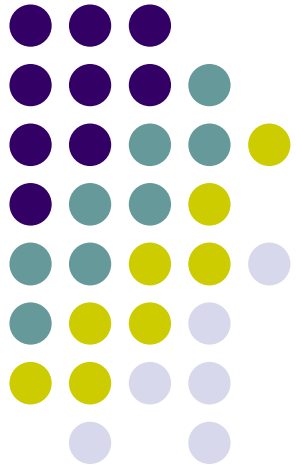
# Introduction to Indexes

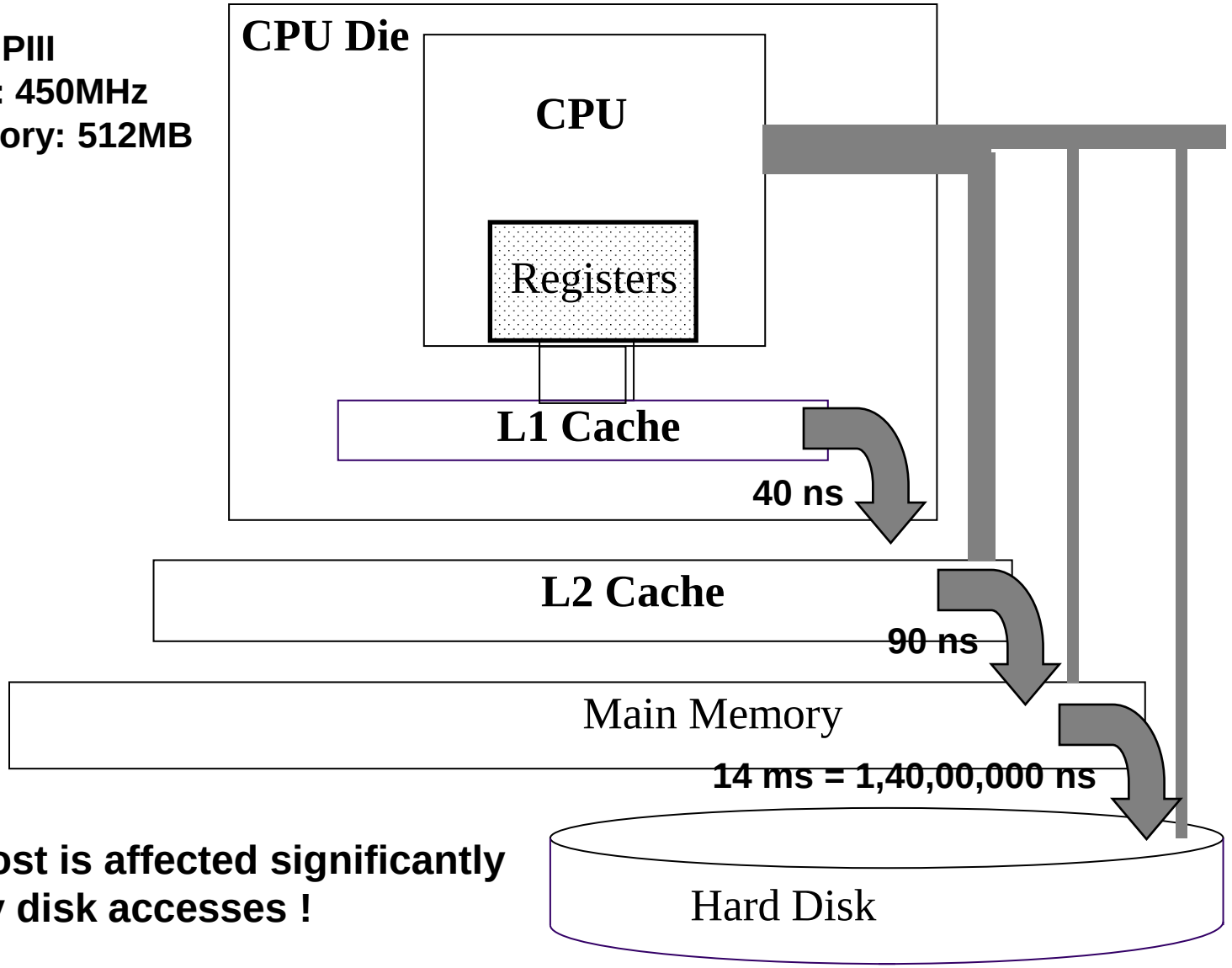


# Outline

- Cost Model and File Organization
  - The memory hierarchy
  - Magnetic disk
  - Cost model
  - Heap files
  - The concept of index
- Indexes : General Concepts and Properties
  - Queries, keys and search keys
  - Simple index files
  - Properties of indexes
  - Indexed sequential access method (ISAM)
- $B^+$-tree
  - Structure
  - Search
  - Insertion
  - Deletion
- Hash tables

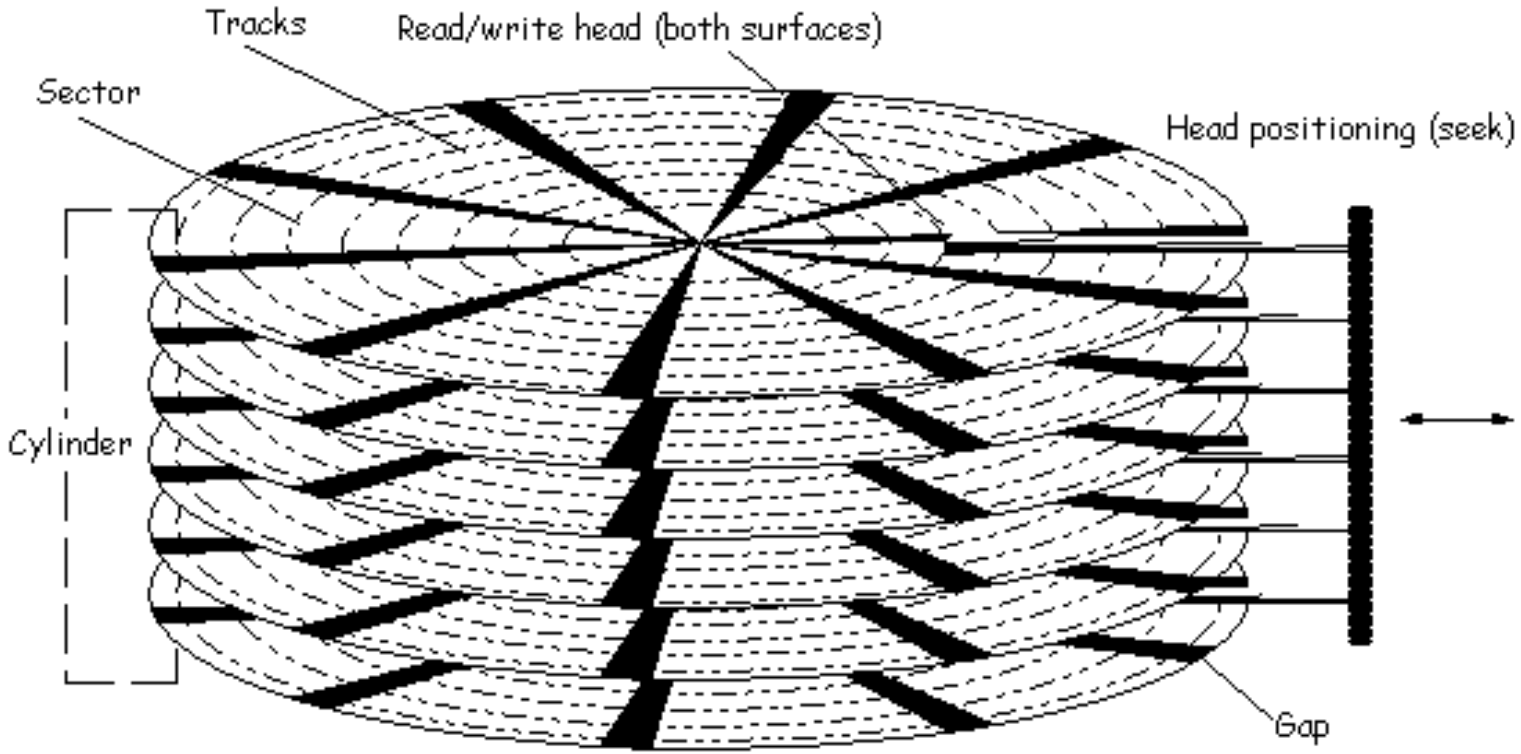
# The Memory Hierarchy

**Intel PIII**
**CPU: 450MHz**
**Memory: 512MB**



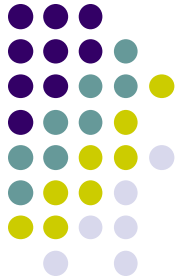

**Cost is affected significantly by disk accesses !**

# The Hard (Magnetic) disk




- The time for a ***disk block access***, or ***disk page access*** or ***disk I/O***
  *access time = seek time + rotational delay + transfer time*
- IBM Deskstar 14GPX: 14.4GB
  - Seek time: 9.1 msec
  - Rotational delay: 4.17 msec
  - Transfer rate: 13MB/sec, that is, 0.3msec/4KB

# The Cost Model

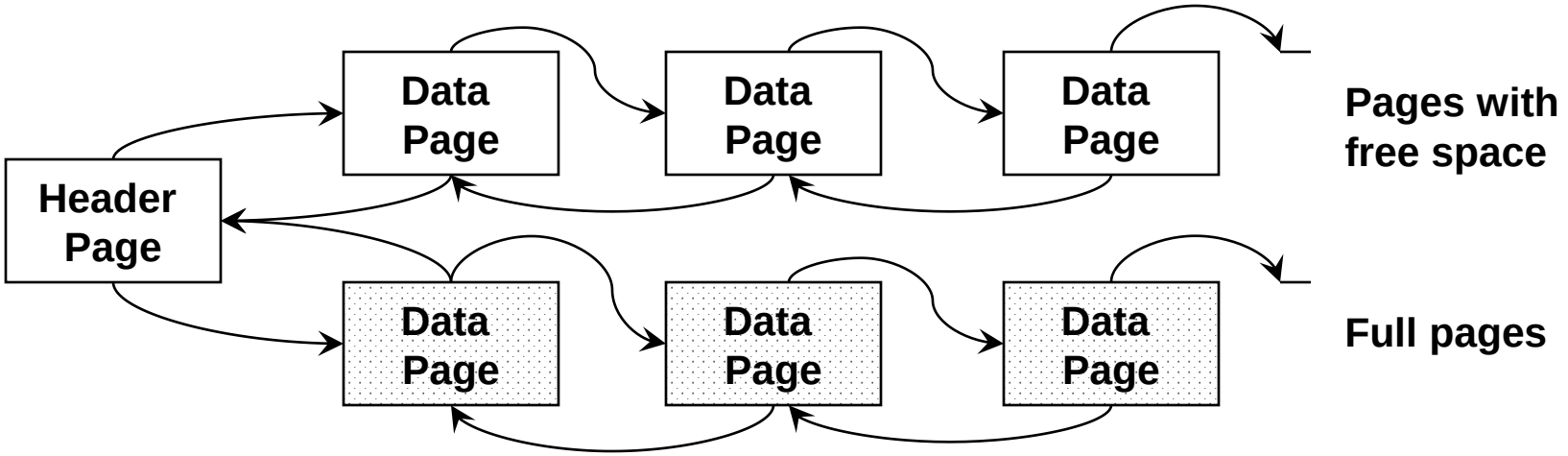


- Cost measure: *number of page accesses*
- Objective
  - A simple way to estimate the cost (in terms of execution time) of database operations
- Reason
  - Page access cost is usually the dominant cost of database operations
- Note
  - This cost model is for disk based databases; ***NOT*** applicable to main memory databases
  - Blocked access: sequential scan of the database

# Basic File Organization : Heap Files

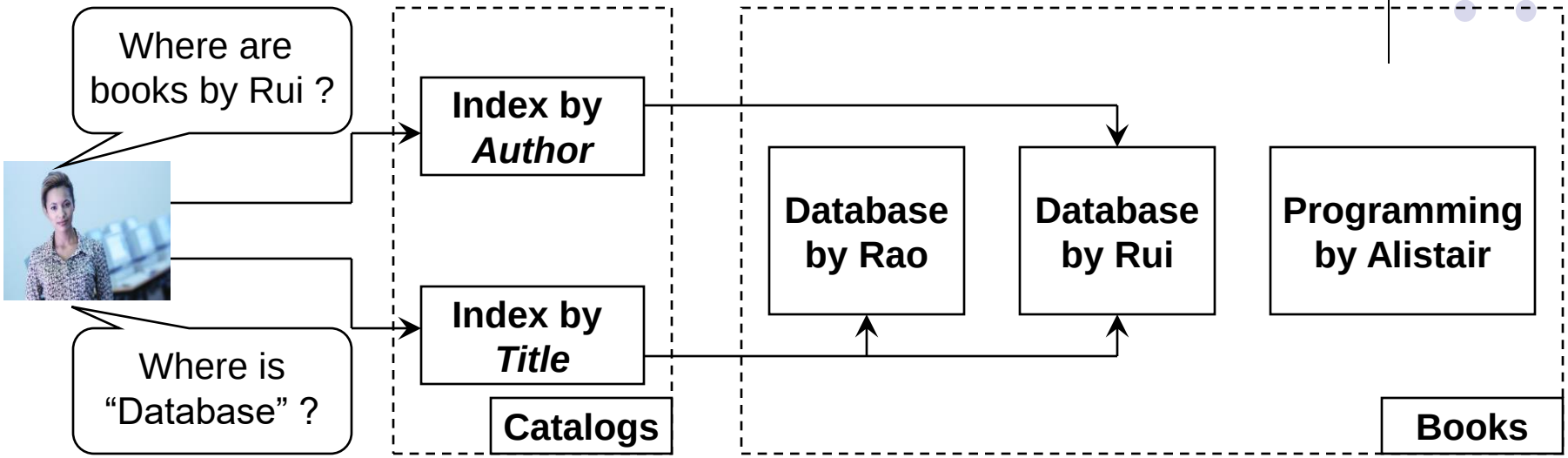
- **File** : a logical collection of data, physically stored as a set of pages.

- **Heap File** (Unordered File)
  - Linked list of pages
  - The DBMS maintains the header page: <*heap_file_name*, *header_page_address*>
  - Operations
    - Insertion
    - Deletion
    - Search
  - Advantage: Simple
  - Disadvantage: Inefficient



# The Concept of Index

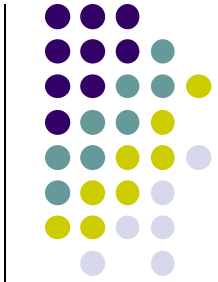
- Searching a Book…




- Index
  - A data structure that helps us find data quickly

- Note
  - Can be a separate structure (we may call it the index file) or in the records themselves.
  - Usually sorted on some attribute

# Query, Key and Search Key

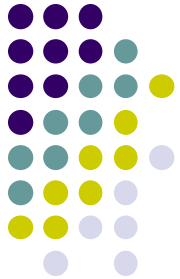


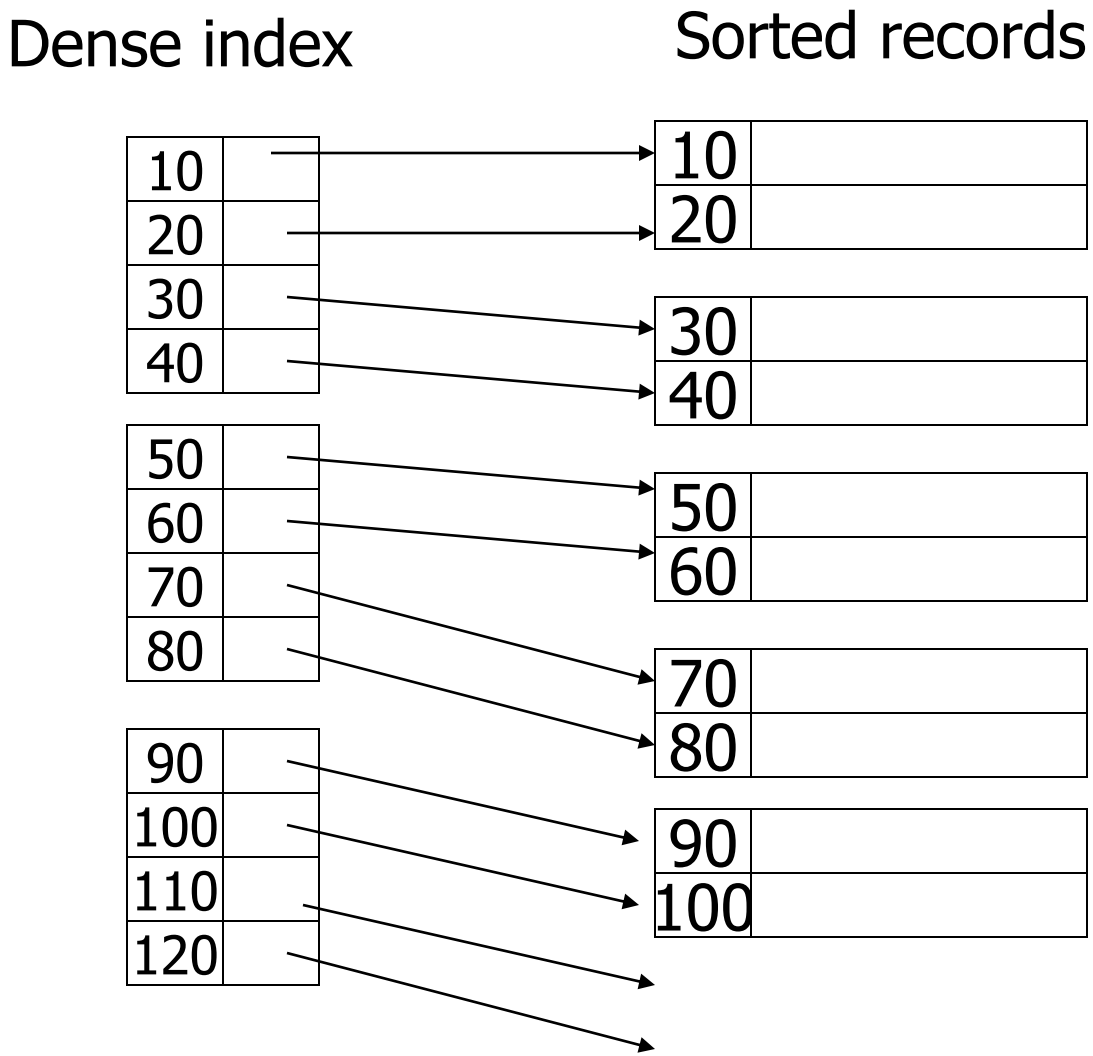
- Queries
  - Exact match (point query)
    - Q1: Find me the book with the name “Database”
  - Range query
    - Q2: Find me the books published between year 2003-2005

- Searching methods
  - Sequential scan — too expensive
  - Using index – if records are sorted on some attribute, we may do a binary search
    - If sorted on “book name”, then we can do binary search for Q1
    - If sorted on “year published”, then we can do binary search for Q2

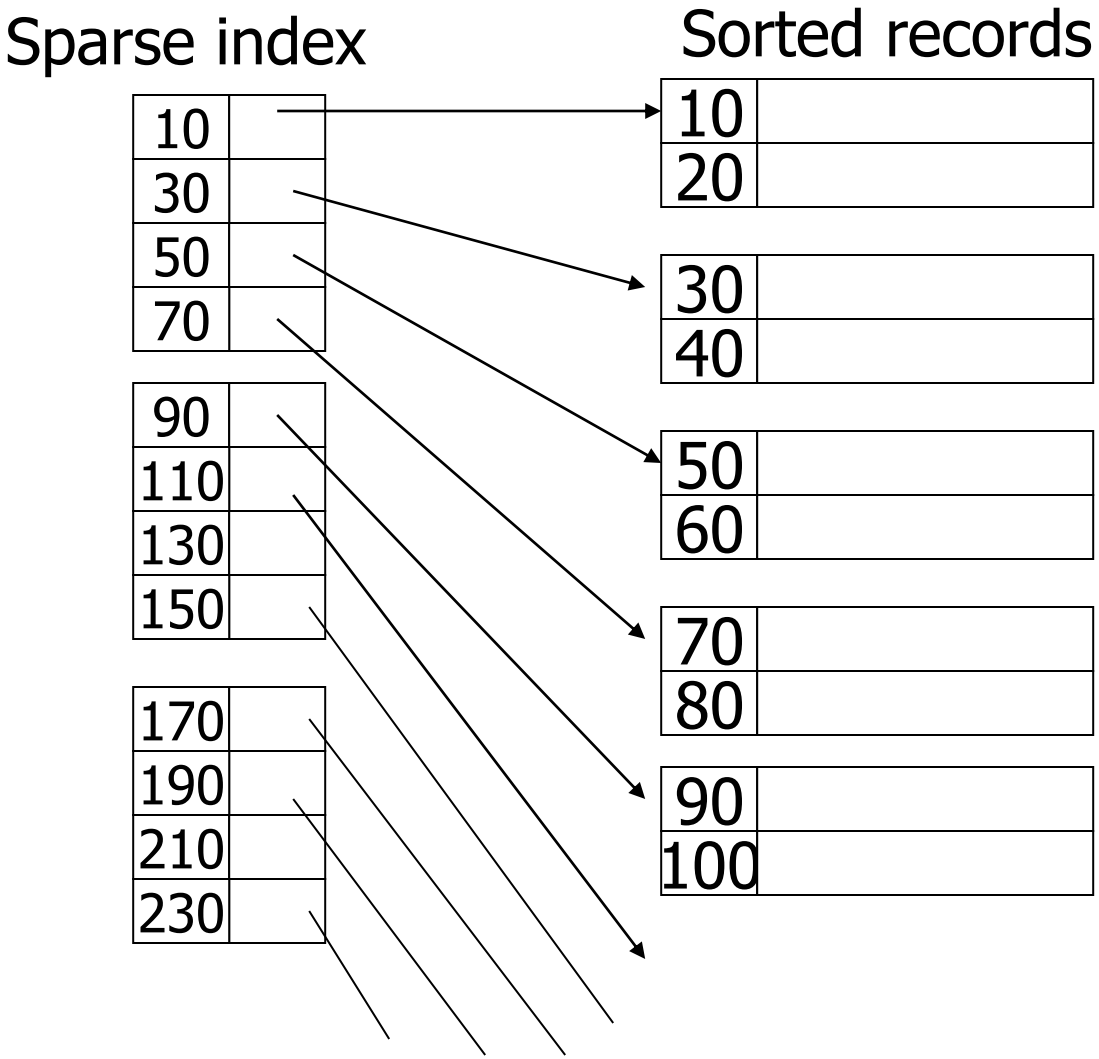
- Key vs. Search key
  - Key: the indexed attribute
  - Search key: the attribute queried on

# Simple Index File (Clustered, Dense)

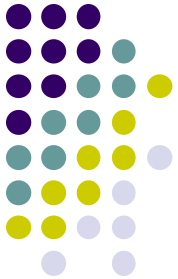


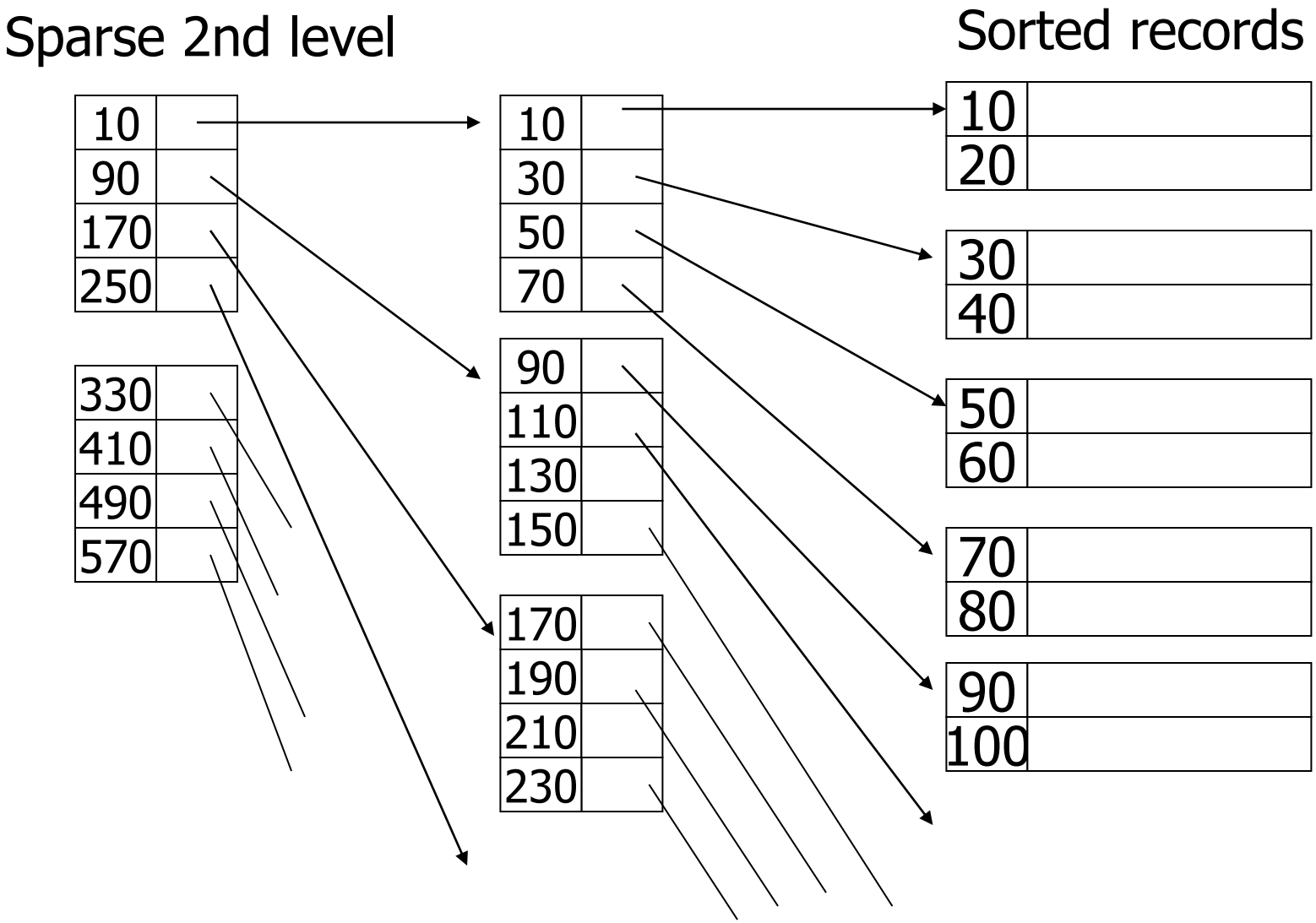


# Simple Index File (Clustered, Sparse)

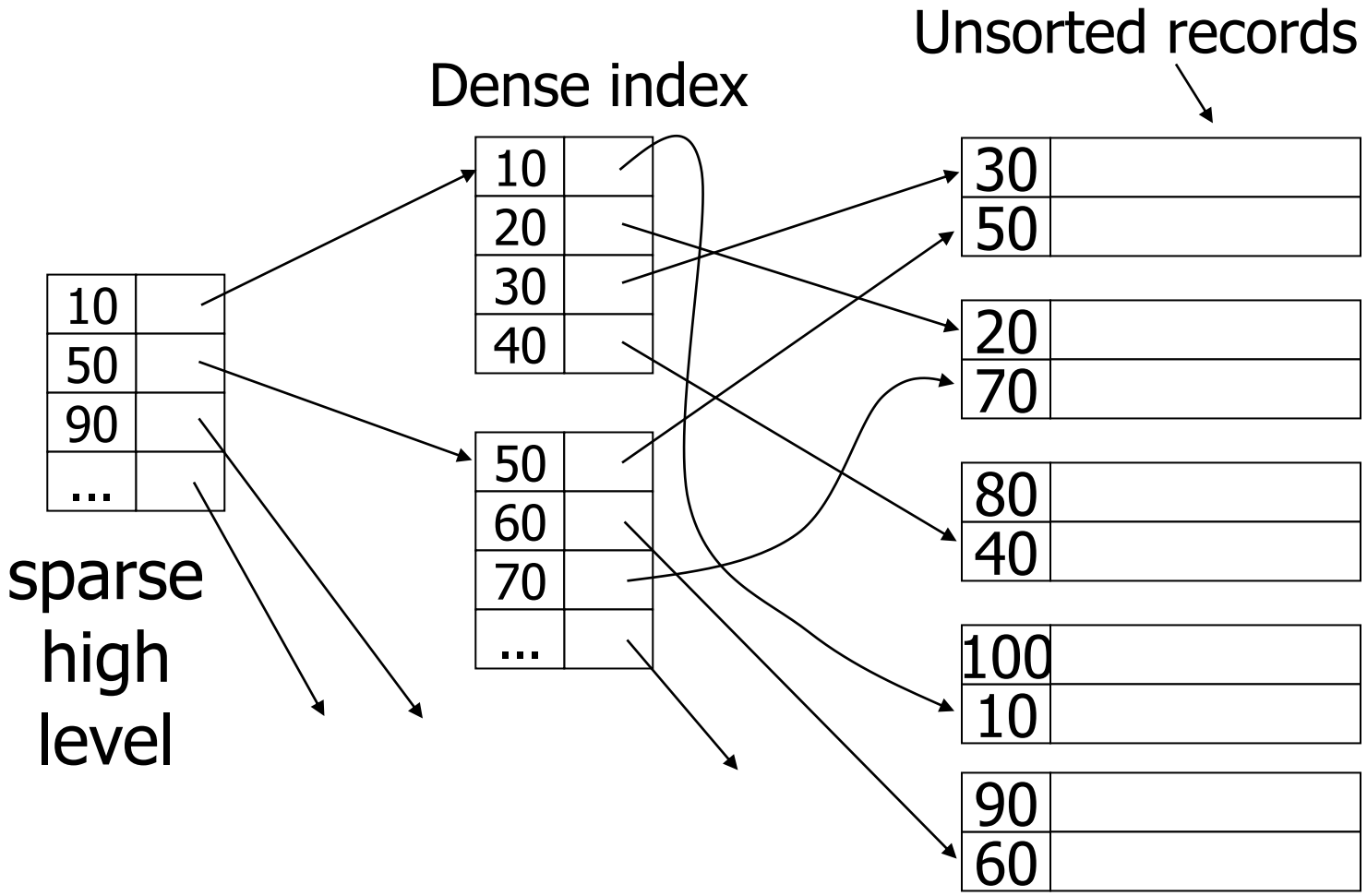


# Simple Index File (Clustered, Multi-level)

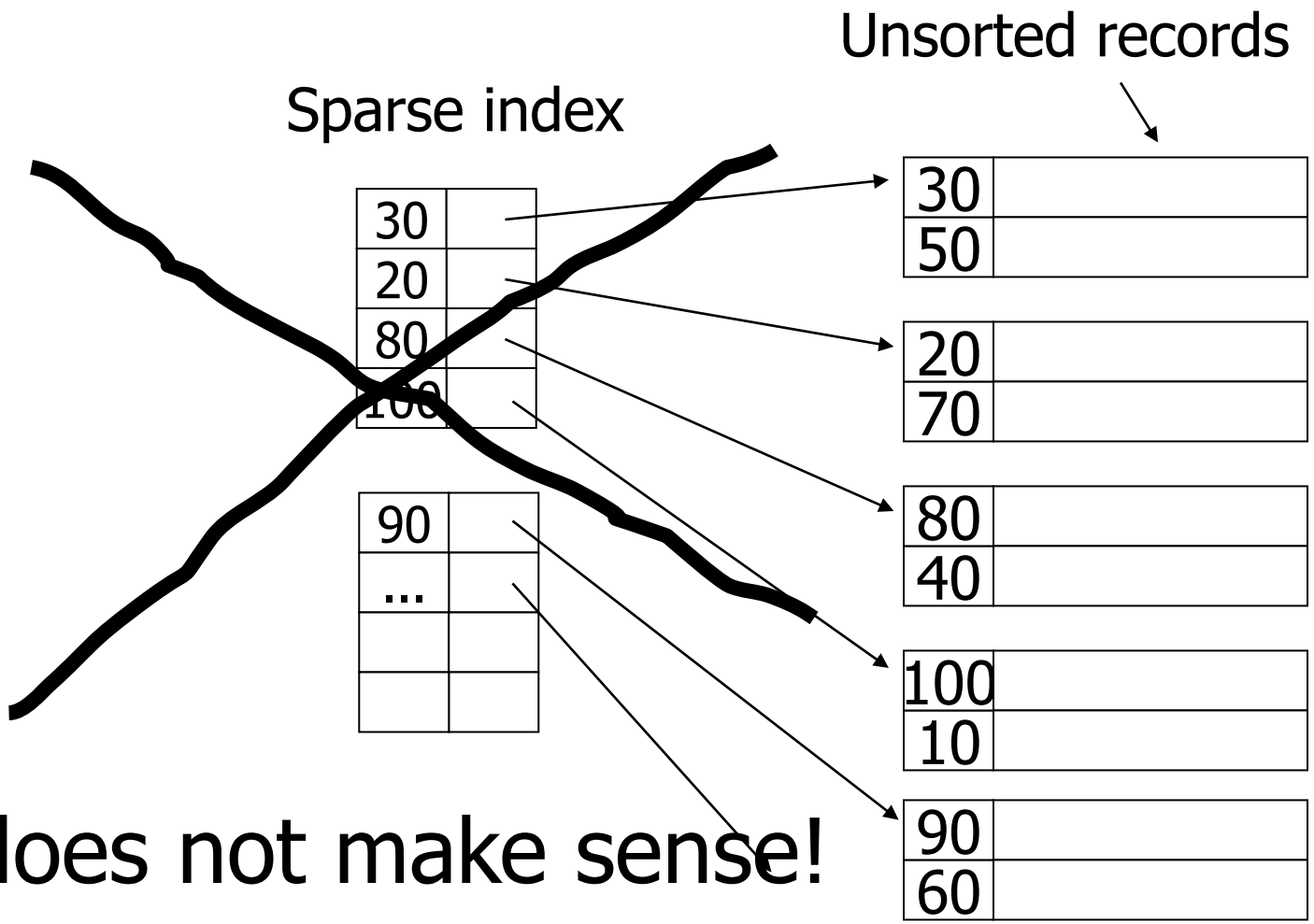




# Simple Index File (Unclustered, Dense)

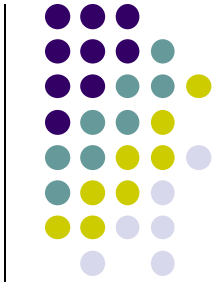


# Simple Index File (Unclustered, Sparse ?)




does not make sense!

# Properties of Indexes

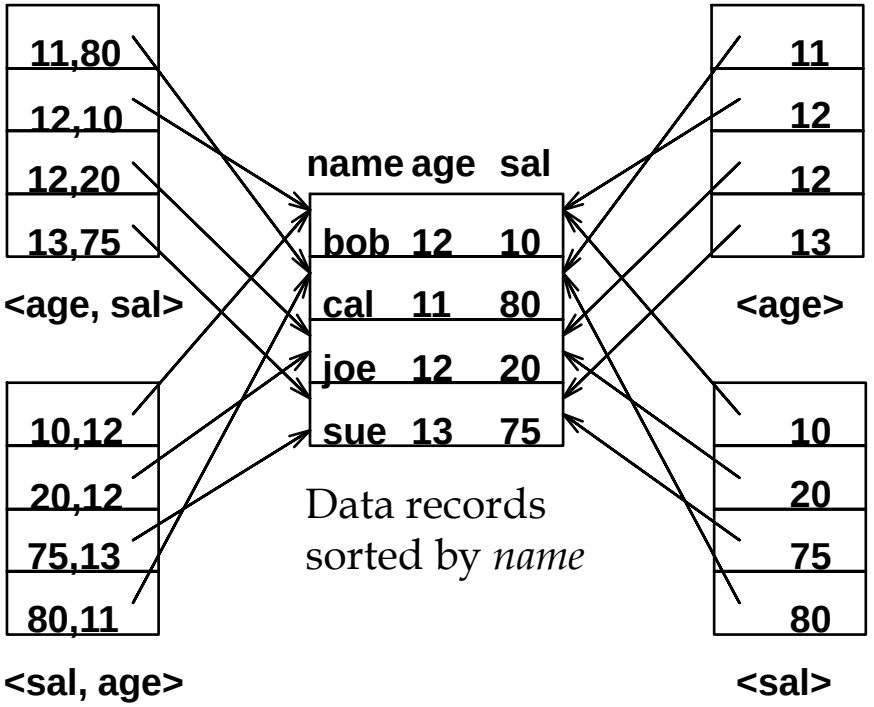


- Different types of entries in an index
  - An actual data record
  - A pair (key, ptr)
  - A pair (key, ptr-list)
- Clustered vs. Unclustered indexes
- Dense vs. Sparse indexes
  - Dense index on clustered or unclustered attributes
  - Sparse index only on clustered attribute
- Primary vs. Secondary indexes
  - Primary index on clustered attribute
  - Secondary index on unclustered attribute

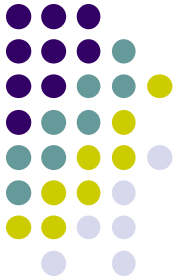
# Indexes on Composite Keys

- Q3: age=20 & sal=10
- Index on two or more attributes: entries are sorted first on the first attribute, then on the second attribute, the third …
- Q4: age=20 & sal>10
- Q5: sal=10 & age>20
- Note
  - Different indexes are useful for different queries

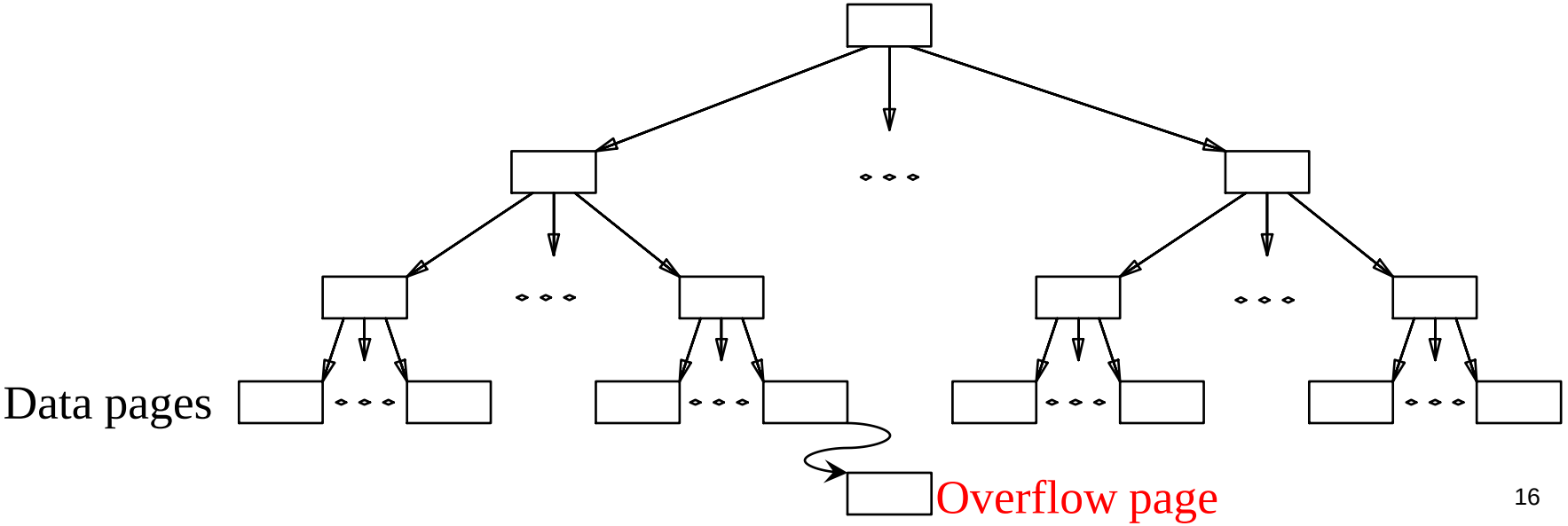
Examples of composite key indexes using lexicographic order.



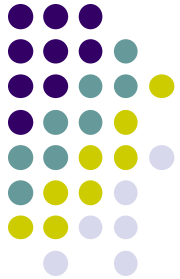
# Indexed sequential access method (ISAM)



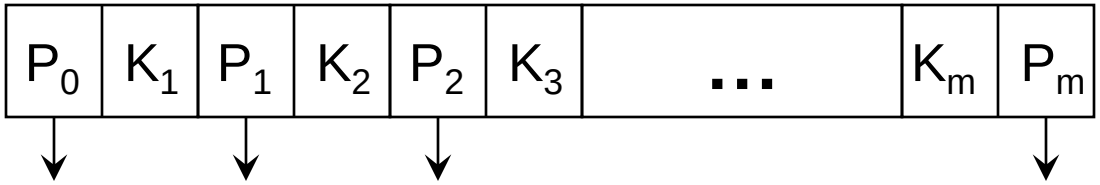
- Tree structured index
- Support queries
  - Point queries
  - Range queries
- Problems
  - Static: inefficient for insertions and deletions



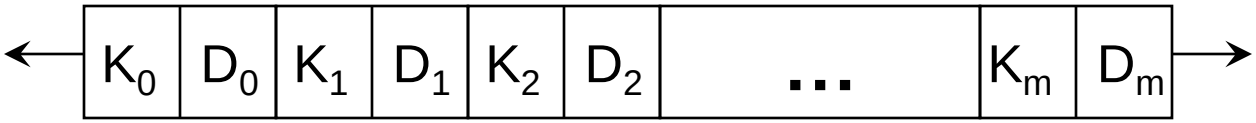
# The B$^+$-Tree: A Dynamic Index Structure



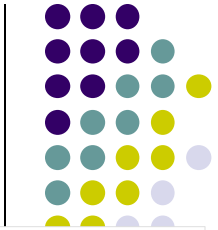
- Grows and shrinks dynamically
- Minimum 50% occupancy (except for root).
  - Each node contains *d* <= *m* <= *2d* entries. The parameter *d* is called the ***order of the tree***.
- Height-balanced
  - Insert/delete at $log_f N$ cost (*f* = fanout, *N* = No. leaf pages)
- Pointers to sibling pages
  - Non-leaf pages (internal pages)




  - Leaf pages
    - If directory page, same as non-leaf pages; pointers point to data page addresses
    - If data page

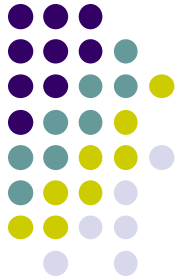


# B and B+ trees

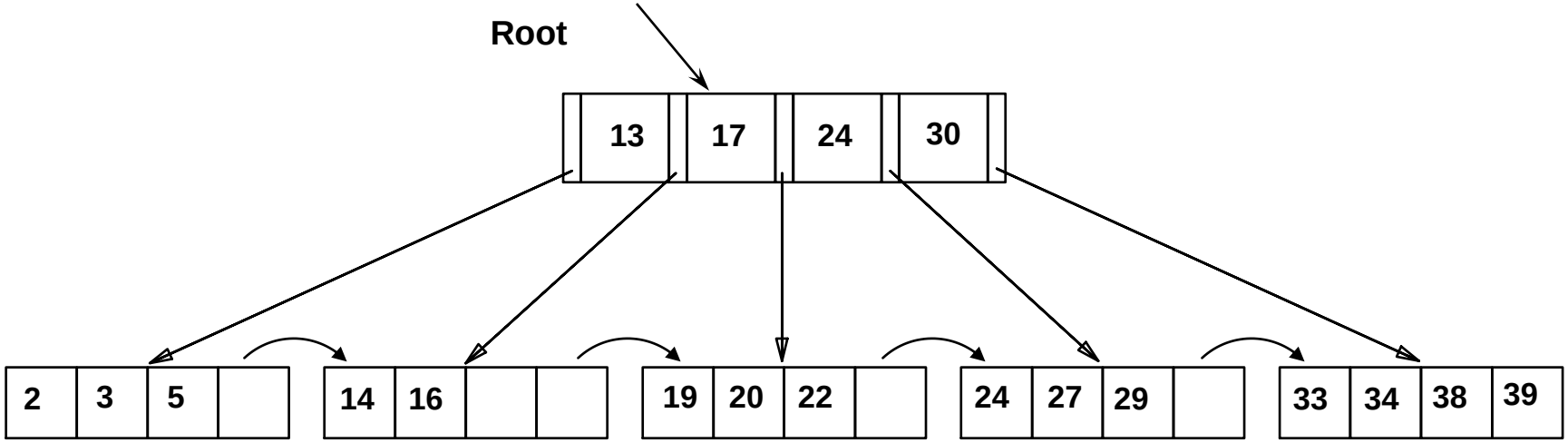


| | B tree | B+ tree |
|---|---|---|
| **Pointers** | All internal and leaf nodes have data pointers | Only leaf nodes have data pointers |
| **Search** | Since all keys are not available at leaf, search often takes more time. | All keys are at leaf nodes, hence search is faster and more accurate. |
| **Redundant Keys** | No duplicate of keys is maintained in the tree. | Duplicate of keys are maintained and all nodes are present at the leaf. |
| **Insertion** | Insertion takes more time and it is not predictable sometimes. | Insertion is easier and the results are always the same. |
| **Deletion** | Deletion of the internal node is very complex and the tree has to undergo a lot of transformations. | Deletion of any node is easy because all node are found at leaf. |
| **Leaf Nodes** | Leaf nodes are not stored as structural linked list. | Leaf nodes are stored as structural linked list. |
| **Access** | Sequential access to nodes is not possible | Sequential access is possible just like linked list |

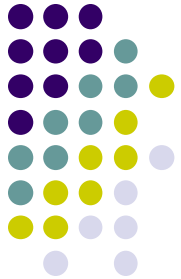
# Searching in a B$^+$-Tree



- Search begins at root, and key comparisons direct it to a leaf
- Search for 5, 15, all data entries >= 24 ...
- What about all entries <= 24 ?

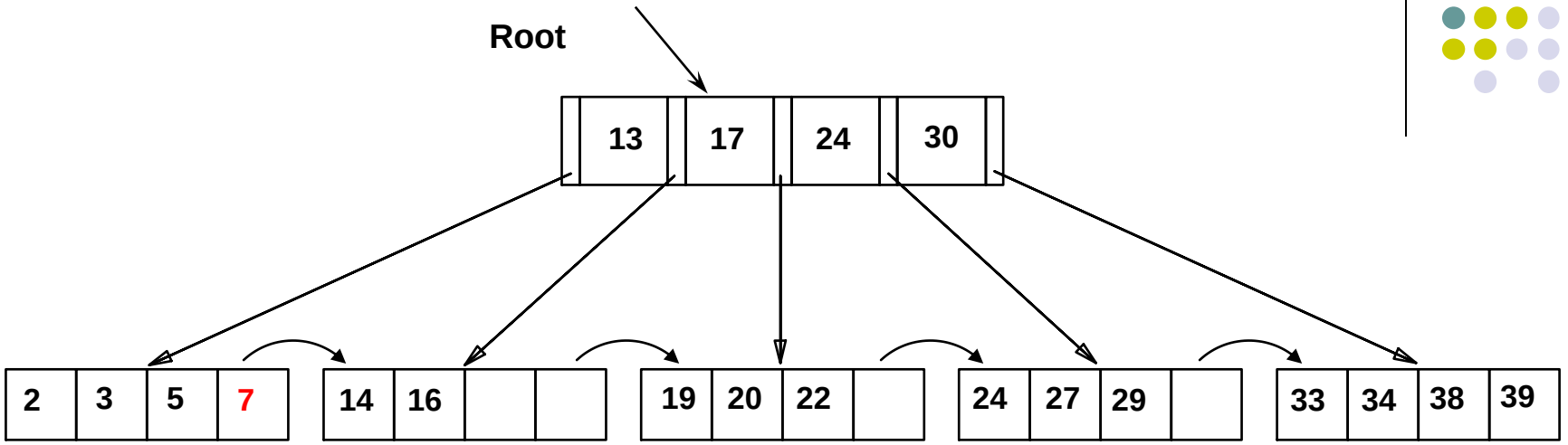


# Insertion in a B+-Tree



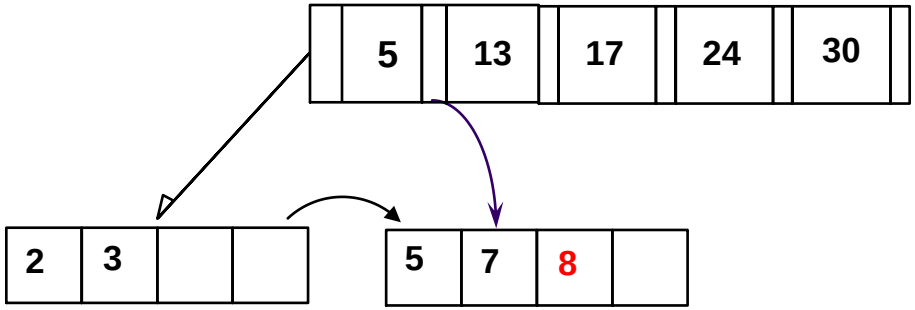
- Find correct leaf *L*
- Put data entry onto *L*
  - If *L* has enough space, *done*!
  - Else, must *split L (into L and a new node L2)*
    - Redistribute entries evenly, ***copy up*** middle key.
    - Insert index entry pointing to *L2* into parent of *L*.
- This can happen recursively
  - *To split index node*, redistribute entries evenly, but ***push up*** middle key. (Contrast with leaf splits.)
- Splits "grow" tree; root split increases height.
  - Tree growth: gets *wider* or *one level taller at top.*

# Inserting 7 & 8 into the B+-Tree




- Observe how minimum occupancy is guaranteed in both leaf and index pg splits

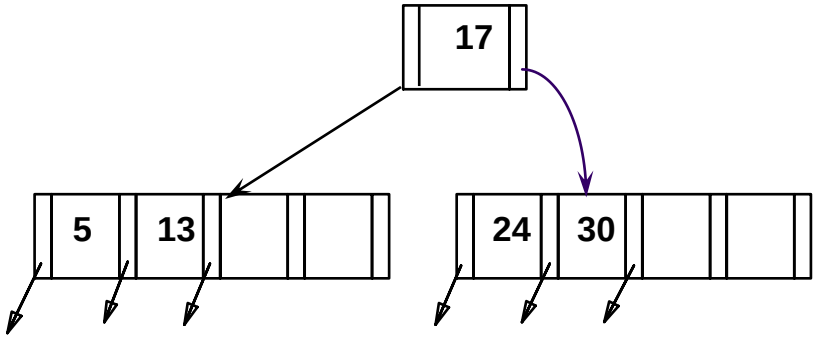
**(Note that 5 is copied up and continues to appear in the leaf.)**

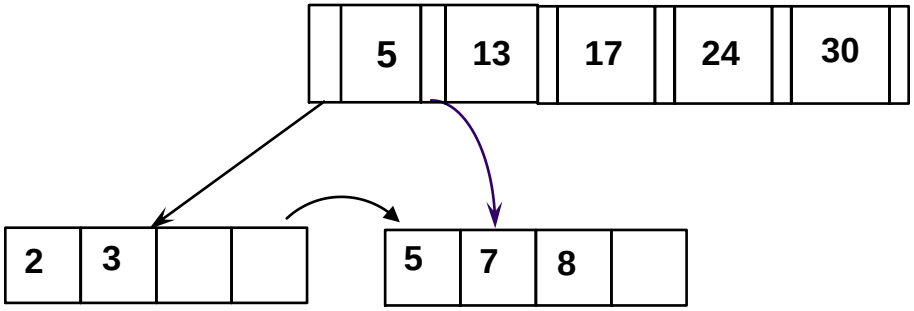


# Inserting 8 into the B+-Tree (continued)

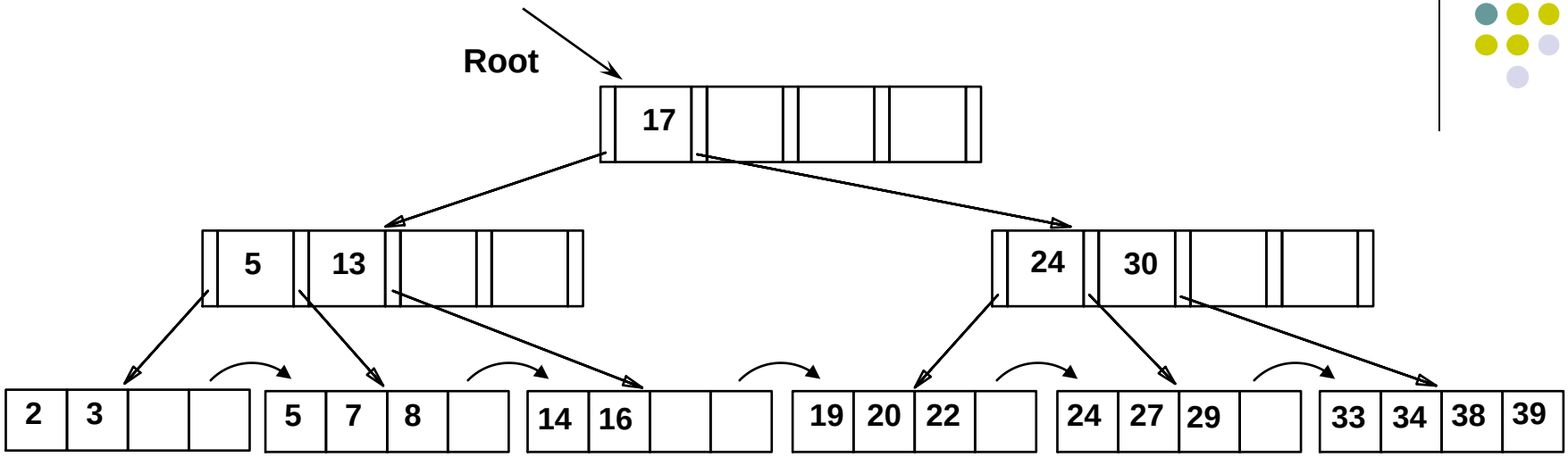
- Note the difference between ***copy up*** and ***push up***

**(17 is pushed up and only appears once in the index. Contrast this with a leaf split.)**



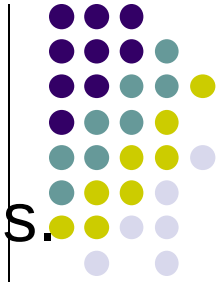



# The B+-Tree After Inserting 8



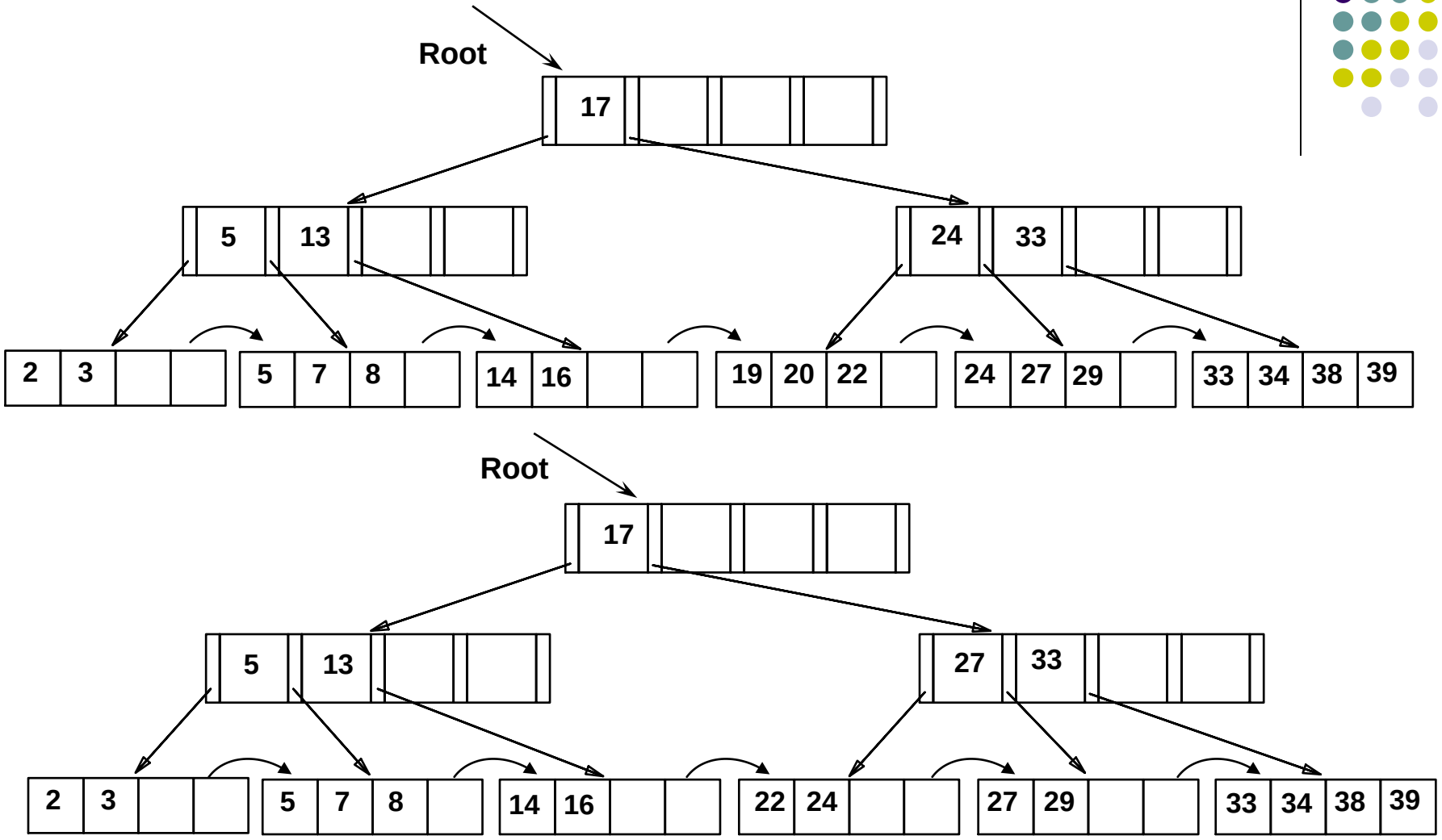

- Note that root was split, leading to increase in height
- We can avoid splitting by re-distributing entries. However, this is usually not done in practice. Why?

# Deletion in a B$^+$-Tree



- Start at root, find leaf *L* where the entry belongs.
- Remove the entry.
  - If L is at least half-full, *done!*
  - If L has only **d-1** entries,
    - Try to *redistribute*, borrowing from *sibling (adjacent node with same parent as L)*.
    - If redistribution fails, *merge L* and sibling.
- If merge occurred, must delete entry (pointing to *L* or sibling) from parent of *L*.
- Merge could propagate to root, decreasing height.

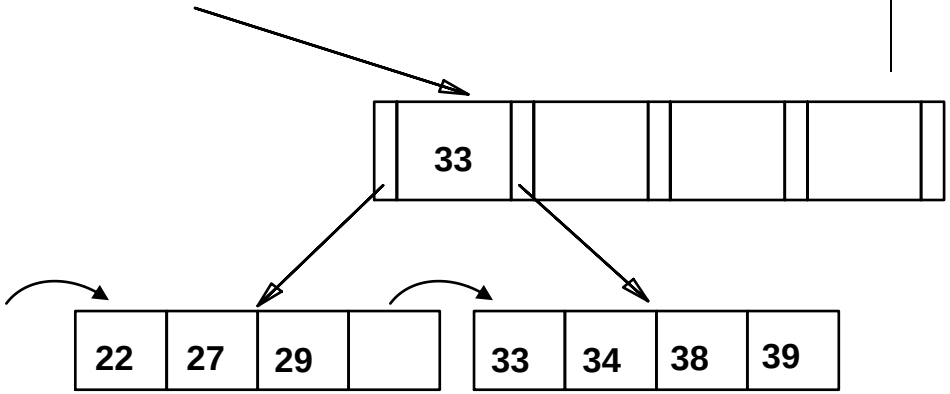
# Deleting 19 & 20 from the B+-Tree

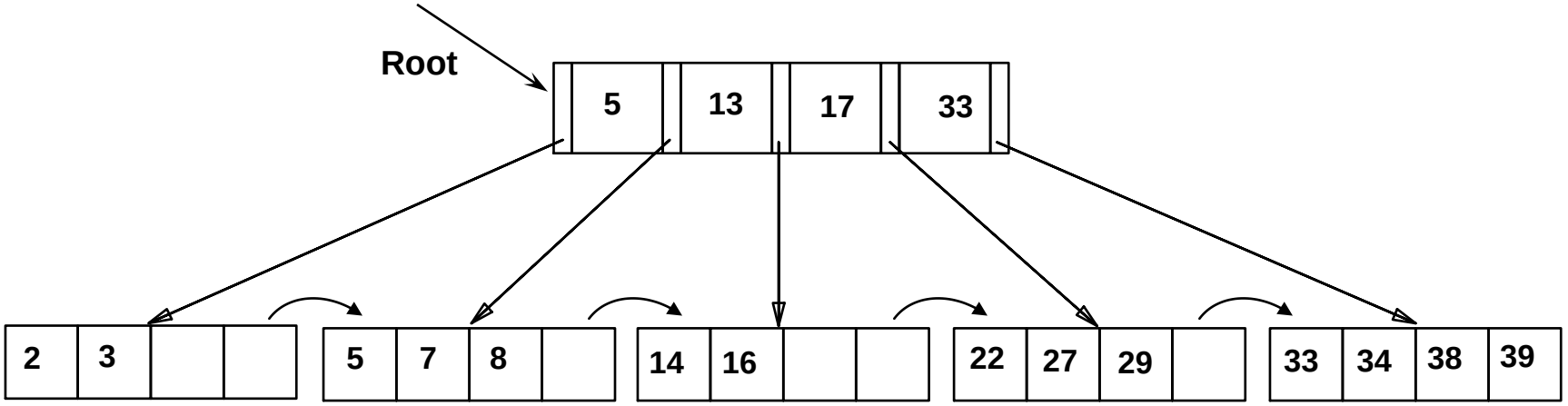



- Deleting 20 is done with re-distribution. Note how the middle key is ***copied up***

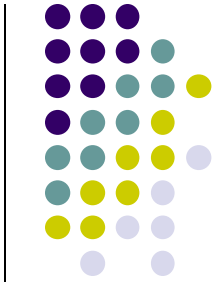
# Deleting 24 from the B+-Tree

- Merge happens
- Observe '***toss***' of index entry (on right), and '***pull down***' of index entry (below).
- Note the decrease in height



# $B^+$-Tree For All and Forever?



- Can the B+-tree being a single-dimensional index be used for emerging applications such as:
  - Spatial databases
  - High-dimensional databases
  - Temporal databases
  - Main memory databases
  - String databases
  - Genomic/sequence databases
  - ...????