B+ Tree's: Primary method for managing data on disks (in RDBMS)

Objective: Understand the implications disk access on the design of data structures.

- fat fanout
- split/promote

Reading:

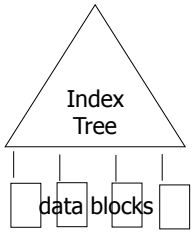
- Text Ch. 14
- For very helpful interactive demo, see

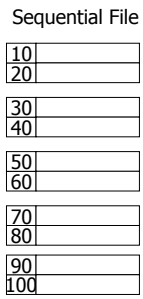
  https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html

Slide thanks: many slides, Garcia-Molina, (but you may want to see his original set), http://infolab.stanford.edu/~hector/cs245/notes.htm

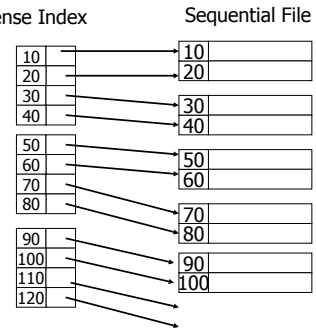
Data records (rows) stored in one file

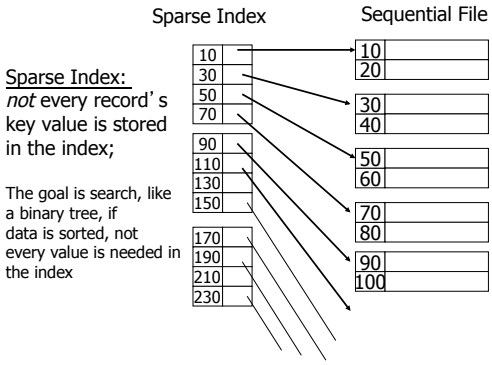
Index records stored separately.





Dense Index:
every record's key value is stored in the index





Sparse Index:
not every record's key value is stored in the index;

The goal is search, like a binary tree, if data is sorted, not every value is needed in the index

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

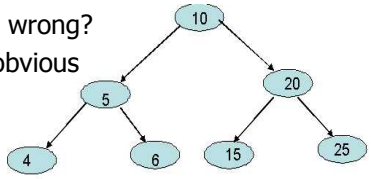
The B+ Tree

- Main memory binary trees evolved to disk.
- First there were B Trees
- Then B+ Trees.
  - So superior, B Trees are gone,
    - So gone, we say B Trees and mean B+ Tree

Naïve Use of Binary Tree on Disk

One node per disk block

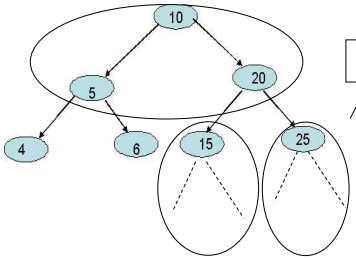
- What goes wrong?
- What are obvious fixes?

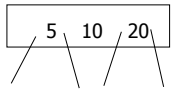


A simple binary tree

Fatten the Nodes

(e.g. not a B+ node yet)





Disk access?

But these are integers what about records?

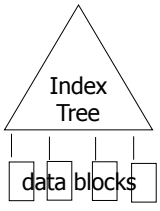
Do better?

In databases

- We like algorithmic complexity (why?)
- But we don't really get to throw away the constants, etc.

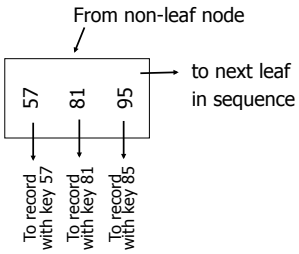
In the index, store only search keys, not the records

- Fatter fanout - more space for *split* keys
- All the data is in the data file.
- (if dense index, all keys are in the index)

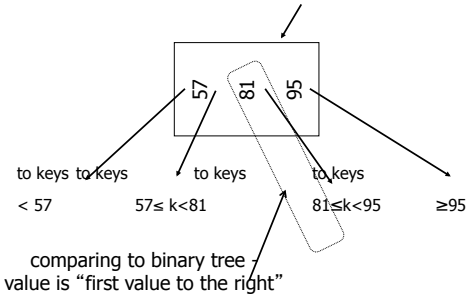


This is the primary difference between, B and B+ trees

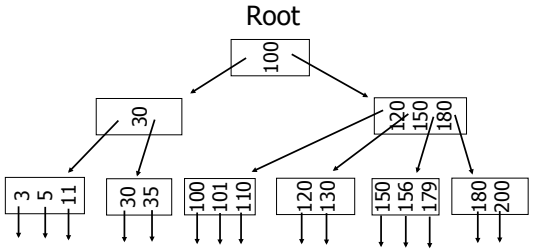
Sample (dense) leaf node:



Sample non-leaf



B+Tree Example n=3



Size of nodes:

n+1 pointers
n keys
(fixed)

In textbook's notation n=3

Leaf:

Non-leaf:

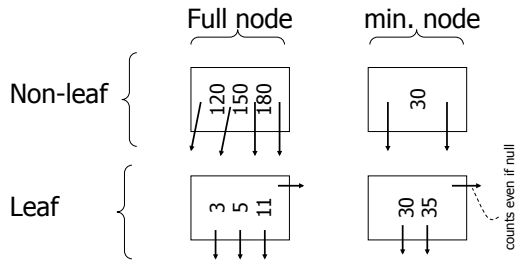
Don't want nodes to be too empty

- 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

(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

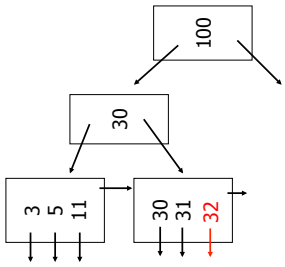
(b) leaf overflow

(c) non-leaf overflow

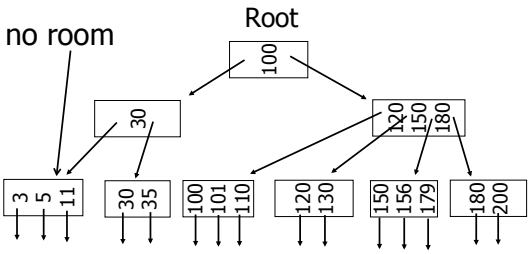
(d) new root

(a) Insert key = 32

n=3

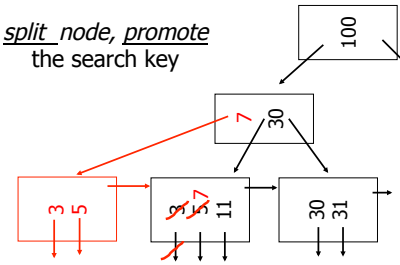


(b) Insert key = 7



(b) Insert key = 7 n=3

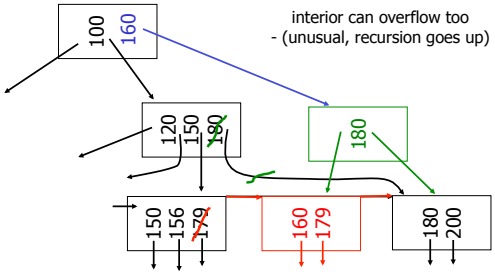
split node, promote the search key



6: B+trees Data Engineering 25

(c) Insert key = 160 n=3

interior can overflow too
- (unusual, recursion goes up)

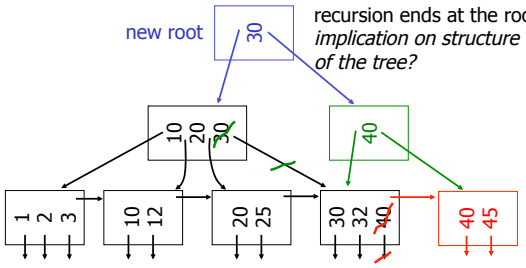


6: B+trees Data Engineering 26

(d) New root, insert 45 n=3

new root

recursion ends at the root
implication on structure of the tree?



6: B+trees Data Engineering 27

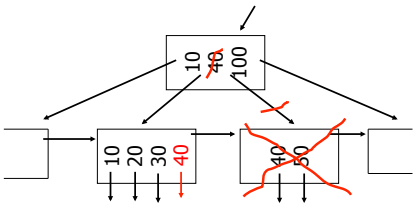
Deletion from B+tree

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

6: B+trees Data Engineering 28

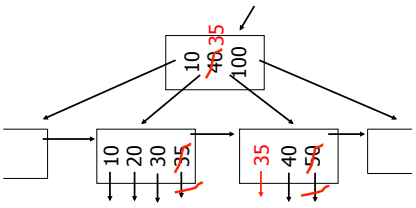
(b) Coalesce with sibling
– Delete 50

n=4



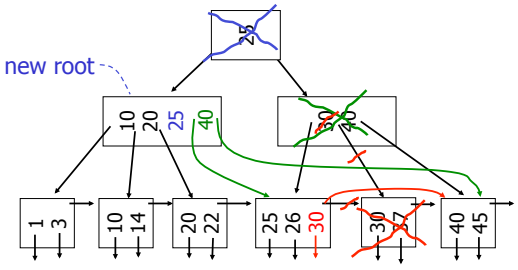
(c) Redistribute keys
– Delete 50

n=4



(d) Non-leaf coalesce
– Delete 37

n=4



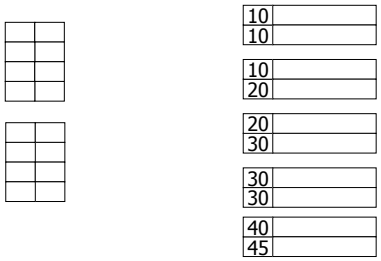
B+tree deletions in practice

– Often, coalescing is not implemented
  – Overhead not worth it! (why?)

Next:

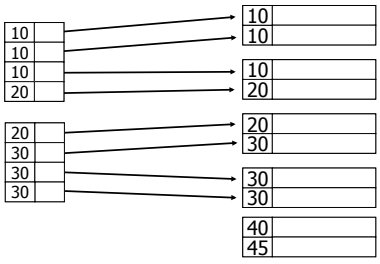
- Duplicate keys
- Deletion/Insertion
- Secondary indexes

Duplicate keys



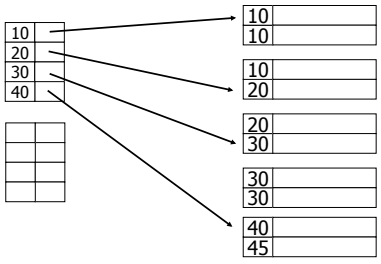
Duplicate keys

Dense index, one way to implement?



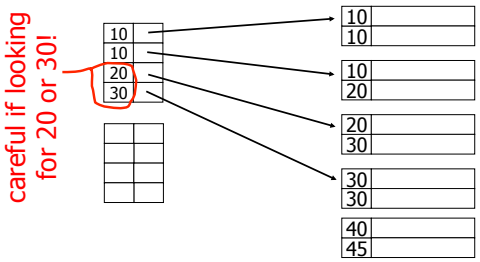
Duplicate keys

Dense index, better way?



Duplicate keys

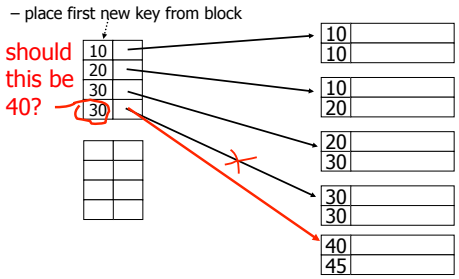
Sparse index, one way?



Duplicate keys

Sparse index, another way?

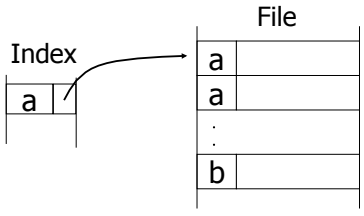
– place first new key from block



Summary

Duplicate values, primary index

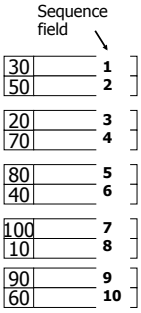
- Index may point to first instance of each value only



Primary vs. Secondary Index

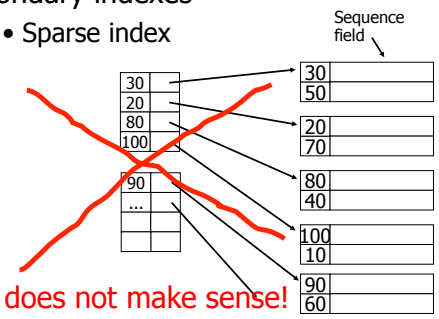
- Primary is sorted on the index key *and placed on disk that way*
- Secondary index,
  - does not "cluster"
  - consider (x, y)

Secondary indexes



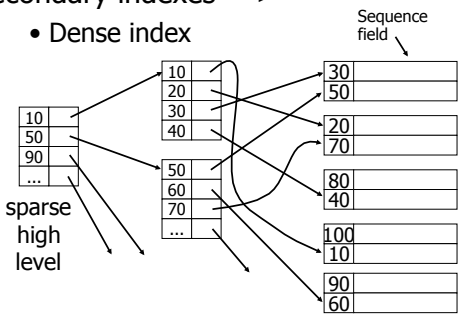
Secondary indexes

- Sparse index



Secondary indexes -->

- Dense index



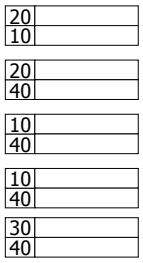
With secondary indexes:

- Lowest level is dense
- Other levels are sparse

Also: Pointers are record pointers

(not block pointers; not computed)

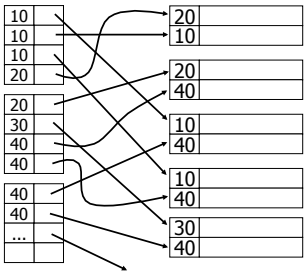
Duplicate values & secondary indexes



Duplicate values & secondary indexes

one option...

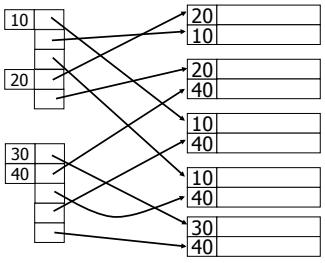
Problem:
excess overhead!
- disk space
- search time



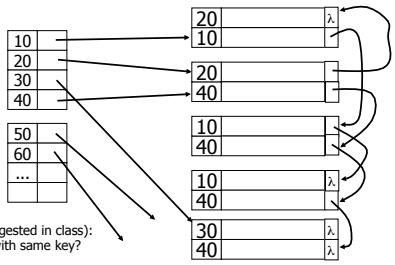
Duplicate values & secondary indexes

another option...

Problem:
variable size records in index!



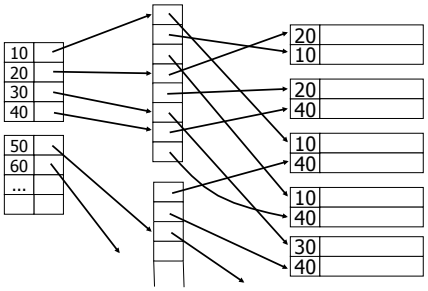
Duplicate values & secondary indexes



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

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

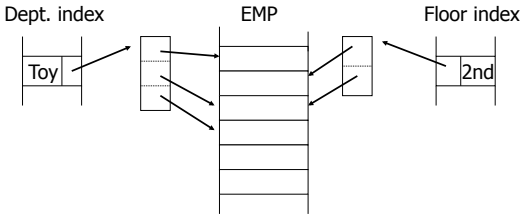
Duplicate values & secondary indexes



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)



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

This idea used in text information retrieval