Indexes

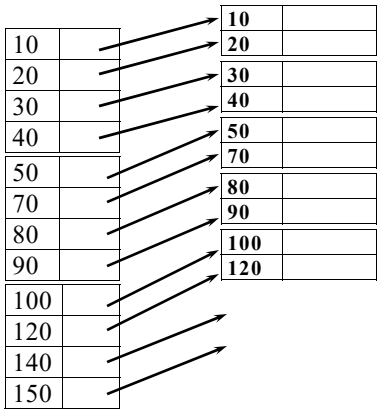
- Data structures used for quickly locating tuples that meet a specific type of condition
  - ***Equality*** condition:
    - *find Movie tuples where Director=Bertolucci*
  - Other conditions possible, e.g., ***range*** conditions:
    - *find Employee tuples where Salary>40 AND Salary<50*
- Many types of indexes. Evaluate them on
  - *Access* time
  - *Insertion* time
  - *Deletion* time
  - Disk *Space* needed

Basic notions

- ***Primary*** index
  - the index on the attribute (a.k.a. search key) that ***determines the sequencing*** of the table on disk
- ***Secondary*** index
  - index on any other attribute
- ***Dense*** index
  - every value of the indexed attribute appears in the index
- ***Sparse*** index
  - many values do not appear

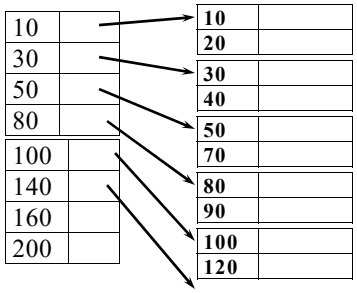
Dense and Sparse Primary Indexes

Dense Primary Index



Sparse Primary Index

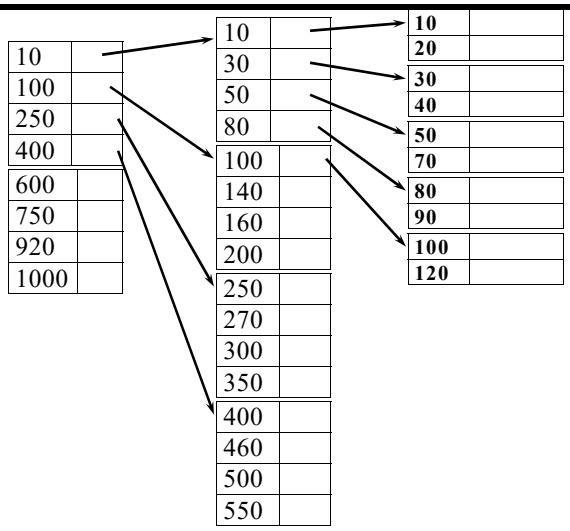
(e.g., one pointer into each data block)



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

Multi-Level Indexes

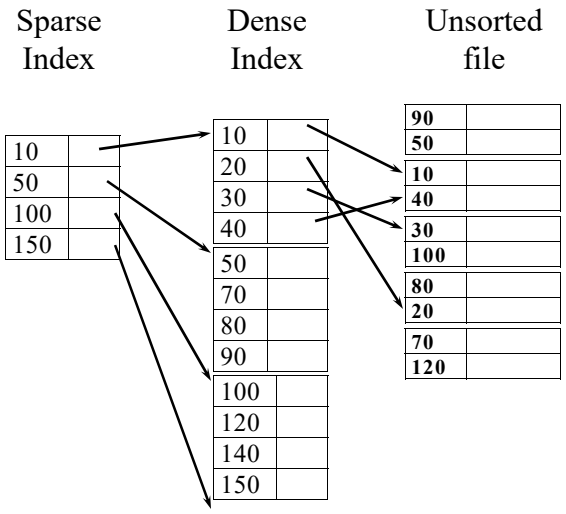
- Treat the index as data and build an index on it
- Two levels are usually sufficient. More than three levels are rare (use B-Trees instead)
- Q: Can we build a **dense second level** index for a dense index ?



37

Secondary Indexes

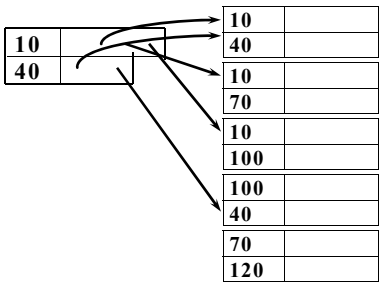
- The file is not sorted according to the secondary search key
- ***secondary index*** has to be dense
- a ***second level*** index on that one would be ***sparse***



38

Duplicate Values and Secondary Indexes

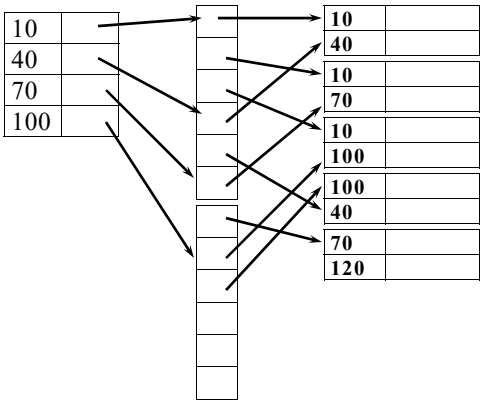
- store together all pointers with the same search key value



39

Duplicate Values and Secondary Indexes: Buckets

- store together all pointers with the same search key value
- introduce a separate level of buckets
  - if many pointers for each search key value it is better to separate the pointers from the values

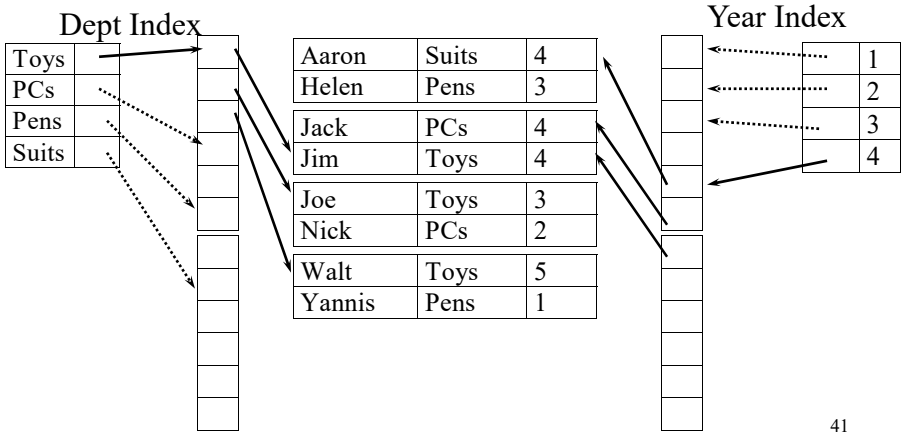


40

Advantage of Buckets: Process Queries Using Pointers Only

Find employees of the Toys dept with 4 years in the company

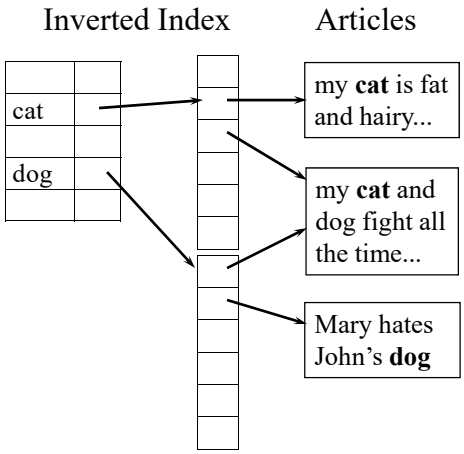
SELECT Name FROM Employee
WHERE Dept="Toys" AND Year=4



41

Buckets and Pointers Operation Used in Information Retrieval

- known as ***inverted index***
- an entry in an inverted list represents occurrence of a word in an article
- lists range from 1 to 1,000,000's of words
- compression also used



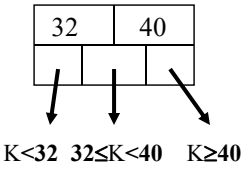
42

B+-Tree Indexes

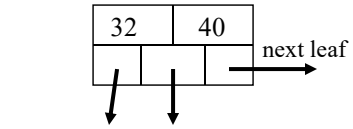
- Balanced trees
  (equal length paths from root to leaves)
- for minimizing disk I/O
- number of levels (logarithmic) automatically maintained w.r.t. size of the data file
- guaranteed upper limits on access, insert, delete times

Properties of B+Trees

- parameter ***n***: a node holds
  - *n* search key values (sorted) and
  - *n*+1 pointers (to interior nodes or records)
- left key <= pointed-to value < right key
- choose *n* so large that a node fits in a block
- interior node:
  - between half and all of the *n*+1 pointers are used
- leaf node:
  - rightmost pointer to the next leaf

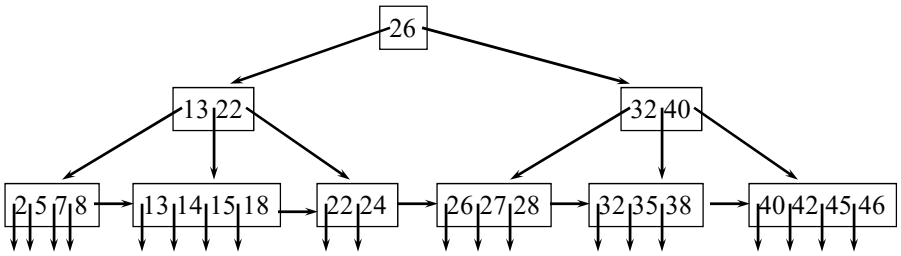


interior node for n=2



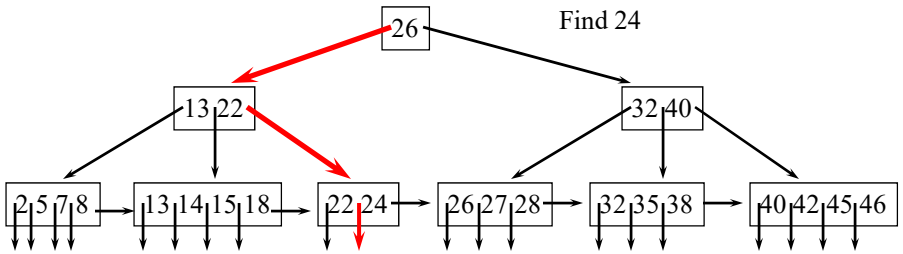
leaf node for n=2

Example B+Tree

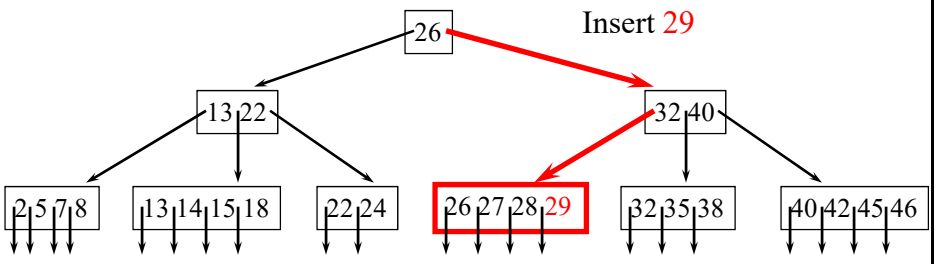


- if data file not sorted, then leaves have to constitute a dense index
- data file sorted, then leaves may constitute a sparse index

Lookup Algorithm

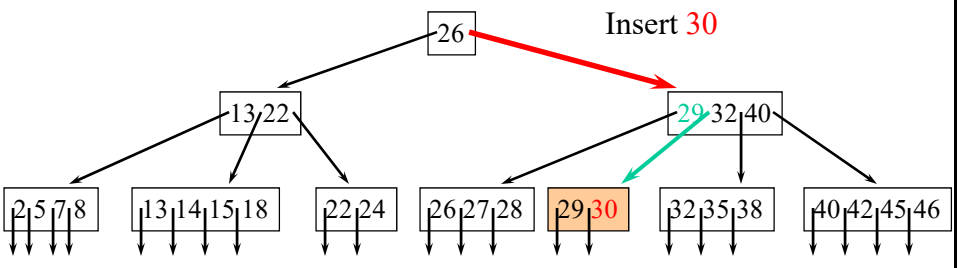


Insertion Algorithm

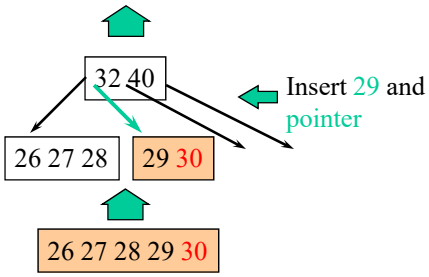


- first locate the leaf page where the item should appear
- if the leaf page is not full simply include item in the page

Insertion Algorithm: Splitting Nodes



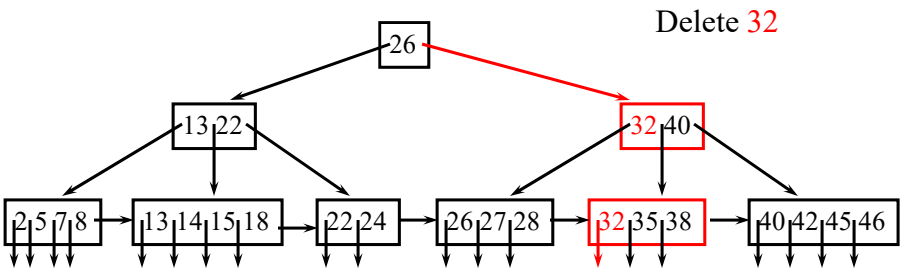
if the leaf page has *2m+1* items after the insertion then
- create a new page with *m* items
- insert the pointer of the new page and the first item in the parent directory



Insertions: Splitting Recursively

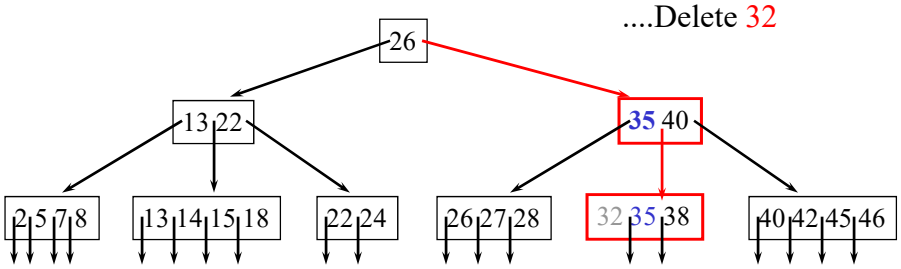
- splitting at one level can cause an insertion at the higher level: **recursively** apply procedure at the **higher level**
- when reaching the root an there is no more space, then **create a new level**

Deleting from B+ Trees



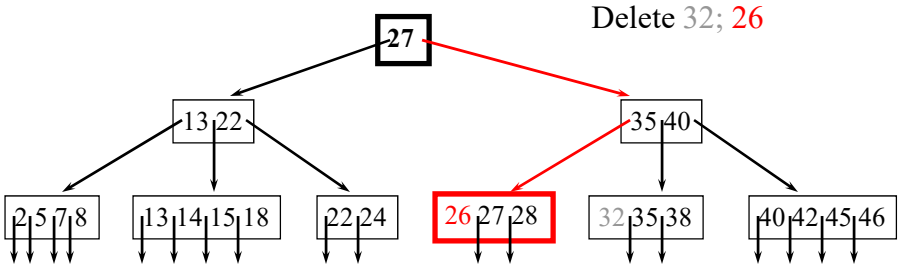
- locate the record
- delete the pointed-to record from the data file
- delete the key-pointer pair from the B+-tree ...

Deletion: The No-Combining Pages Case



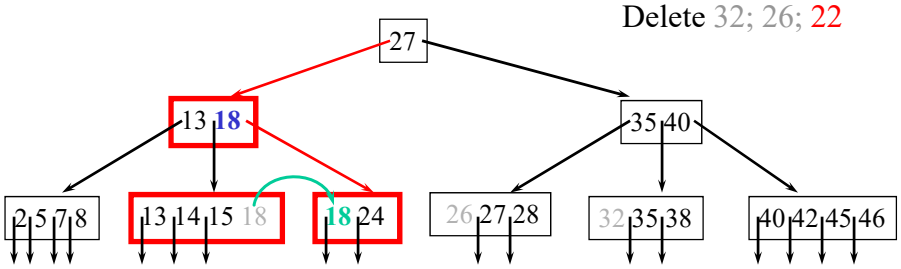
- recall that n=4, i.e., each **internal node** has **at least** m=2 keys and m+1= 3 pointers (at most 4 keys, 5 pointers)
- if the node from which was deleted is still **half full** (has m=2 keys)**:**
  - DONE (lookup still works), or
  - **update parent if deleted leftmost key**

Deletion: The No-Combining Pages Case



- if the node from which was deleted is still half full:
  - DONE, or
  - update parent if deleted leftmost key
- otherwise (Delete 22 ??)

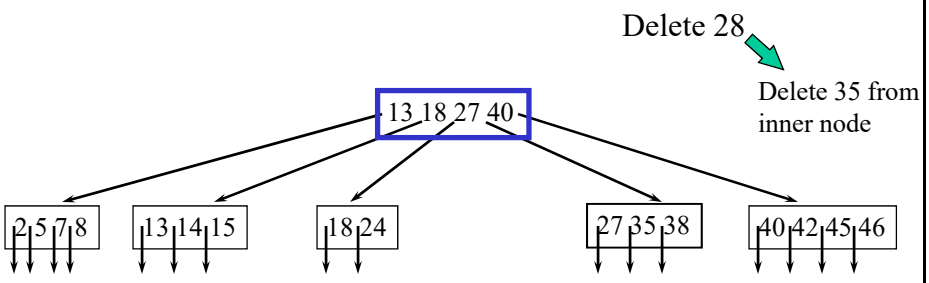
Deletion: Transferring Items From Siblings



- if the node N from which is deleted has minimum (m=2) items:
- if there is a neighbor N' (left or right)* with >m items then
  - transfer the first (or last) item of N' to N, and
  - update the appropriate ancestors of N
- else ... (Delete 28: next page)

* transfer the last element of the left neighbor or the first of the right neighbor

Deletion: Reducing Levels



When the root is left with two children a deletion may cause **removal of a level**

B+Tree Summary

- B+-tress automatically maintain as many index levels as appropriate (no overflow blocks necessary!)
- a node (block/page) holds up to *n* keys and *n*+1 pointers
- nodes are maintained to be between half-full and full
- range queries are supported

B+Tree Indexes in Practice

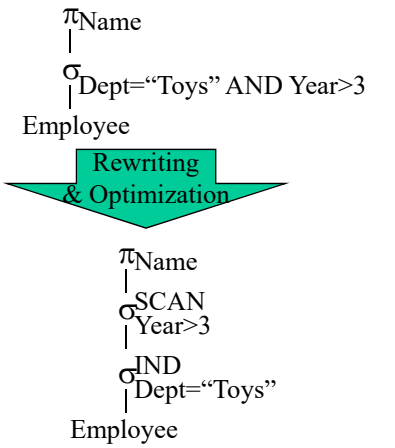
- The SQL standard does ***not*** talk about indexes!
- But every real DBMS allows statements like

```
CREATE INDEX IndAgeRating ON Students
    WITH STRUCTURE = BTREE,
    KEY = (age, gpa)
```

Multi-Key Indexing

- **Motivation**: queries of the form
  - SELECT … FROM R WHERE *cond1* and *cond2*
  - *cond1* and *cond2* are equality or range conditions
- **Solution 1**: use index for only one of the conditions
  - suggested if there is a very selective condition
- **Solution 2**: pointer intersection
  - fairly selective conditions

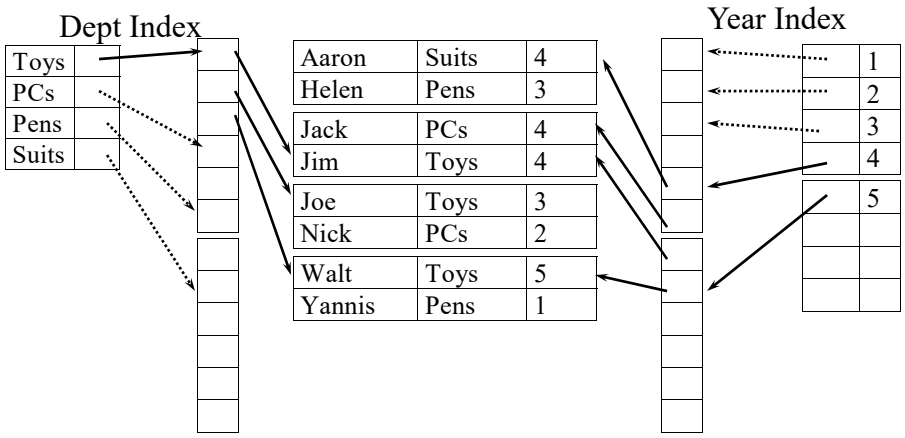
SELECT Name FROM Employee
WHERE Dept="Toys" AND Year > 3



Pointer Intersection (shown earlier)

Find employees of the Toys dept with >3 years in the company

```
SELECT Name FROM Employee
WHERE Dept="Toys" AND Year > 3
```



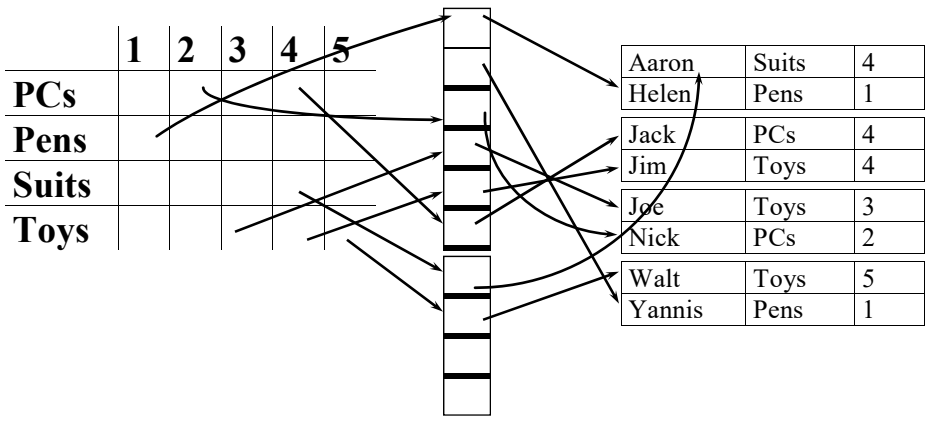
Solution 3: Multi-Key Indexing

- Appropriate when
  - each condition is not very selective
  - but their conjunction is very selective
- Brute force
- Grid structure

Common Applications of Multi-Key Indexing

- Geographic Data
  - find the city located at latitude 33, longitude 50
  - find cities in within … coordinates
- Many types of geographic index
  - R-trees: indexing of spatial objects
  - LSD trees: indexing of multidimensional points
  - k-d trees
- Similar indexing methods for multimedia queries
  - find *k* nearest neighbors

Grid Structure



- Space overhead (very sparse structure)
- Expensive insertion and deletion if new key values