held & Stonebraker
"B-Trees Re-examined"
CACM, Feb. 1978

Ref # 1 claims:

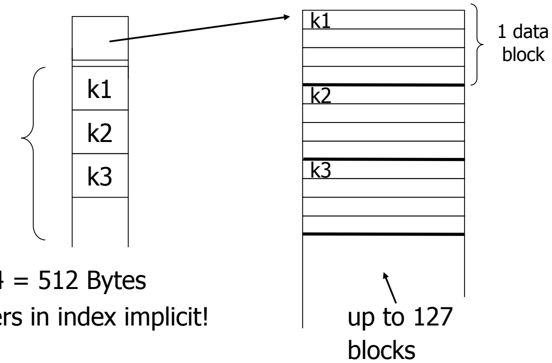
- Concurrency control harder in B-Trees
- B-Tree consumes more space

For their comparison:

block = 512 bytes
key = pointer = 4 bytes
4 data records per block

Example: 1 block static index

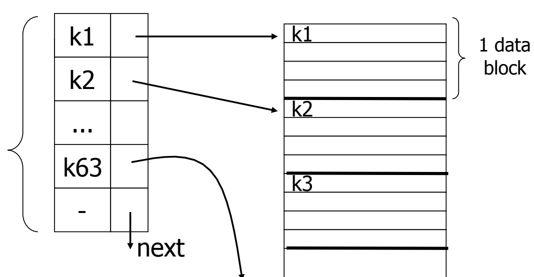
127 keys



(127+1)4 = 512 Bytes
-> pointers in index implicit!

Example: 1 block B-Tree

63 keys



up to 63 blocks

63x(4+4)+8 = 512 Bytes
-> pointers needed in B-Tree blocks because index is not contiguous

Size comparison Ref. #1

Static Index: # data blocks	height	B-Tree: # data blocks	height
2 -> 127	2	2 -> 63	2
128 -> 16,129	3	64 -> 3,968	3
16,130 -> 2,048,383	4	3,969 -> 250,047	4
		250,048 -> 15,752,961	5

Ref. #1 analysis claims

- For an 8,000 block file,
  - after 32,000 inserts
  - after 16,000 lookups

  ⇒ Static index saves enough accesses to allow for reorganization

Ref. #1 conclusion ⇒ Static index better!!

Ref #2: M. Stonebraker, "Retrospective on a database system," TODS, June 1980

Ref. #2 conclusion ⇒ B-Trees better!!

Ref. #2 conclusion ⇒ B-Trees better!!

- DBA does not know when to reorganize
- DBA does not know how full to load pages of new index

Ref. #2 conclusion ⇒ B-Trees better!!

- Buffering
  - B-Tree: has fixed buffer requirements
  - Static index: must read several overflow blocks to be efficient (large & variable size buffers needed for this)

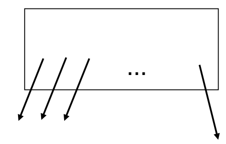
- Speaking of buffering…
  Is LRU a good policy for B+Tree buffers?

→ Of course not!

→ Should try to keep root in memory at all times
(and perhaps some nodes from second level)

Interesting problem:

For B+Tree, how large should n be?



n is number of keys / node

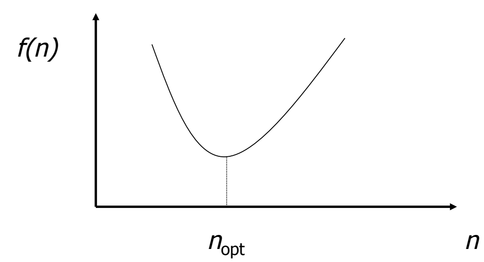
Sample assumptions:

(1) Time to read node from disk is
(S+Tn) msec.

(2) Once block in memory, use binary search to locate key:
($a + b \, \text{LOG}_2 \, n$) msec.
For some constants a,b; Assume a << S

(3) Assume B+Tree is full, i.e.,
\# nodes to examine is $\text{LOG}_n N$
where N = # records

➛Can get:
$f(n)$ = time to find a record



➸ FIND n_{opt} by $f'(n) = 0$

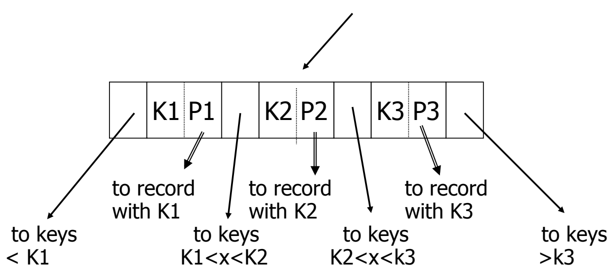
Answer should be n_{opt} = "few hundred"

➸ What happens to n_{opt} as

- Disk gets faster?
- CPU get faster?

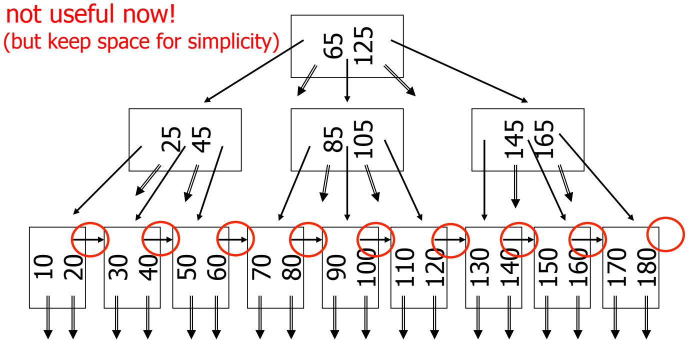
Variation on B+Tree: B-Tree (no +)

- Idea:
  - Avoid duplicate keys
  - Have record pointers in non-leaf nodes



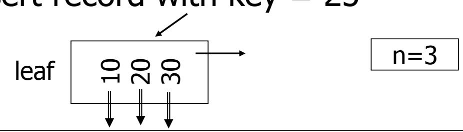
B-Tree example n=2

- sequence pointers not useful now!
  (but keep space for simplicity)

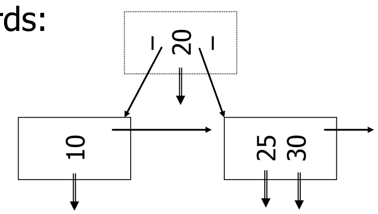


Note on inserts

- 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	⌈(n+1)/2⌉	⌈(n+1)/2⌉-1	⌈(n+1)/2⌉-1
Leaf non-root	1	n	n	1	⌊(n+1)/2⌋	⌊(n+1)/2⌋
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

☹ in B-Tree, smaller fan-out

☹ in B-Tree, deletion more complicated

➨ B+Trees preferred!

But note:

- If blocks are fixed size
  (due to disk and buffering restrictions)

Then lookup for B+Tree is
actually better!!

Example:

- Pointers 4 bytes
- Keys 4 bytes
- Blocks 100 bytes (just example)
- Look at full 2 level tree

B-Tree:

Root has 8 keys + 8 record pointers + 9 son pointers
= 8x4 + 8x4 + 9x4 = 100 bytes

Each of 9 sons: 12 rec. pointers (+12 keys)
= 12x(4+4) + 4 = 100 bytes

2-level B-Tree, Max # records =
12x9 + 8 = 116

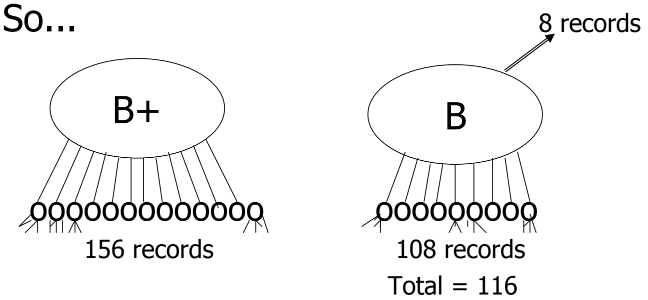
B+Tree:

Root has 12 keys + 13 son pointers
= 12x4 + 13x4 = 100 bytes

Each of 13 sons: 12 rec. ptrs (+12 keys)
= 12x(4 +4) + 4 = 100 bytes

2-level B+Tree, Max # records
= 13x12 = 156

So...



- Conclusion:
  - For fixed block size,
  - B+Tree is better because it is bushier

An Interesting Problem…

- What is a good index structure when:
  - records tend to be inserted with keys that are larger than existing values? (e.g., banking records with growing data/time)
  - we want to remove older data

One Solution: Multiple Indexes

- Example: I1, I2

day	days indexed I1	days indexed I2
10	1,2,3,4,5	6,7,8,9,10
11	11,2,3,4,5	6,7,8,9,10
12	11,12,3,4,5	6,7,8,9,10
13	11,12,13,4,5	6,7,8,9,10

•advantage: deletions/insertions from smaller index
•disadvantage: query multiple indexes

Another Solution (Wave Indexes)

day	I1	I2	I3	I4
10	1,2,3	4,5,6	7,8,9	10
11	1,2,3	4,5,6	7,8,9	10,11
12	1,2,3	4,5,6	7,8,9	10,11, 12
13	13	4,5,6	7,8,9	10,11, 12
14	13,14	4,5,6	7,8,9	10,11, 12
15	13,14,15	4,5,6	7,8,9	10,11, 12
16	13,14,15	16	7,8,9	10,11, 12

•advantage: no deletions
•disadvantage: approximate windows

This Time

- Conventional Indexes
  - Chapter 14: 14.1
    - Sparse vs. dense
    - Primary vs. secondary
- B-Trees
  - Chapter 14: 14.2
    - B+Trees vs. B-Trees vs. indexed sequential