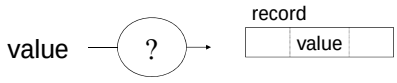
CS 245: Database System Principles

Notes 4: Indexing

Hector Garcia-Molina

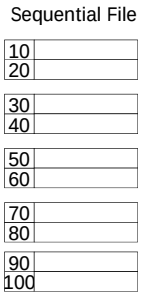
Chapter 4

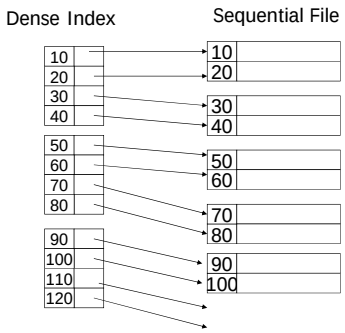
Indexing & Hashing

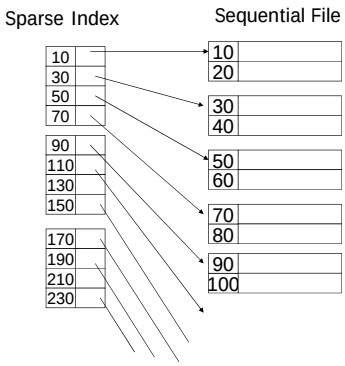


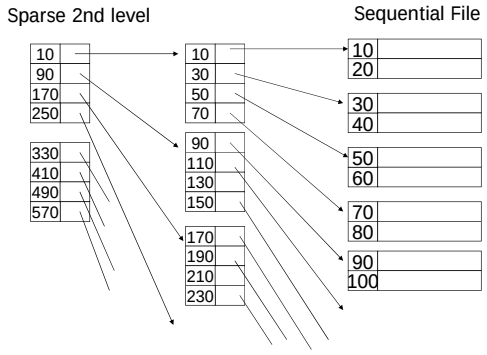
Topics

- Conventional indexes
- B-trees
- Hashing schemes









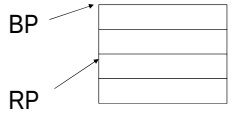
- Comment:
  {FILE,INDEX} may be contiguous
  or not (blocks chained)

Question:

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

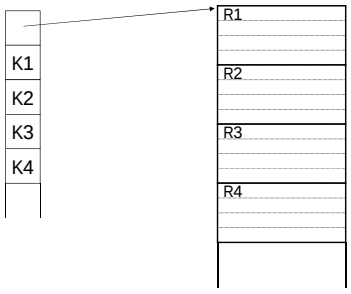
Notes on pointers:

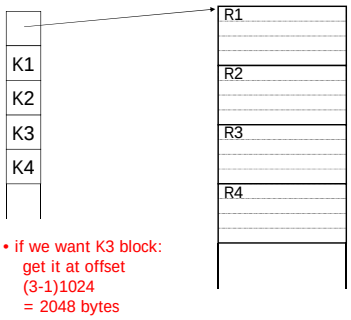
(1) Block pointer (sparse index) can be smaller than record pointer



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

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

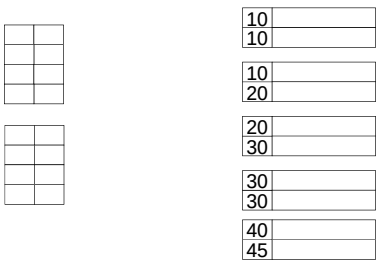
Terms

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

Next:

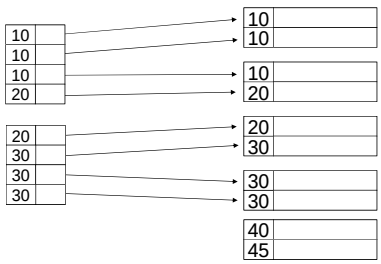
- Duplicate keys
- Deletion/Insertion
- Secondary indexes

Duplicate keys



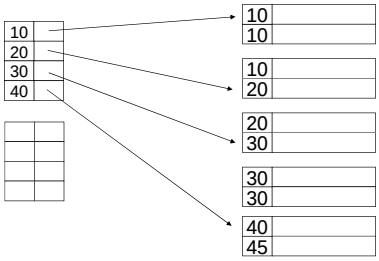
Duplicate keys

Dense index, one way to implement?



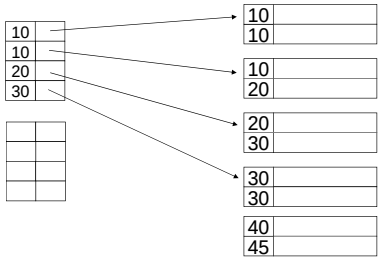
Duplicate keys

Dense index, better way?



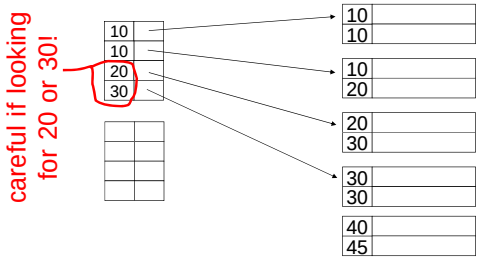
Duplicate keys

Sparse index, one way?



Duplicate keys

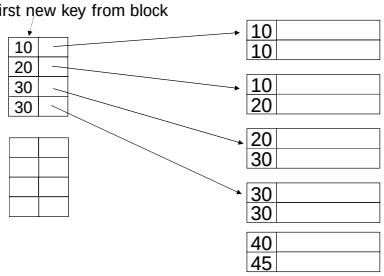
Sparse index, one way?



Duplicate keys

Sparse index, another way?

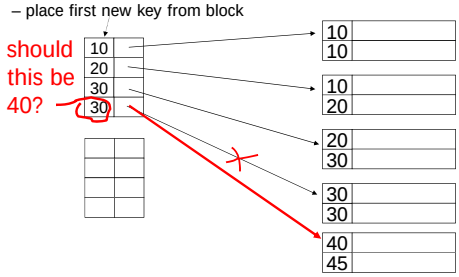
– place first new key from block



Duplicate keys

Sparse index, another way?

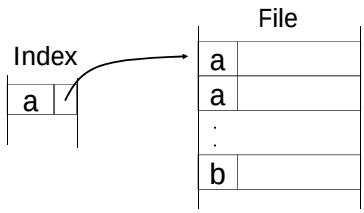
– place first new key from block



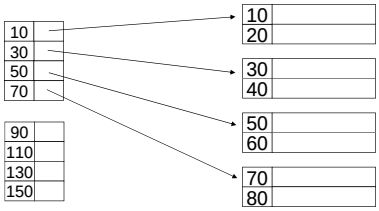
Summary

Duplicate values, primary index

- Index may point to first instance of each value only

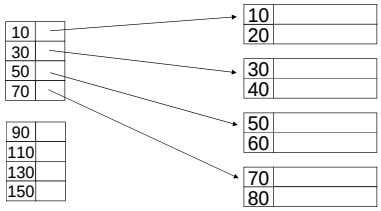


Deletion from sparse index



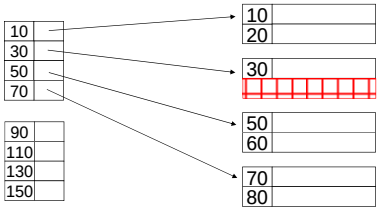
Deletion from sparse index

– delete record 40



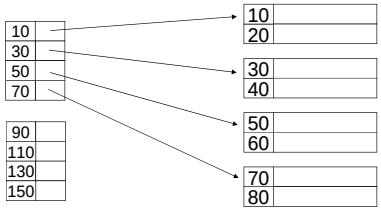
Deletion from sparse index

– delete record 40



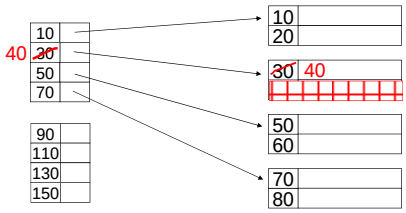
Deletion from sparse index

– delete record 30



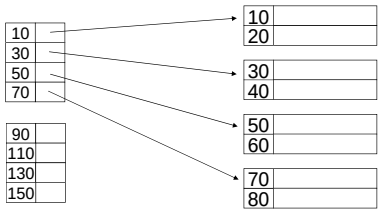
Deletion from sparse index

– delete record 30



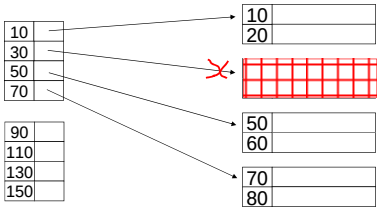
Deletion from sparse index

– delete records 30 & 40



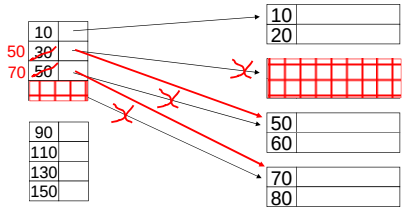
Deletion from sparse index

– delete records 30 & 40

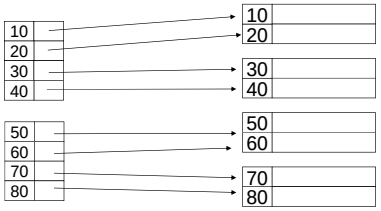


Deletion from sparse index

– delete records 30 & 40

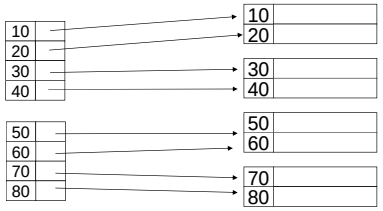


Deletion from dense index



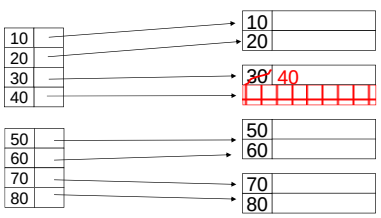
Deletion from dense index

– delete record 30



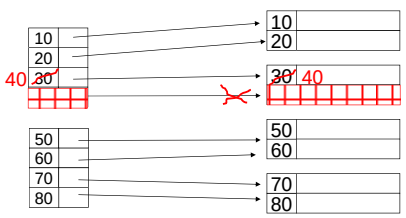
Deletion from dense index

– delete record 30

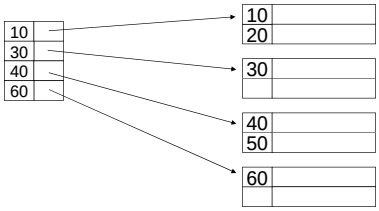


Deletion from dense index

– delete record 30

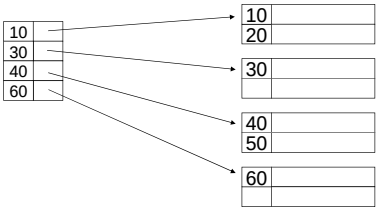


Insertion, sparse index case



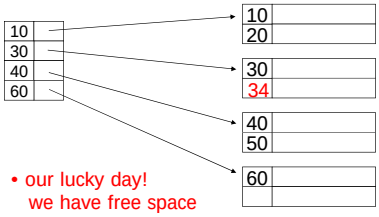
Insertion, sparse index case

– insert record 34



Insertion, sparse index case

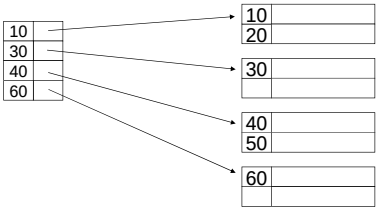
– insert record 34



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

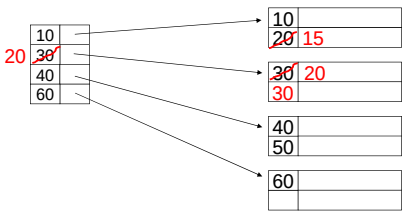
Insertion, sparse index case

– insert record 15



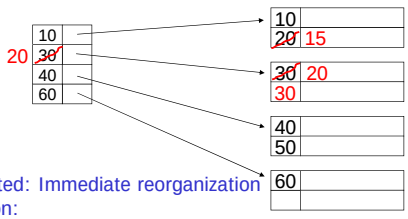
Insertion, sparse index case

– insert record 15



Insertion, sparse index case

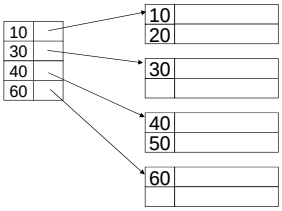
– insert record 15



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

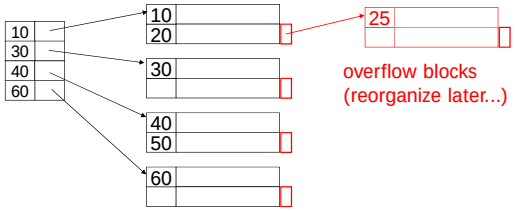
Insertion, sparse index case

– insert record 25



Insertion, sparse index case

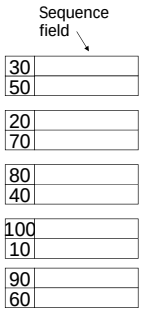
– insert record 25



Insertion, dense index case

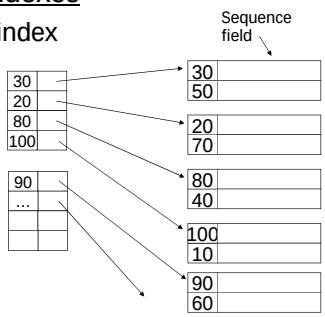
- Similar
- Often more expensive . . .

Secondary indexes



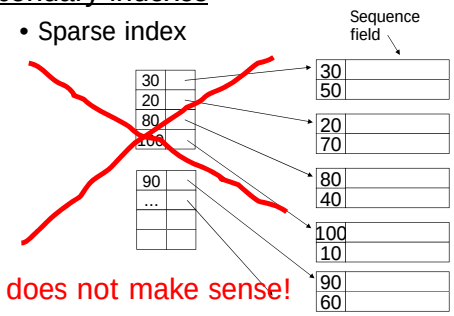
Secondary indexes

- Sparse index



Secondary indexes

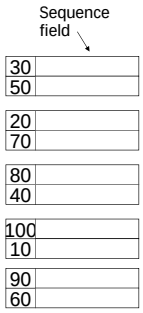
- Sparse index



does not make sense!

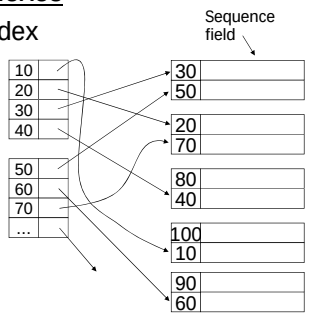
Secondary indexes

- Dense index



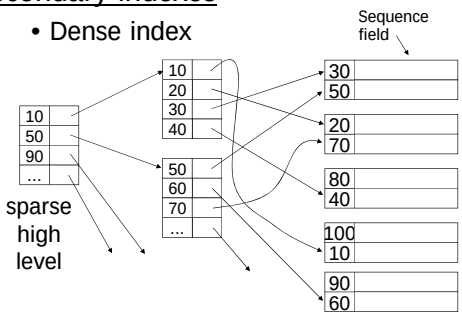
Secondary indexes

- Dense index



Secondary indexes

- Dense index



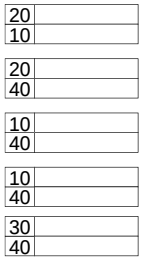
With secondary indexes:

- Lowest level is dense
- Other levels are sparse

Also: Pointers are record pointers

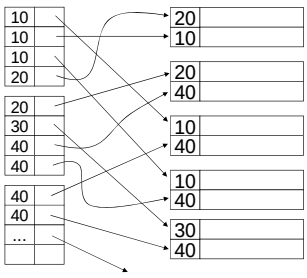
(not block pointers; not computed)

Duplicate values & secondary indexes



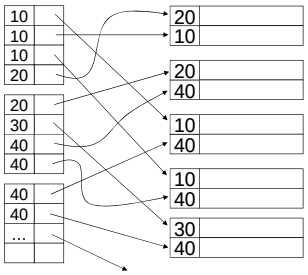
Duplicate values & secondary indexes

one option...



Duplicate values & secondary indexes

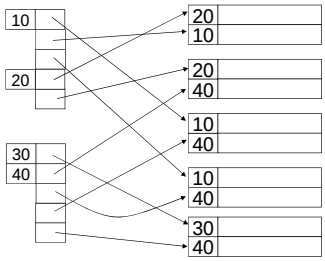
one option...



Problem:
excess overhead!
- disk space
- search time

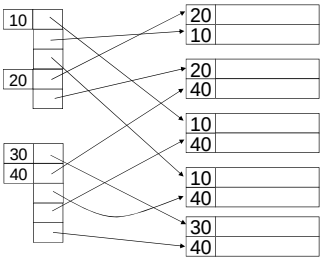
Duplicate values & secondary indexes

another option...



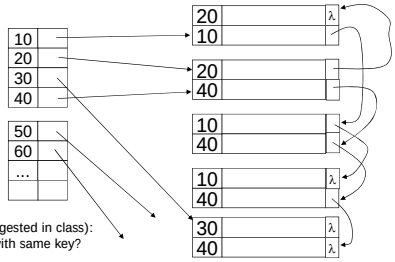
Duplicate values & secondary indexes

another option...



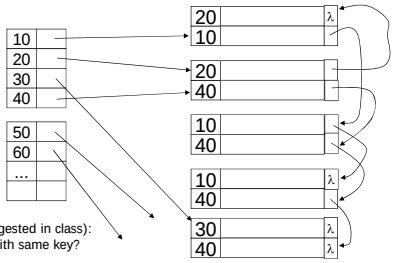
Problem:
variable size records in index!

Duplicate values & secondary indexes



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

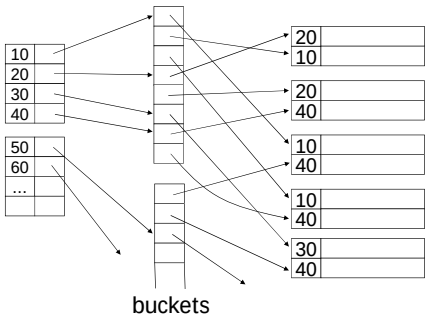
Duplicate values & secondary indexes



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

Problems:
- Need to add fields to records
- Need to follow chain to know records

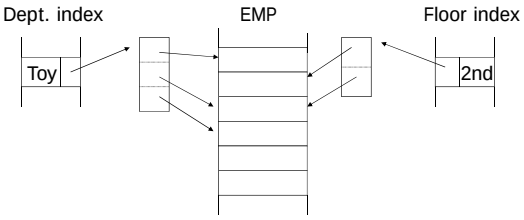
Duplicate values & secondary indexes



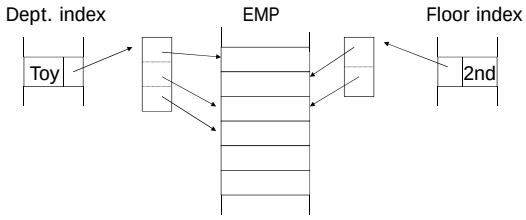
Why "bucket" idea is useful

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

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

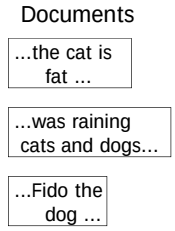


Query: Get employees in
(Toy Dept) ∧ (2nd floor)

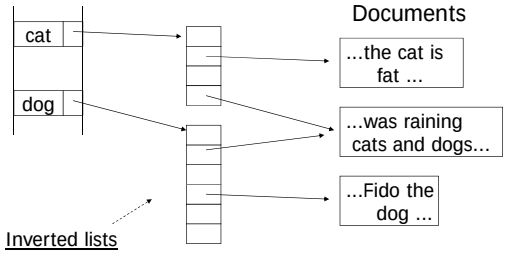


→ Intersect toy bucket and 2nd Floor bucket to get set of matching EMP's

This idea used in
text information retrieval



This idea used in
text information retrieval



IR QUERIES

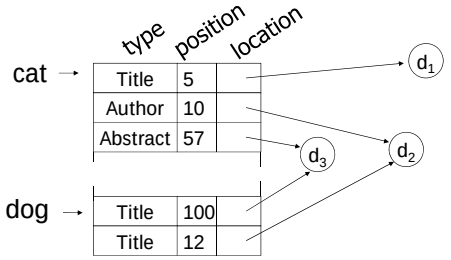
- Find articles with "cat" and "dog"
- Find articles with "cat" or "dog"
- Find articles with "cat" and not "dog"

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



Posting: an entry in inverted list. Represents occurrence of term in article

Size of a list: 1 Rare words or miss-spellings
(in postings)

10^6 Common words

Size of a posting: 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 ...>

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...

- How to process V.S. Queries?

	w1	w2	w3	w4	w5	w6	...
Q = <	0	0	0	1	1	0	... >

- Try Stanford Libraries
- Try Google, Yahoo, ...

Summary so far

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

Conventional indexes

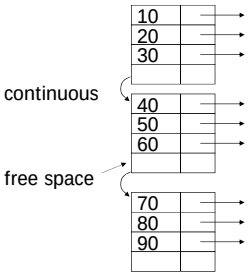
Advantage:

- Simple
- Index is sequential file
  good for scans

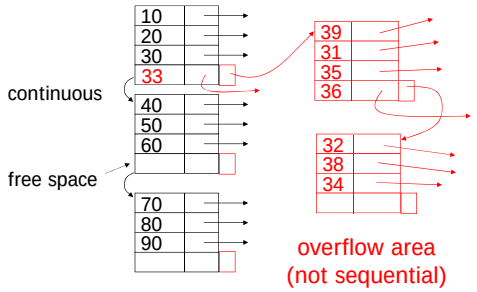
Disadvantage:

- Inserts expensive, and/or
- Lose sequentiality & balance

Example Index (sequential)



Example Index (sequential)



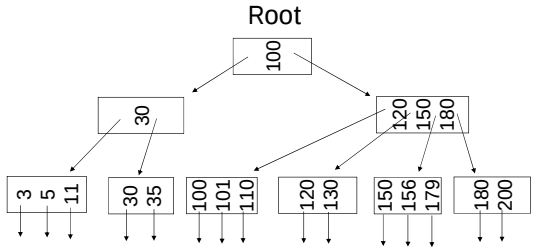
Outline:

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

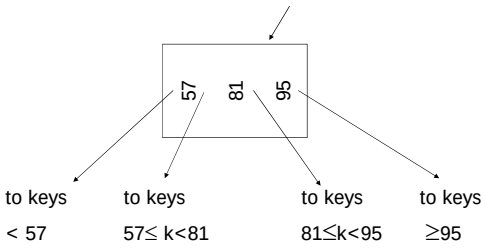
- NEXT: Another type of index
  - Give up on sequentiality of index
  - Try to get "balance"

Note: This index is called B+ tree,
but Gradiance homeworks just call it B-tree.

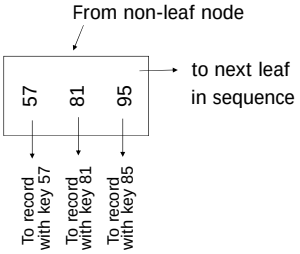
B+Tree Example n=3



Sample non-leaf



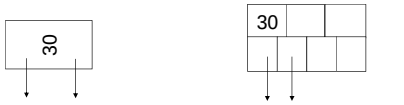
Sample leaf node:



In textbook's notation n=3

Leaf:

Non-leaf:



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

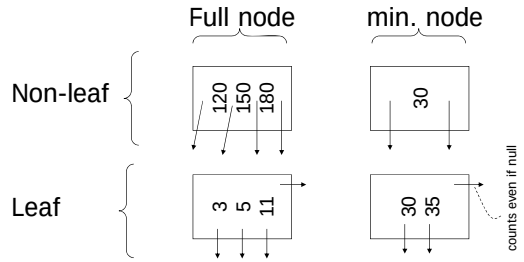
Don't want nodes to be too empty

- 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

Insert into B+tree

(a) simple case
- space available in leaf

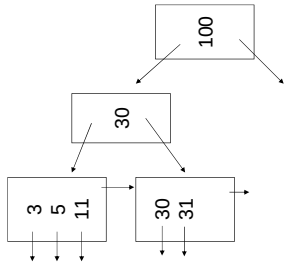
(b) leaf overflow

(c) non-leaf overflow

(d) new root

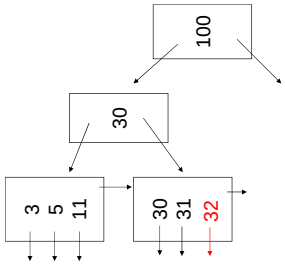
(a) Insert key = 32

n=3



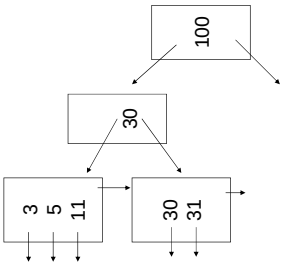
(a) Insert key = 32

n=3



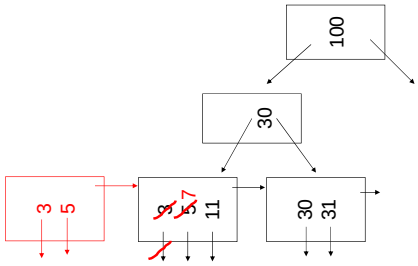
(a) Insert key = 7

n=3



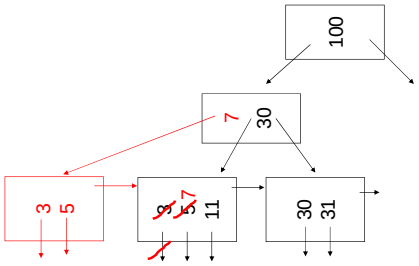
(a) Insert key = 7

n=3

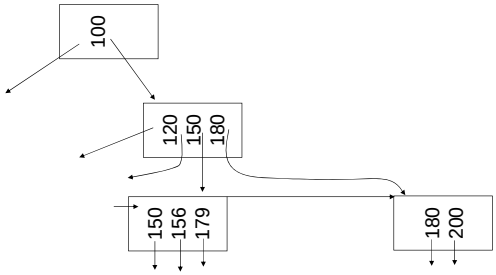


(a) Insert key = 7

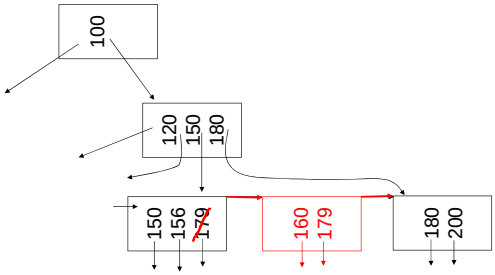
n=3



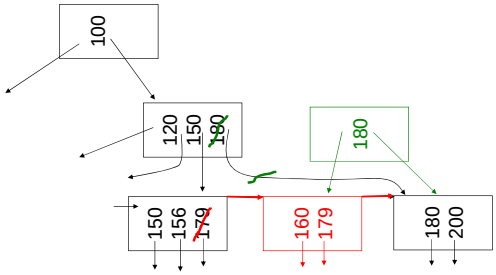
(c) Insert key = 160 n=3



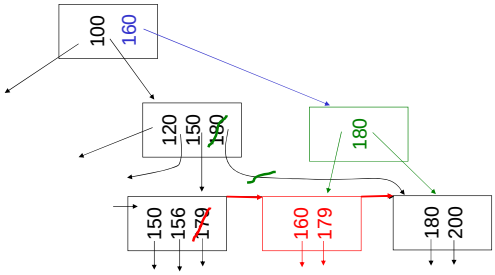
(c) Insert key = 160 n=3



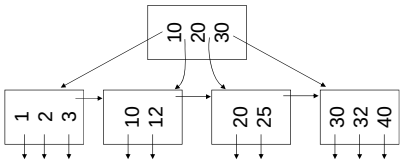
(c) Insert key = 160 n=3



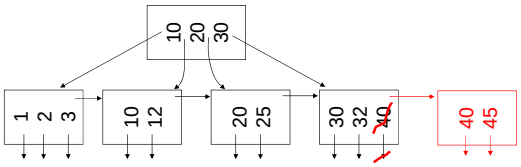
(c) Insert key = 160 n=3



(d) New root, insert 45 n=3

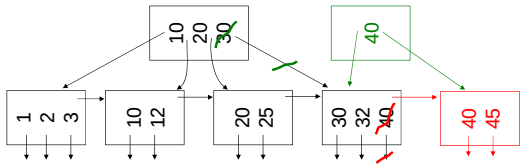


(d) New root, insert 45 n=3



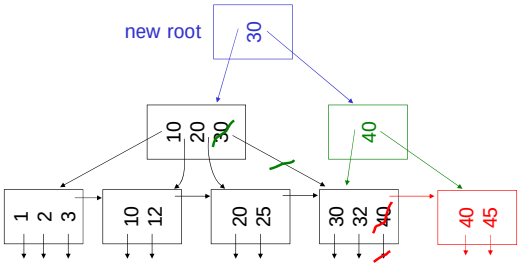
(d) New root, insert 45

n=3



(d) New root, insert 45

n=3

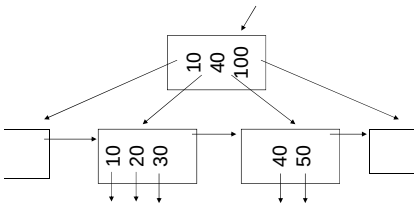


Deletion from B+tree

(a) Simple case - no example
(b) Coalesce with neighbor (sibling)
(c) Re-distribute keys
(d) Cases (b) or (c) at non-leaf

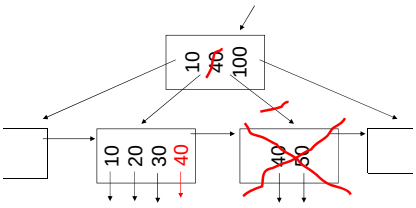
(b) Coalesce with sibling
– Delete 50

n=4



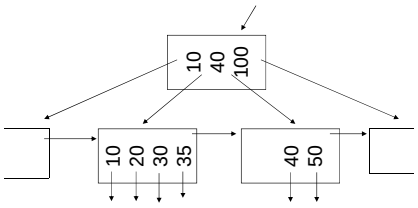
(b) Coalesce with sibling
– Delete 50

n=4



(c) Redistribute keys
– Delete 50

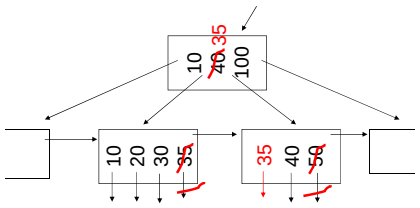
n=4



(c) Redistribute keys

– Delete 50

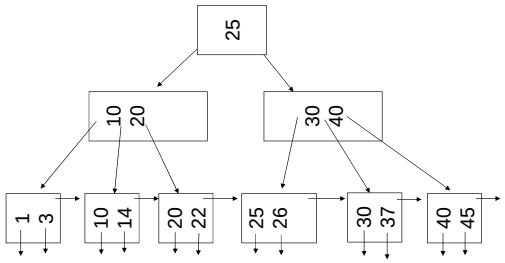
n=4



(d) Non-leaf coalesce

– Delete 37

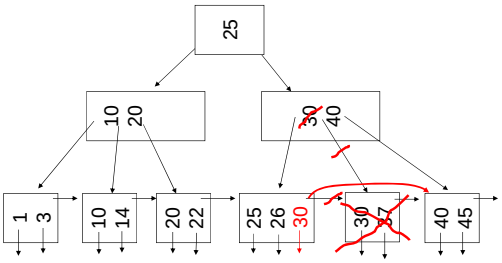
n=4



(d) Non-leaf coalesce

– Delete 37

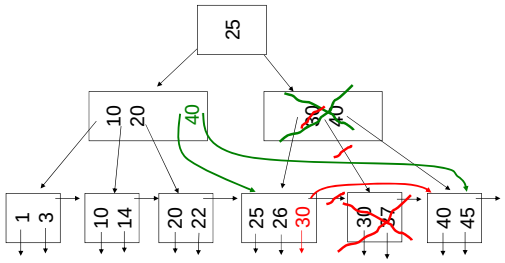
n=4



(d) Non-leaf coalesce

– Delete 37

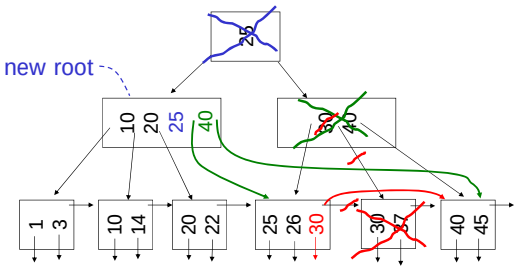
n=4



(d) Non-leaf coalesce

– Delete 37

n=4



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

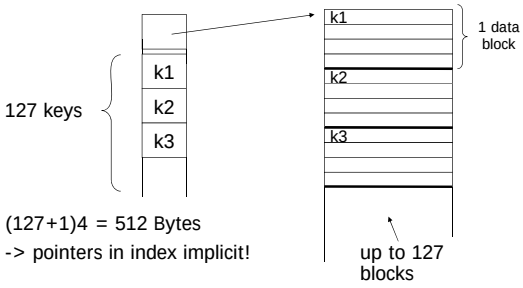
Ref # 1 claims:

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

For their comparison:

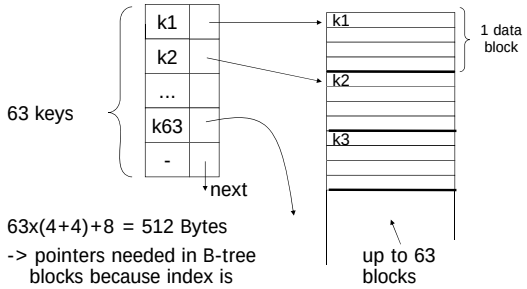
block = 512 bytes
key = pointer = 4 bytes
4 data records per block

Example: 1 block static index



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

Example: 1 block B-tree



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

Size comparison Ref. #1

Static Index		B-tree	
# data blocks	height	# data blocks	height
2 -> 127	2	2 -> 63	2
128 -> 16,129	3	64 -> 3968	3
16,130 -> 2,048,383	4	3969 -> 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

  $\Rightarrow$ Static index saves enough accesses to allow for reorganization

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, "Retrospection 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?

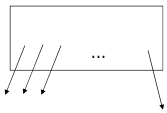
- Speaking of buffering...
  Is LRU a good policy for B+tree buffers?

→ 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+T*n*) msec.

Sample assumptions:

(1) Time to read node from disk is (S+T*n*) msec.

(2) Once block in memory, use binary search to locate key:

($a + b \log_2 n$) msec.

For some constants *a,b;* Assume a << S

Sample assumptions:

(1) Time to read node from disk is (S+T*n*) msec.

(2) Once block in memory, use binary search to locate key:

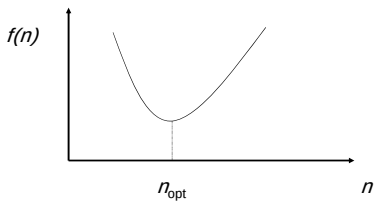
($a + b \log_2 n$) msec.

For some constants *a,b;* Assume a << S

(3) Assume B+tree is full, i.e., # nodes to examine is $\log_n N$ where N = # records

⇘Can get:

f(n) = time to find a record



⇘ FIND n_{opt} by $f'(n) = 0$

Answer is n_{opt} = "few hundred" (see homework for details)

- FIND n_{opt} by $f'(n) = 0$

  Answer is n_{opt} = "few hundred"
  (see homework for details)

- What happens to n_{opt} as
  - Disk gets faster?
  - CPU get faster?

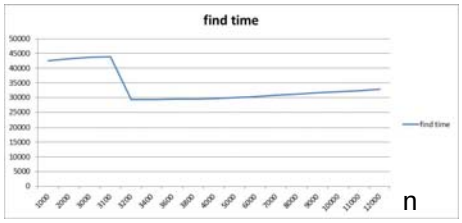
Exercise

S=	14000
T=	0.2
b=	0.002
a=	0
N=	10000000

- $f(n) = \log_n N*[S+T*n+a+b*\log_2(n)]$

N=10 million records

S=	14000
T=	0.2
b=	0.002
a=	0
N=	10000000

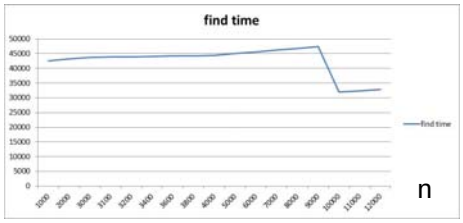


n

times in microseconds

N=100 million records

S=	14000
T=	0.2
b=	0.002
a=	0
N=	10000000

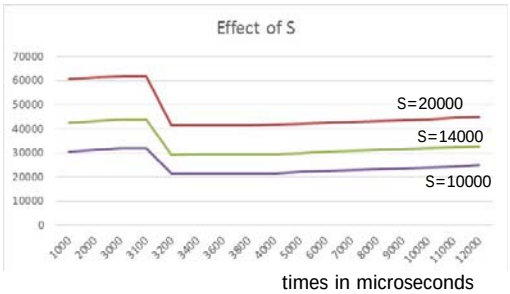


n

times in microseconds

N=100 million records

S=	varies
T=	0.2
b=	0.002
a=	0
N=	10000000

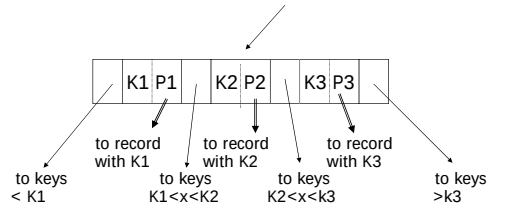


n

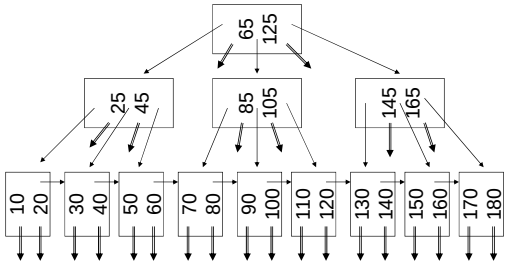
times in microseconds

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

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

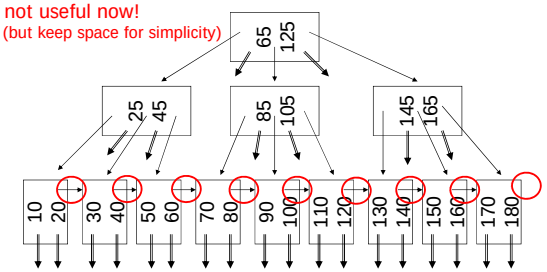


B-tree example n=2



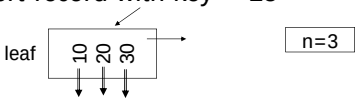
B-tree example n=2

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



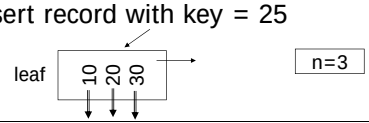
Note on inserts

- Say we insert record with key = 25

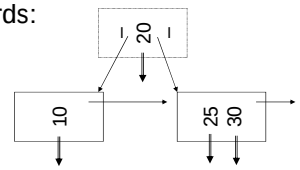


Note on inserts

- 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	$\lfloor n/2 \rfloor$	$\lfloor n/2 \rfloor$
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
☹ in B-tree, deletion more complicated

Tradeoffs:

☺ B-trees have faster lookup than B+trees

☹ in B-tree, non-leaf & leaf different sizes
☹ 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

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

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

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

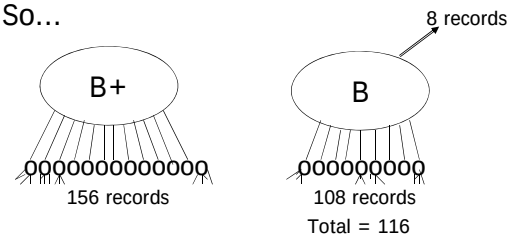
B+tree:

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

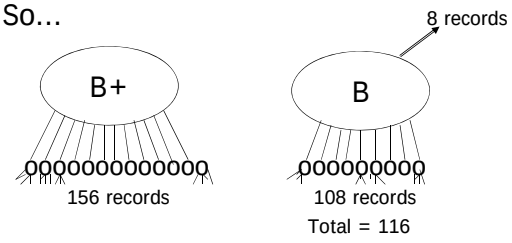
Each of 13 sons: 12 rec. ptrs (+12 keys)
= 12x(4 +4) + 4 = 100 bytes

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

So…



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

Outline/summary

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