ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ
ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ

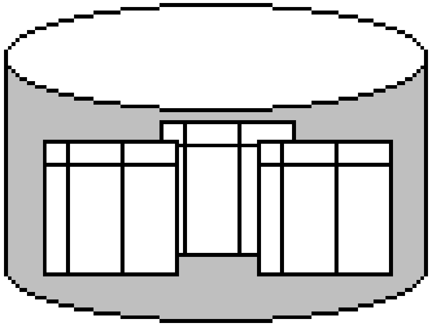
Συστήματα Διαχείρισης Βάσεων Δεδομένων

Φροντιστήριο 3: Tutorial on Indexing part 1 – Properties and Tree-Structured Indexing

Δημήτρης Πλεξουσάκης
Τμήμα Επιστήμης Υπολογιστών

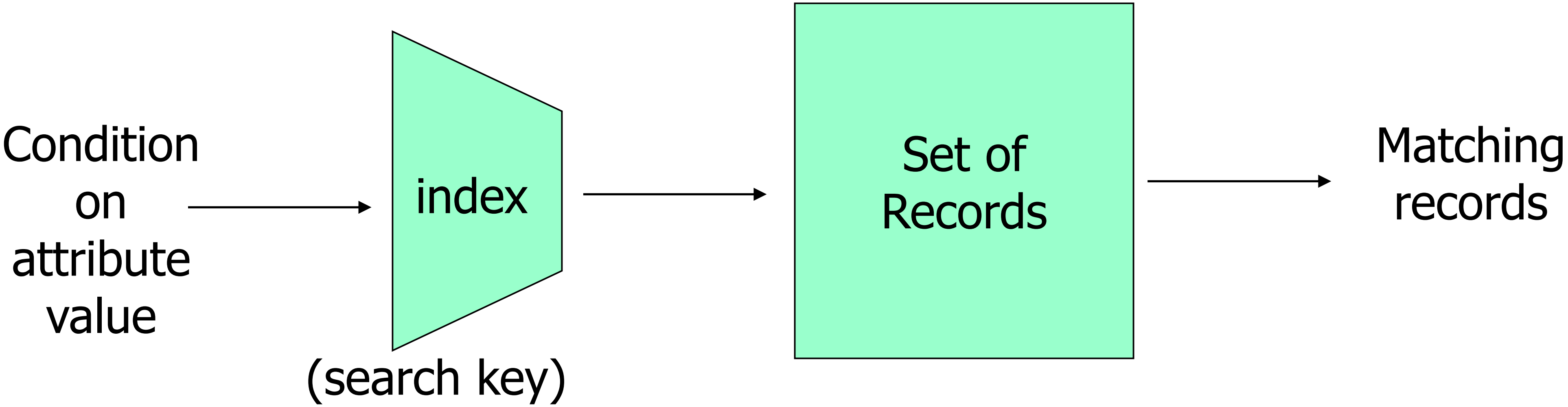
TUTORIAL ON INDEXING PART 1:

PROPERTIES & TREE-STRUCTURED INDEXING



Index

- An index is a data structure that supports efficient access to data (i.e., row(s)) without having to scan entire table
- Based on a search key: rows having a particular value for the search key attributes can be quickly located



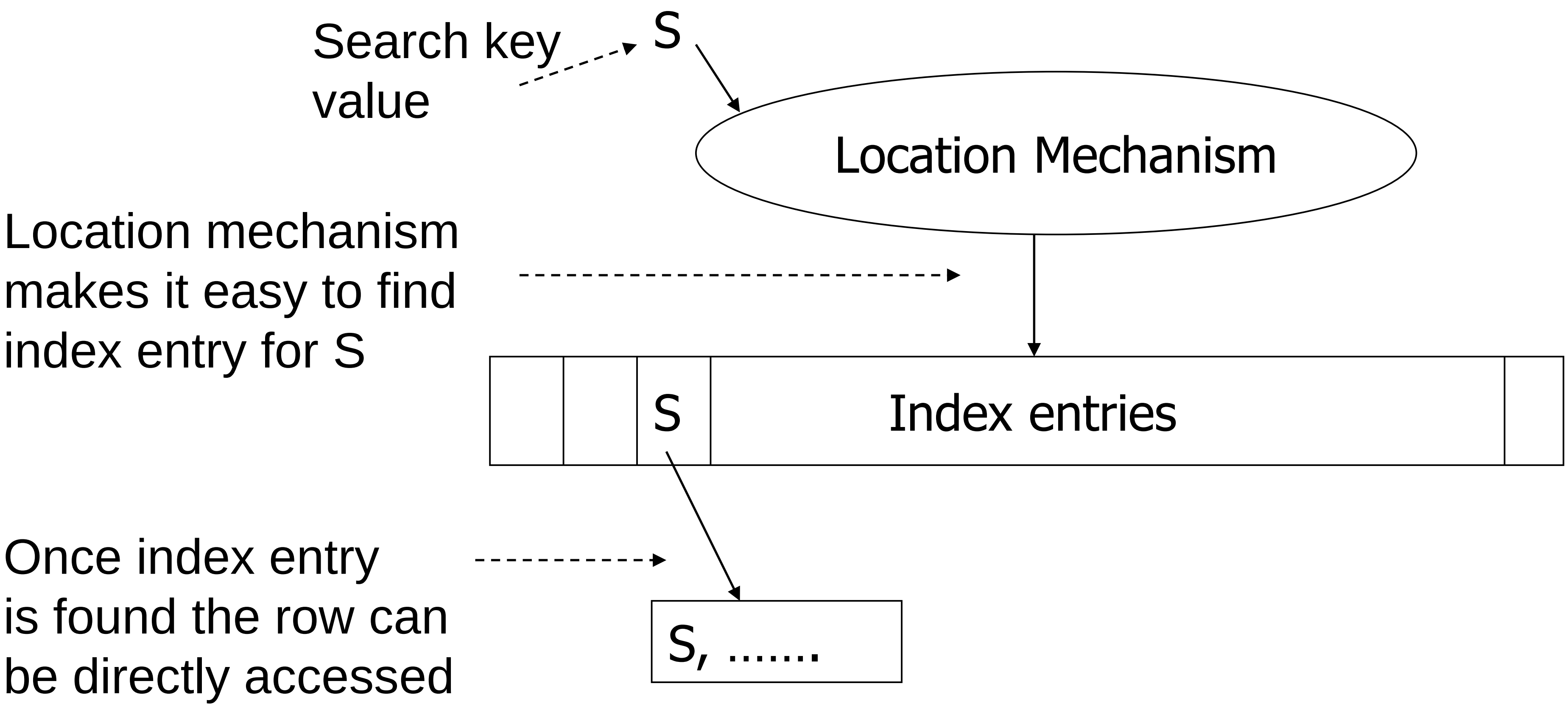
Index Search Keys

- Candidate key vs. search key:
  - Candidate key: set of attributes; guarantees uniqueness
  - Search key: sequence of attributes; does not guarantee uniqueness
- Types of (search) keys
  - Sequential: the value of the key is monotonic with the insertion order (e.g., counter or timestamp)
  - Non sequential: the value of the key is unrelated to the insertion order (e.g., social security number)

Index Structure

- Index entries
  - The row itself (index and table are integrated in this case), or
  - Search key value and a pointer to a row having that value (table is stored separately in this case)
- Location mechanism
  - Algorithm + data structure for locating an index entry with a given search key value
  - Various data structures can serve as indexes, e.g.,
    - simple indexes on sorted (sequential) files
    - secondary indexes on unsorted (heap) files
    - hierarchical index structures (B-trees)
    - hash tables
- Index entries are stored in accordance with the search key value
  - Entries with the same search key value are stored together (hash, B+ tree)
  - Entries may be sorted on search key value (B+ tree)

Index Structure



Search key value

Location mechanism makes it easy to find index entry for S

Once index entry is found the row can be directly accessed

Storage Structure

- Structure of file containing a table
  - Heap file (no index, not integrated)
  - Sorted file (no index, not integrated)
  - Integrated file containing index and rows (index entries contain rows in this case) e.g.,
    - Indexed Sequential Access Method (ISAM)
    - B+ tree
    - Hash
- The simplest of cases: given a sorted file (data file), create another file (index file) consisting of key-pointer pairs:
  - a search key K is associated with a pointer pointing to a record of the data file that has the search key K

Indices: The Down Side

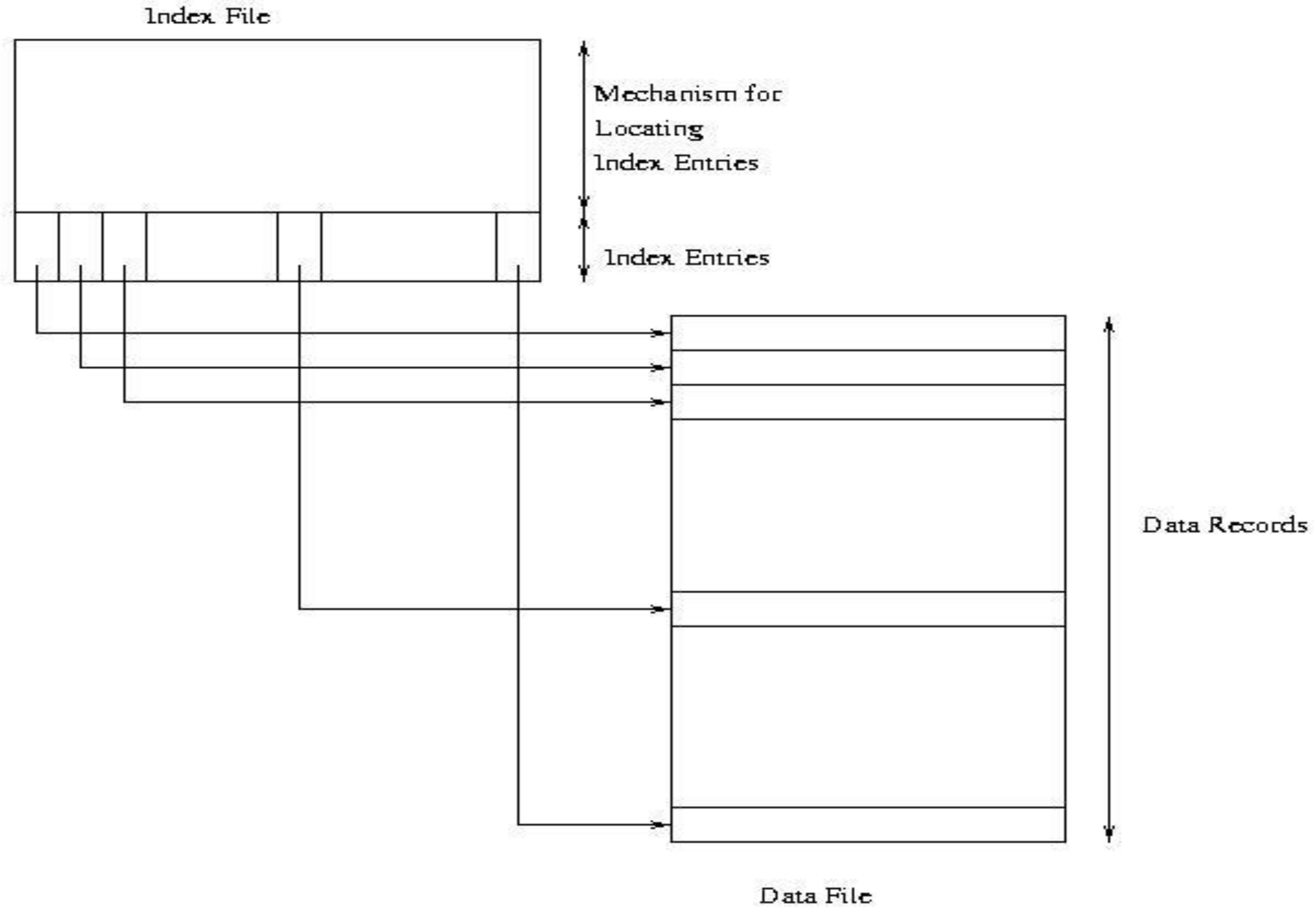
- An index is itself an ordered file
  - The index file is physically ordered on disk by the key field
  - The index file has records of fixed length, containing: key field, pointer to data `<ki, pi>`
- Additional I/O to access index pages (except if index is small enough to fit in main memory)
- Index must be updated when table is modified
- SQL-92 does not provide for creation or deletion of indices
  - Index on primary key generally created automatically
  - Vendor specific statements:
    - `CREATE INDEX ind ON Student (Code)`
    - `DROP INDEX ind`

Clustered Index

- Clustered index: index entries and rows are ordered in the same way
  - ◆An integrated storage structure is always clustered (since rows and index entries are the same)
  - ◆The particular index structure (e.g., hash, tree) dictates how the rows are organized in the storage structure
    - There can be at most one clustered index on a table
  - ◆`CREATE TABLE` generally creates an integrated, clustered (main) index

- Good for range searches when a range of search key values is requested
  - ◆Use location mechanism to locate index entry at start of range
    - This locates first row
  - ◆Subsequent rows are stored in successive locations if index is clustered (not so if unclustered)
  - ◆Minimizes page transfers and maximizes likelihood of cache hits

Clustered Secondary Index

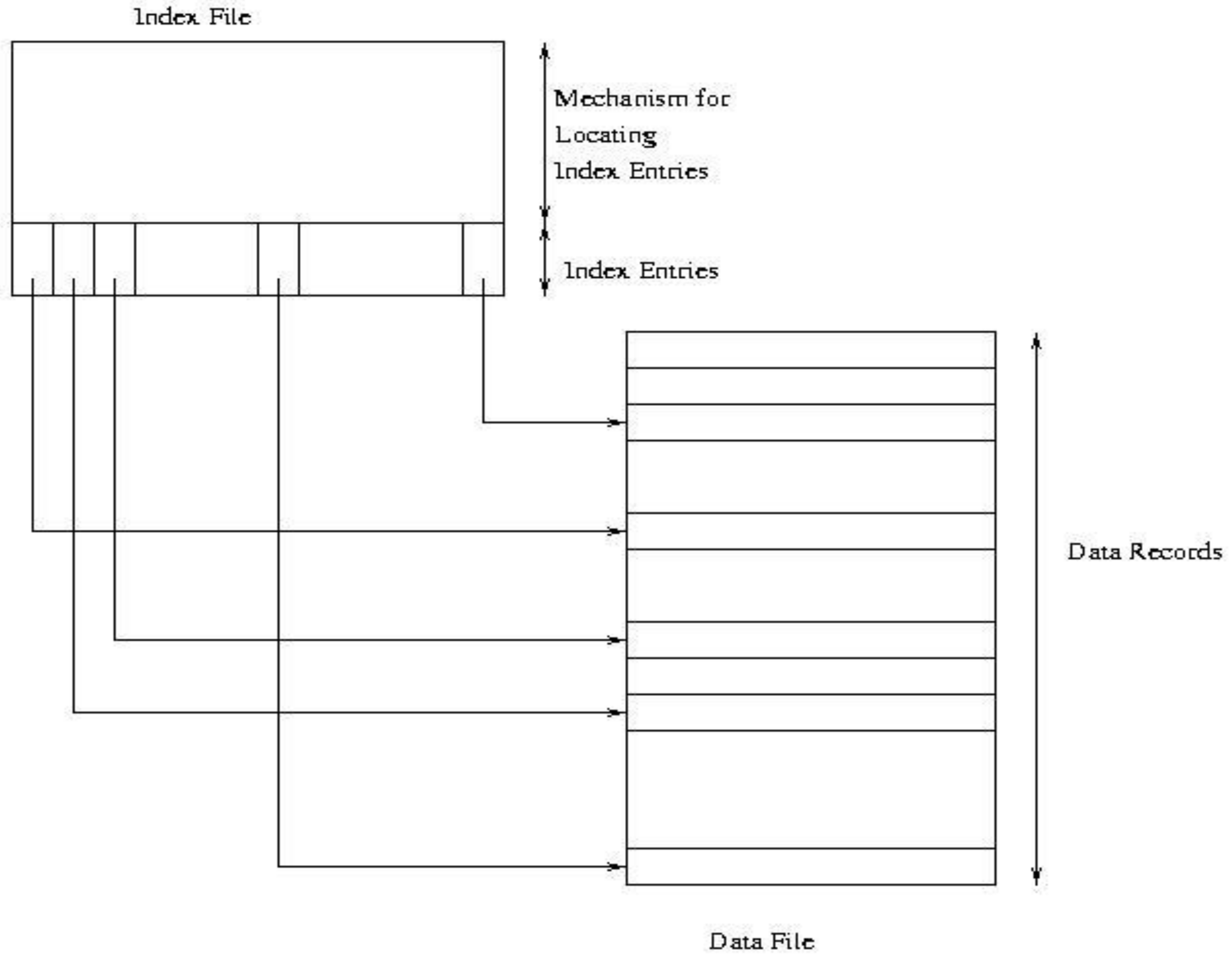


Unclustered Index

- Unclustered index: index entries and rows are not ordered in the same way

- A secondary index might be clustered or unclustered with respect to the storage structure it references
  - It is generally unclustered (since the organization of rows in the storage structure depends on main index)
  - There can be many secondary indices on a table
  - Index created by `CREATE INDEX` is generally an unclustered, secondary index

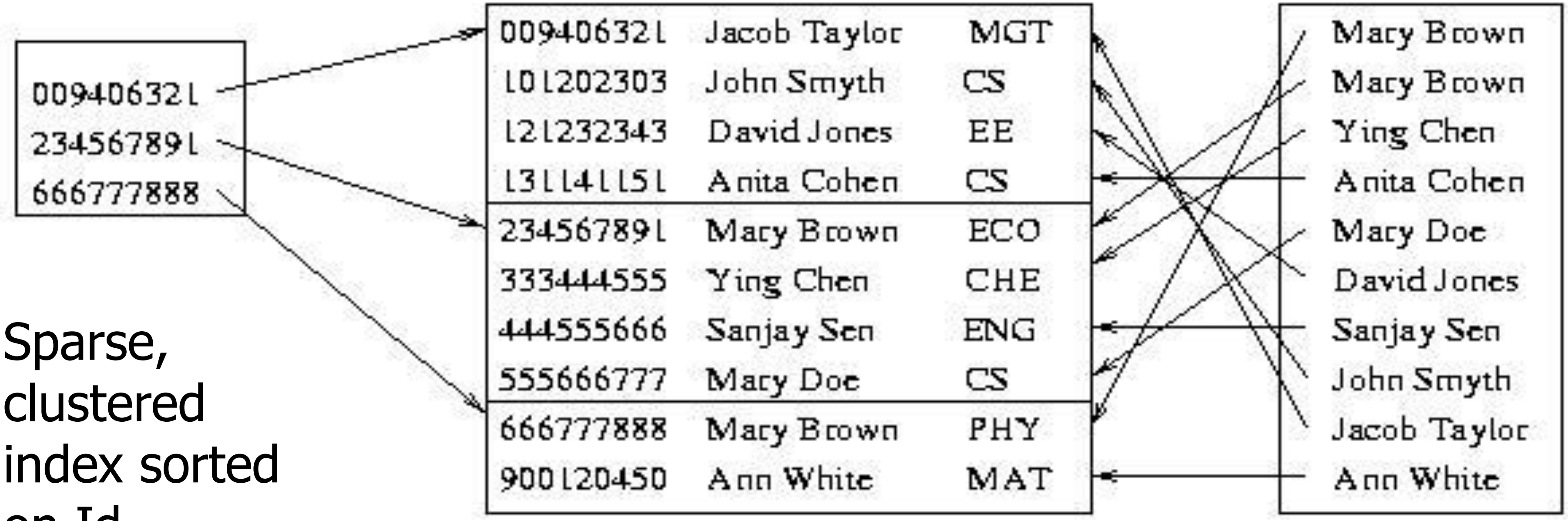
Unclustered Secondary Index



Sparse vs Dense Index

- Dense index: every search key value in the data file is in the index
  - index entry for each data record (faster to locate a record)
    - Unclustered index must be dense
    - Secondary clustered index need not be dense
- Sparse index: not every search key value in the data file is in the index
  - index entry for each page of data file
    - index indicates the block of records
- A good compromise
  - Sparse index with one index entry per block
  - Tradeoff between access time and space overhead

Sparse vs Dense Index



009406321
234567891
666777888

Id	Name	Dept
009406321	Jacob Taylor	MGT
101202303	John Smyth	CS
121232343	David Jones	EE
131141151	Anita Cohen	CS
234567891	Mary Brown	ECO
333444555	Ying Chen	CHE
444555666	Sanjay Sen	ENG
555666777	Mary Doe	CS
666777888	Mary Brown	PHY
900120450	Ann White	MAT

Mary Brown
Mary Brown
Ying Chen
Anita Cohen
Mary Doe
David Jones
Sanjay Sen
John Smyth
Jacob Taylor
Ann White

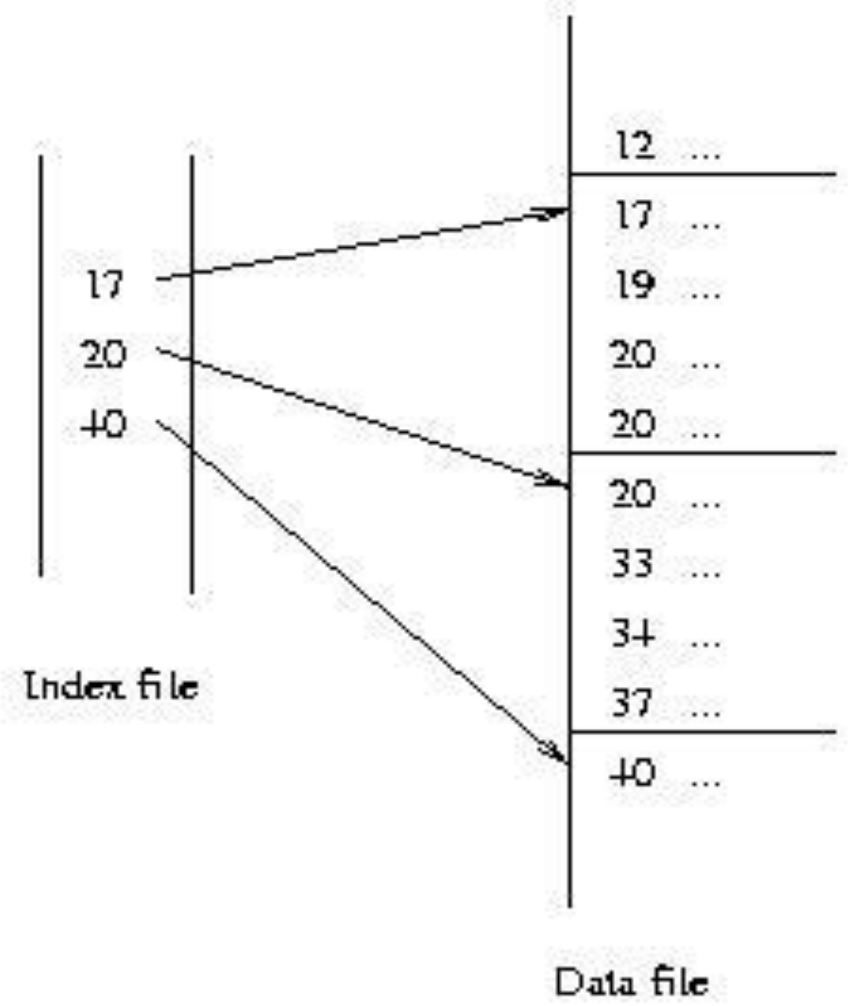
Sparse, clustered index sorted on Id

data file sorted on Id

Dense, unclustered index sorted on Name

Sparse Index

Search key should be candidate key of data file (else additional measures required, more on that later)



Multiple Attribute Search Key

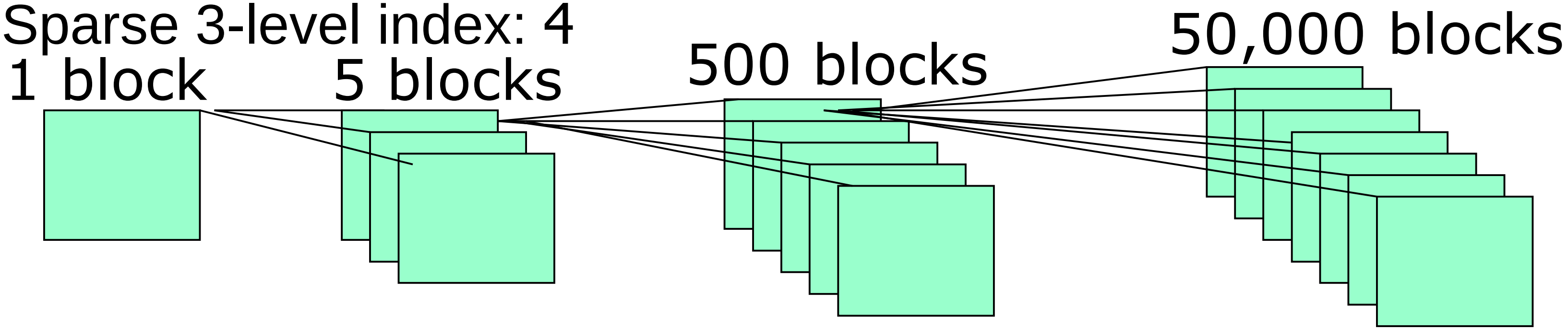
- `CREATE INDEX  Inx ON Tbl  (att1, att2)`
- Search key is a *sequence* of attributes; index entries are lexically ordered
- Supports finer granularity equality search:
  - *"Find row with value (A1, A2) "*
- Supports range search (tree index only):
  - *"Find rows with values between (A1, A2) and (A1′, A2′) "*
- Supports partial key searches (tree index only):
  - Find rows with values of att1 between A1 and A1′
  - But not "Find rows with values of att2 between A2 and A2′ "

Locating an Index Entry

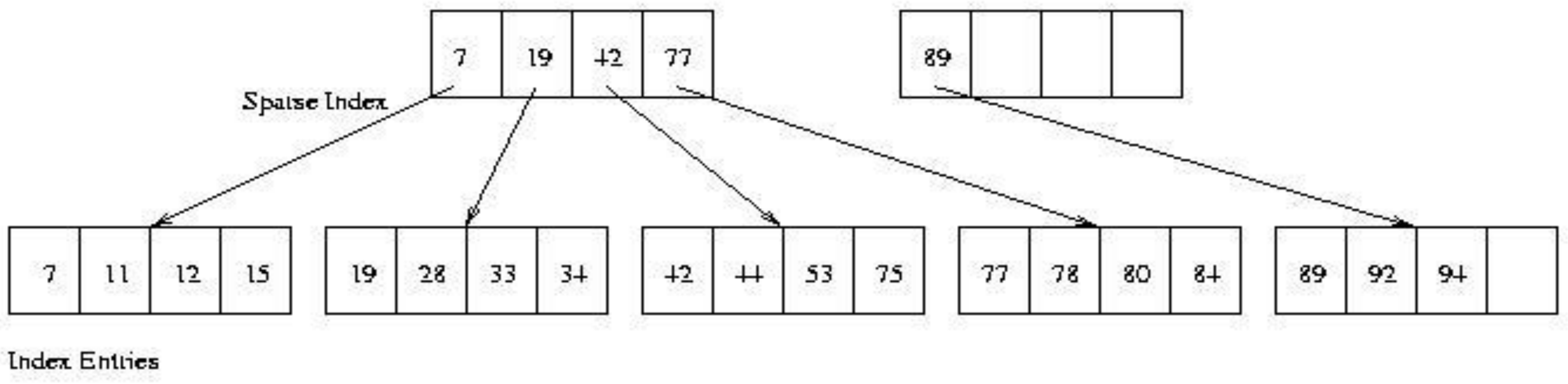
- Use scan (index entries not sorted)
- Use binary search (index entries sorted)
  - If `Q` pages of index entries, then $\log_2 Q$ page transfers (which is a big improvement over binary search of the data pages of a `F` page data file since `F` >>Q)
- Use multilevel index: Sparse index on sorted list of index entries

Multiple Index Levels

- Since index files can be large, an index on an index file may be built
  - Not all levels need to be equally dense or sparse
- A sparse index may be built on a dense 1st-level index (otherwise the 2nd-level index would have as many records as the 1st-level index)
- Example 1: Sparse index occupies 1.000 blocks; a 2nd-level sparse index would need only 10 blocks (for 100 records/block)
  - then to locate a record with a given key value, we need two I/Os
- Example 2: Consider 5.000.000 ordered records, 100 records/block, thus 50.000 blocks
  - Sequential search: average of 50.000 / 2 blocks read
  - Binary search: log $_2$ 50.000 = 16
  - Sparse 3-level index: 4

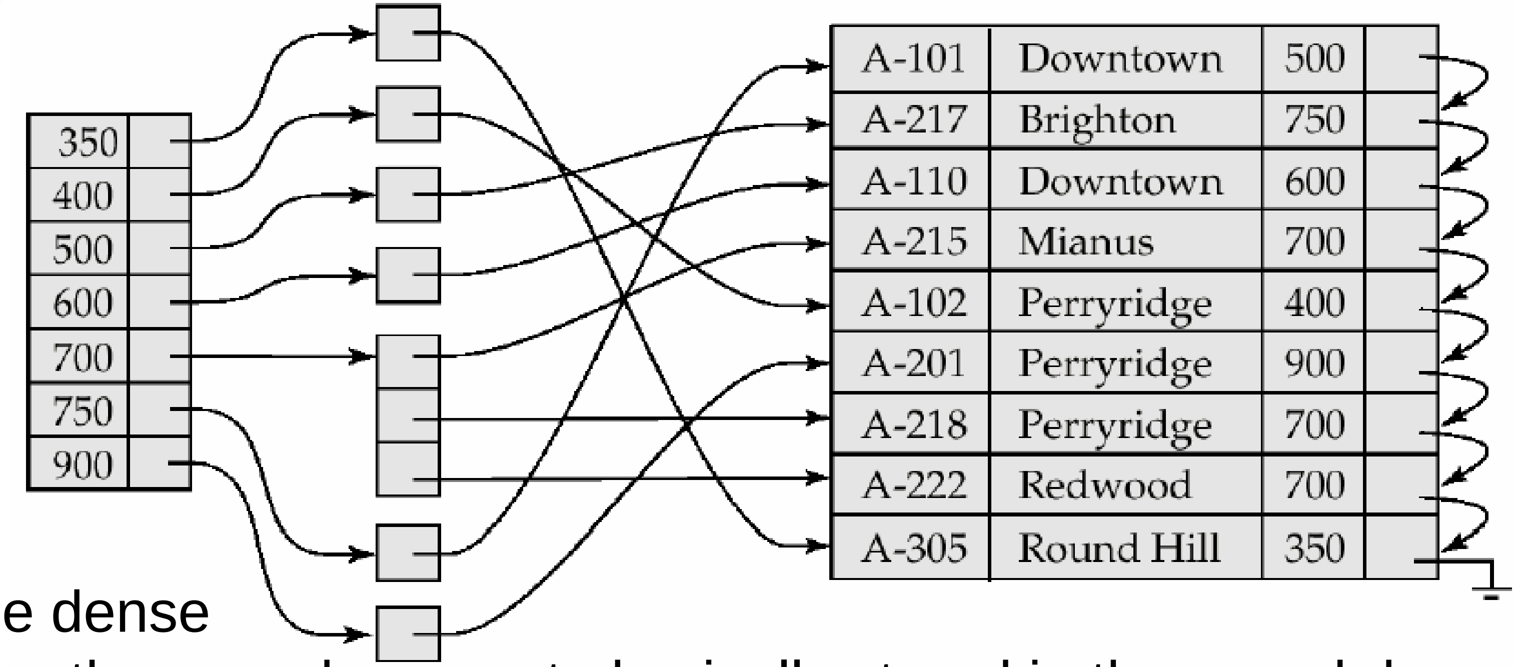


Two-Level Index



- Separator level is a sparse index over pages of index entries
- Leaf level contains index entries
- Cost of searching the separator level << cost of searching index level since separator level is sparse
- Cost or retrieving row once index entry is found is 0 (if integrated) or 1 (if not)

Secondary Indices

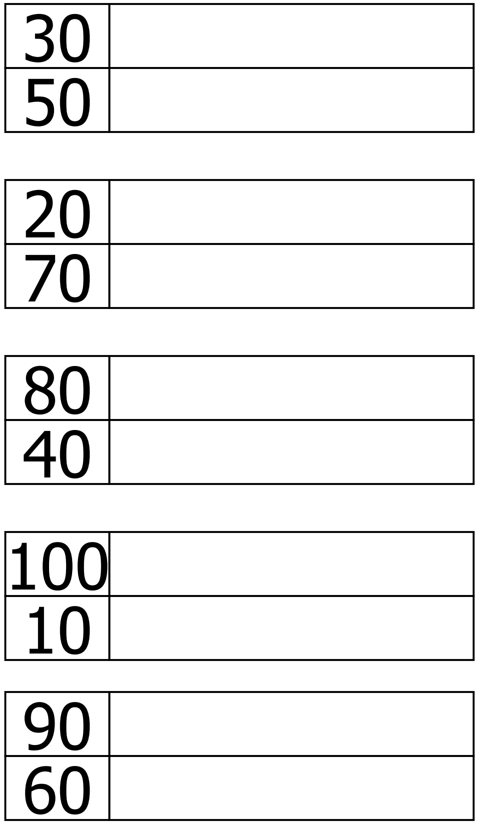


A-101	Downtown	500
A-217	Brighton	750
A-110	Downtown	600
A-215	Mianus	700
A-102	Perryridge	400
A-201	Perryridge	900
A-218	Perryridge	700
A-222	Redwood	700
A-305	Round Hill	350

- Must be dense
  - ◆Since the records are not physically stored in the search key order
  - ◆Can use multiple levels, with only the lowest being necessarily dense
- A sequential scan of the records in the search key of a secondary index is often very slow
  - ◆Requires many block accesses
- Data updates lead to updates of all index levels
  - ◆A likely significant overhead on modification of the database

Secondary Indices

File not sorted on secondary search key



30
50

20
70

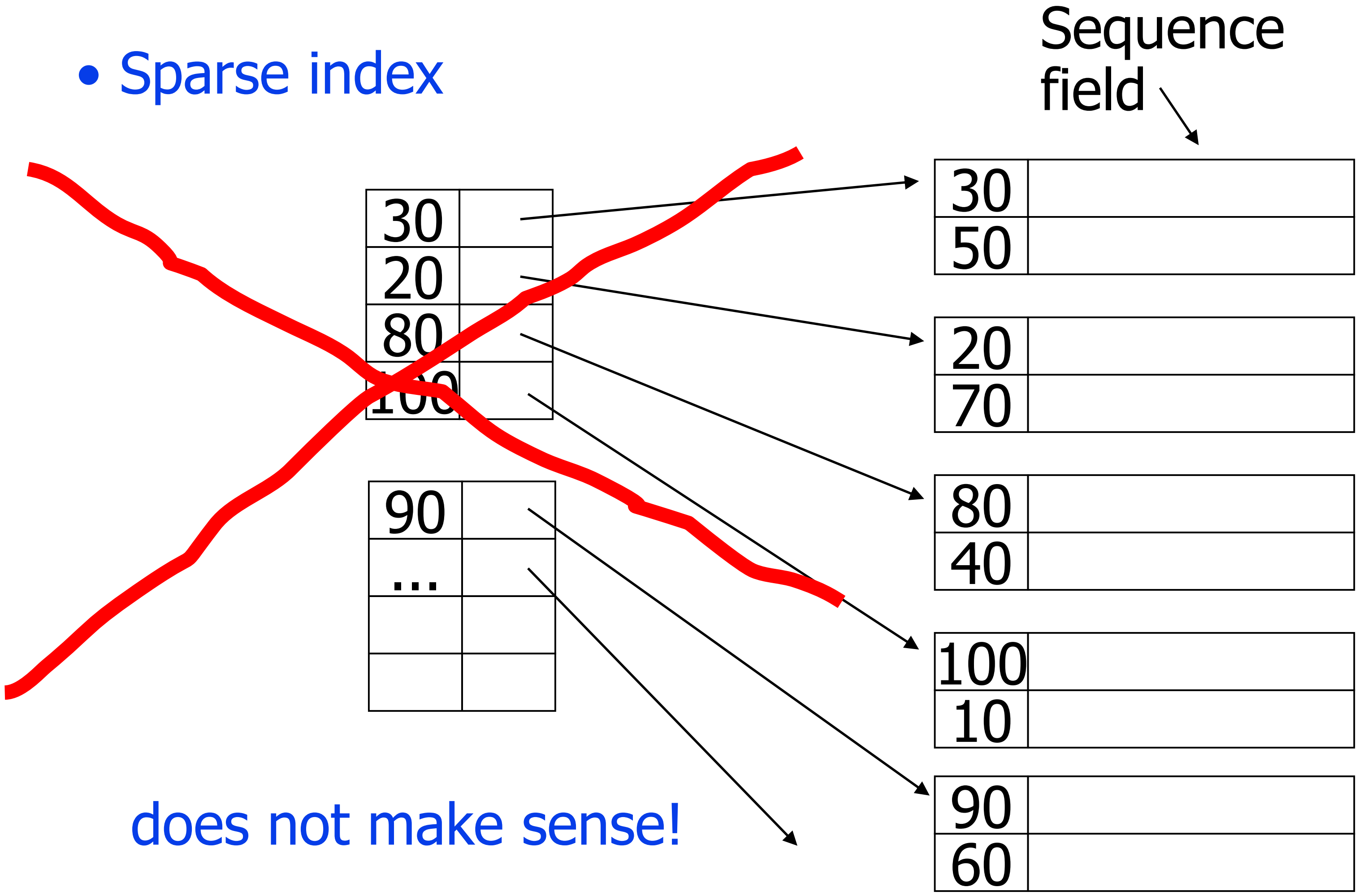
80
40

100
10

90
60

Secondary Indices

- Sparse index



Sequence field

30	
20	
80	
100	

90	
...	

30	
50	

20	
70	

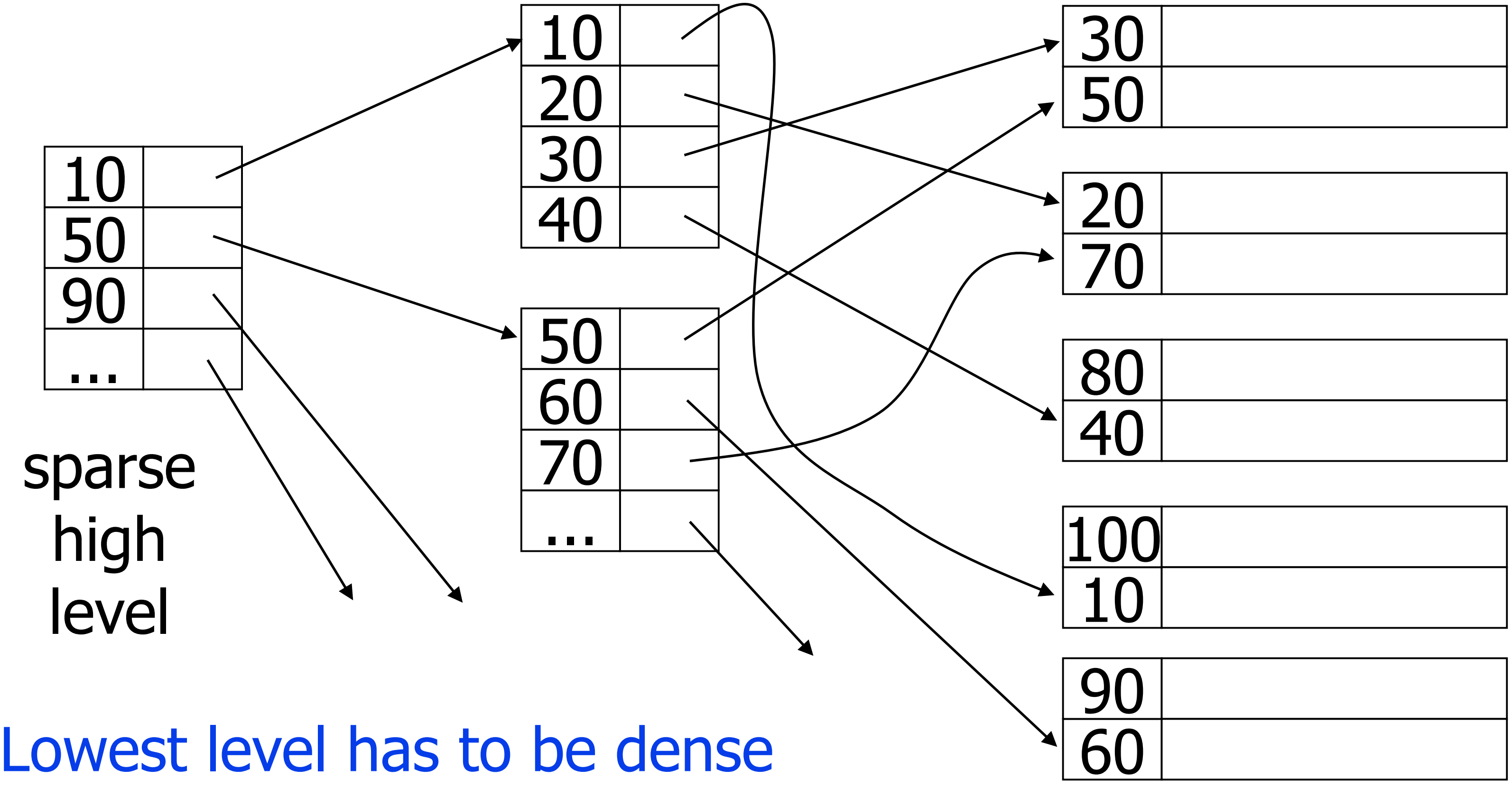
80	
40	

100	
10	

90	
60	

does not make sense!

Secondary Indices



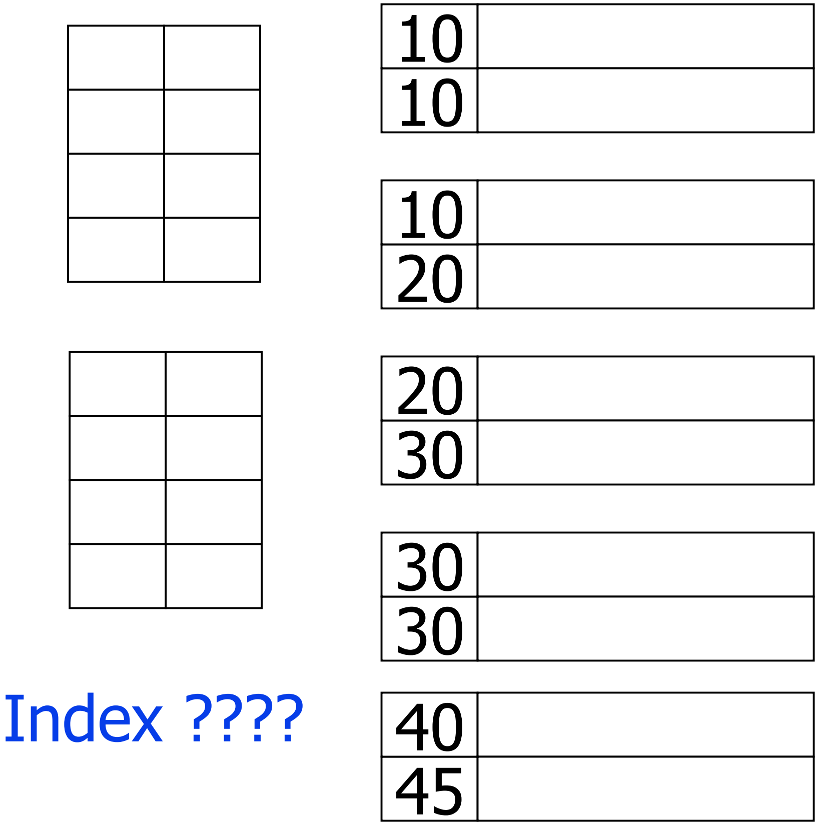
Lowest level has to be dense

Indexes with Duplicate Search Keys

- If indexes are built on non-key attributes, then more than one record may have a given key value
- Assume that the order of records with identical search key values does not matter
- A dense index can be built with one entry with key K for each record that has search key K
- Searching: look for the first record with key K in the index file and find all the other records with the same value (they must immediately follow)

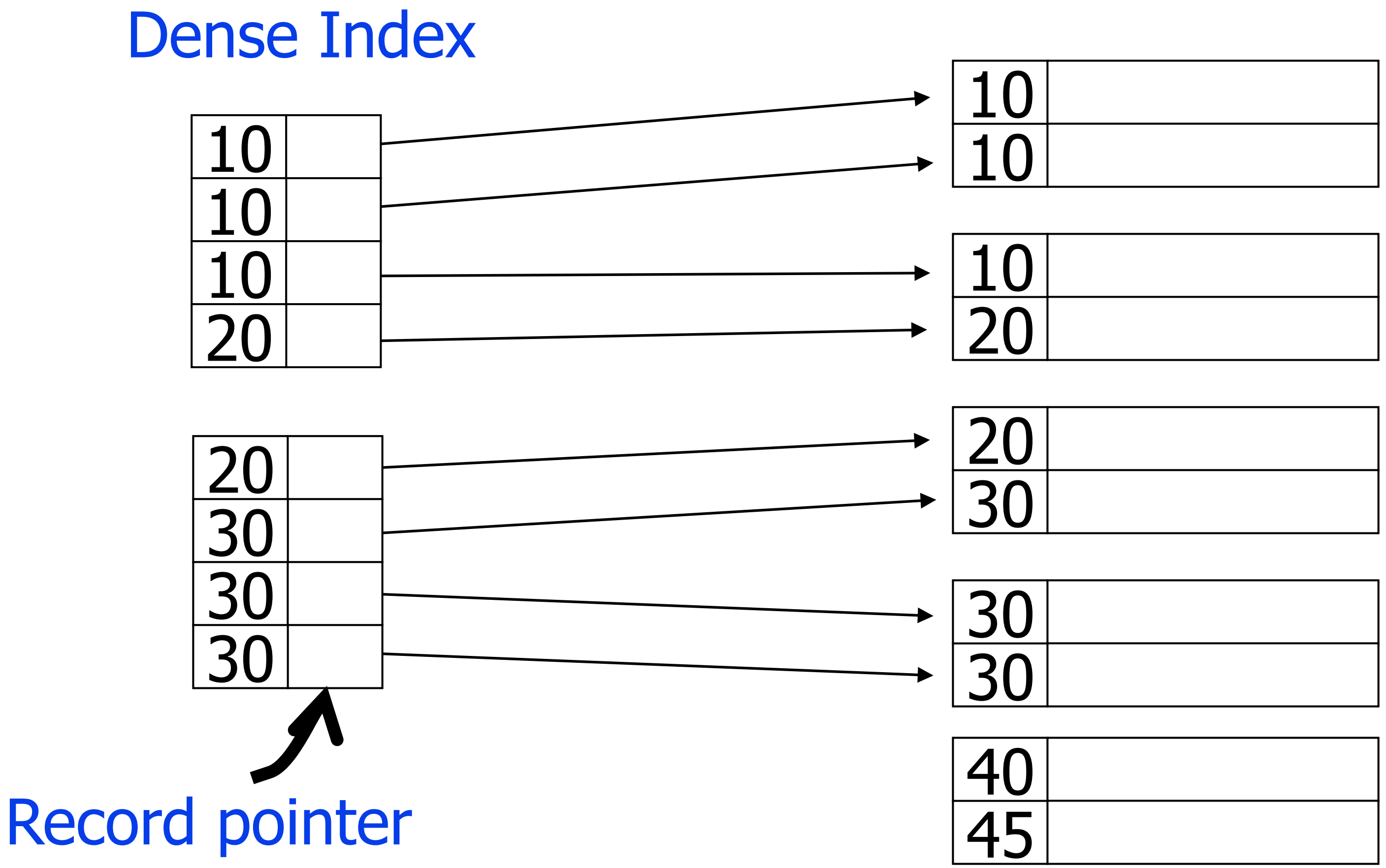
Duplicate Keys & Primary Indices

- Different records can have the same key value
- How to design the indexing structure?

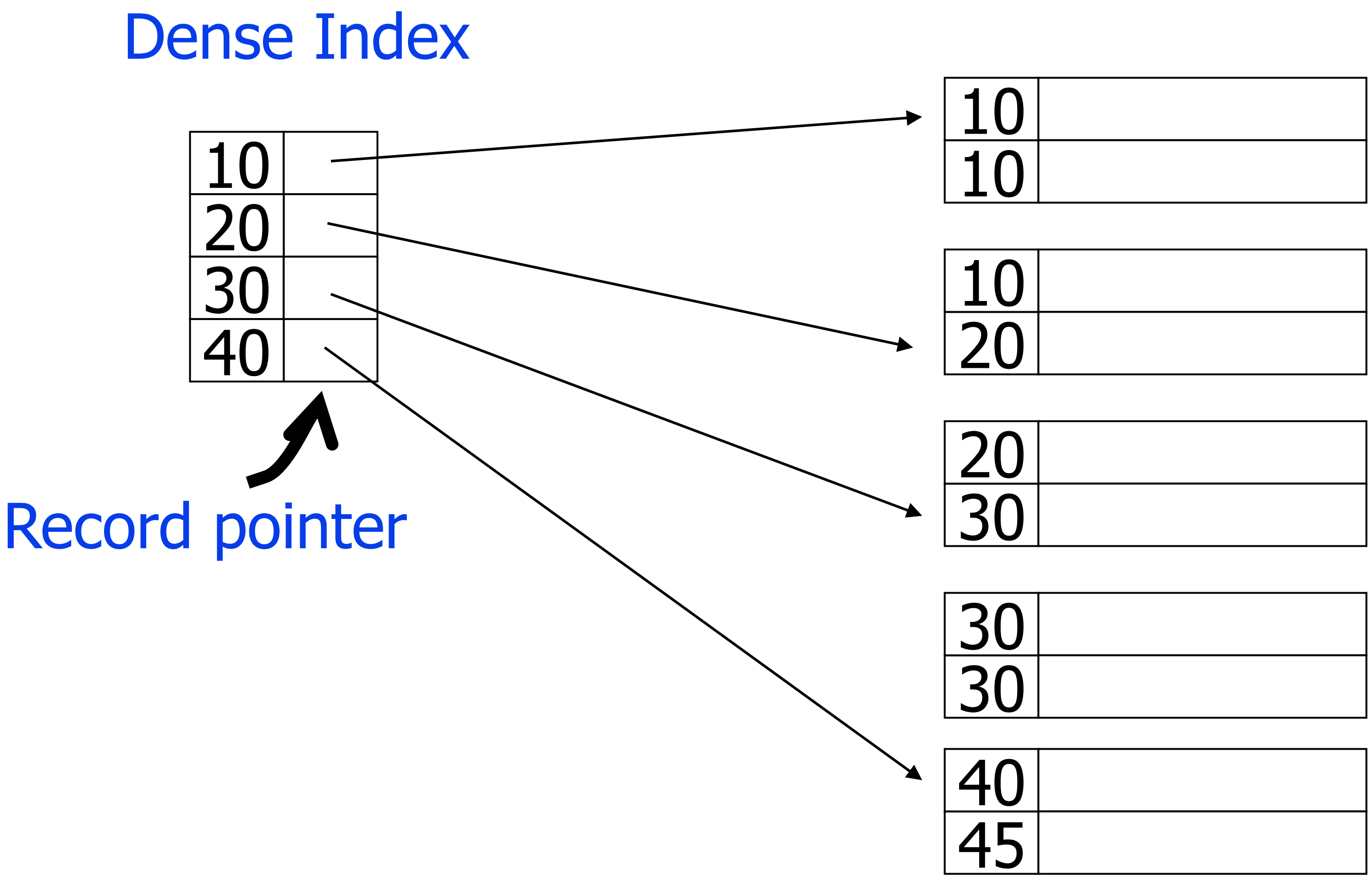


Index ????

Duplicate Keys & Dense Index



Duplicate Keys & Better Dense Index



Dense Index

10	
20	
30	
40	

Record pointer

10	
10	

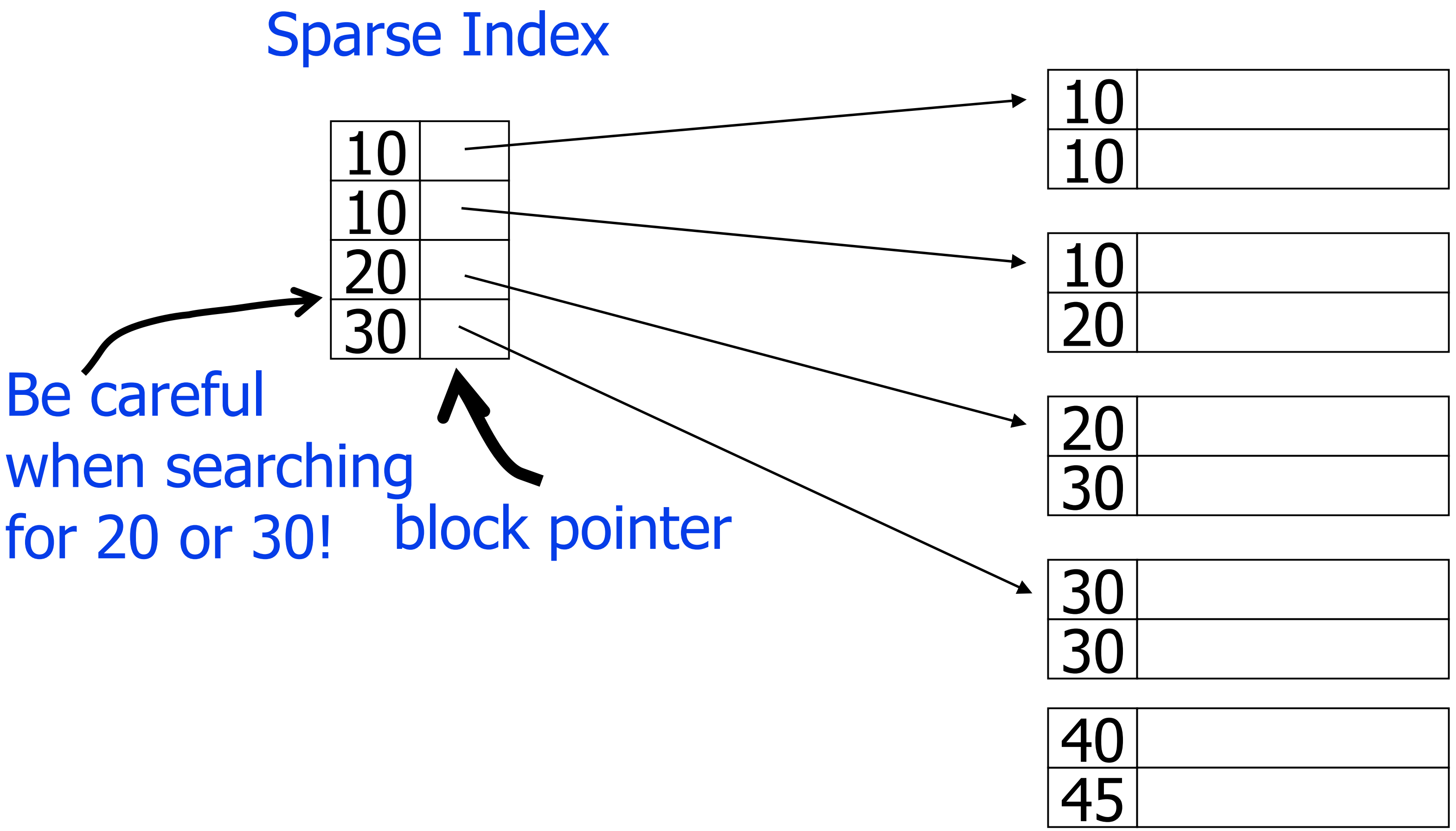
10	
20	

20	
30	

30	
30	

40	
45	

Duplicate Keys and Sparse Index



Sparse Index

10
10
20
30

Be careful when searching for 20 or 30!

block pointer

10
10

10
20

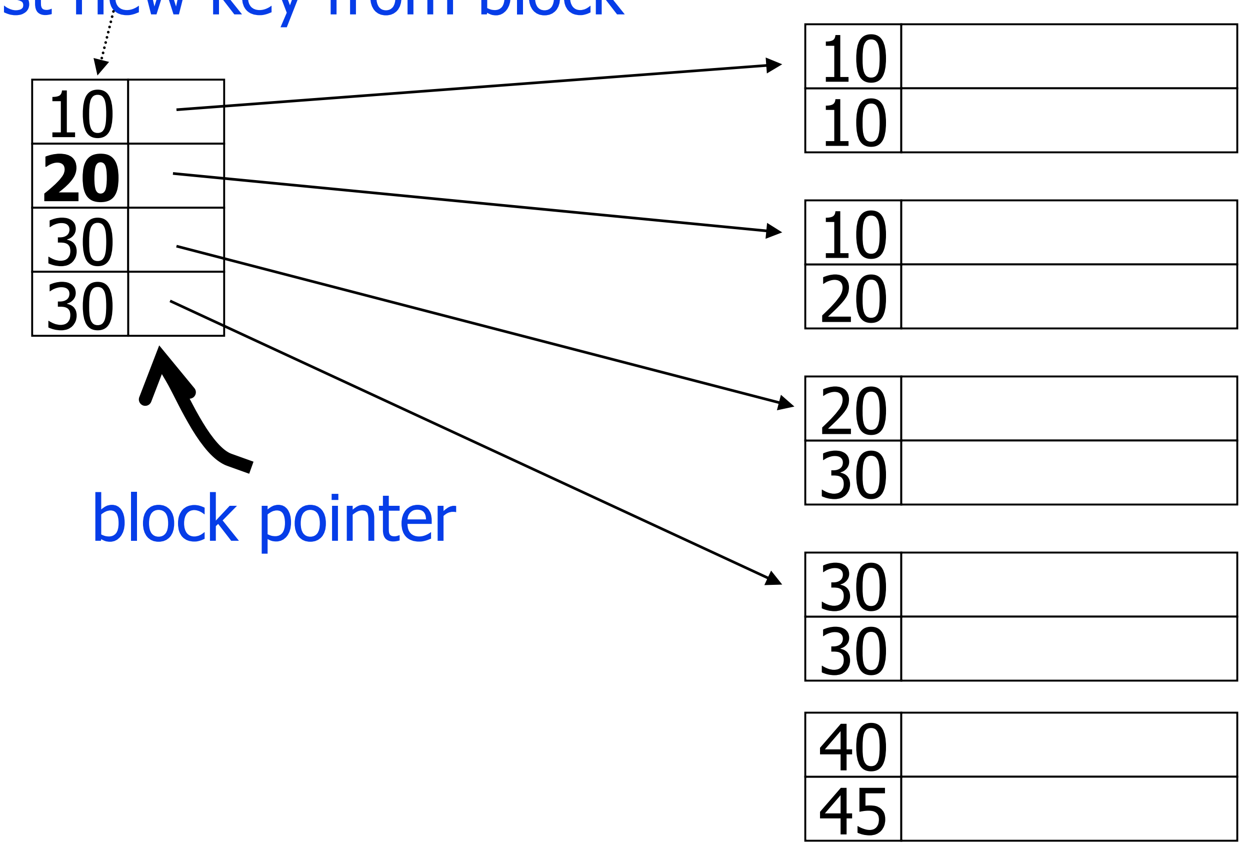
20
30

30
30

40
45

Duplicate Keys and Better Sparse Index

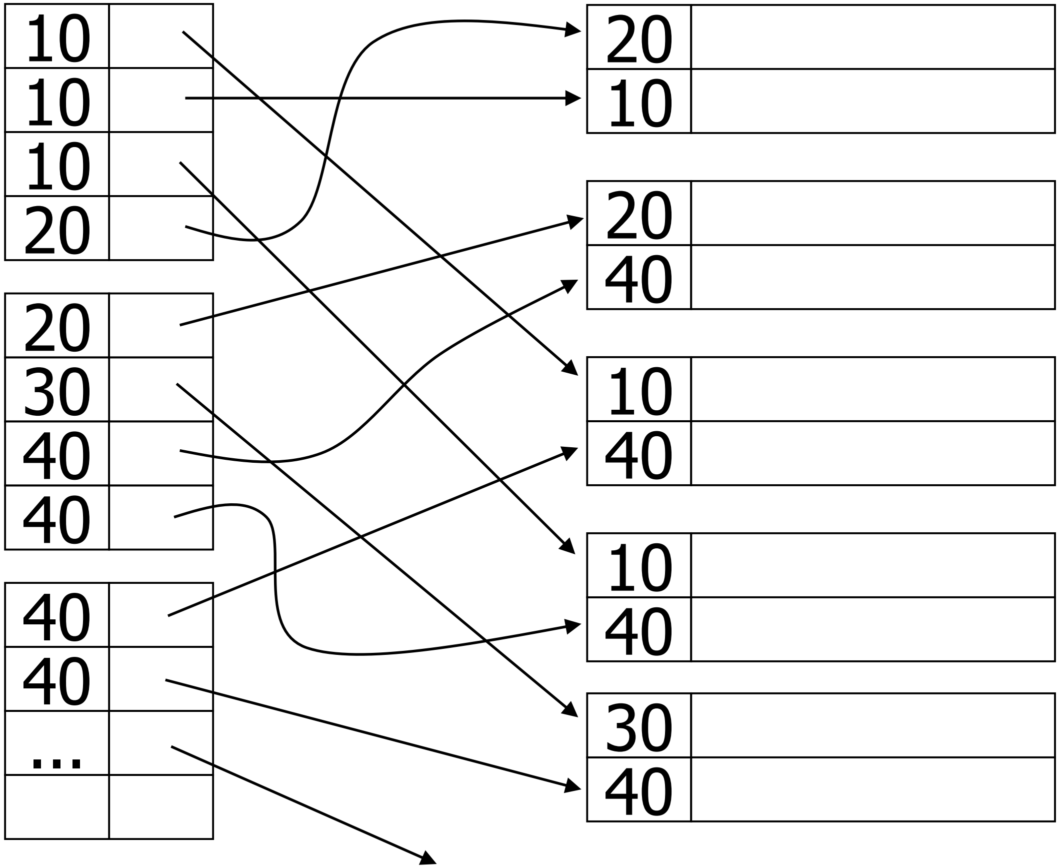
– place first new key from block



block pointer

Duplicate Values & Secondary Indices

one option...

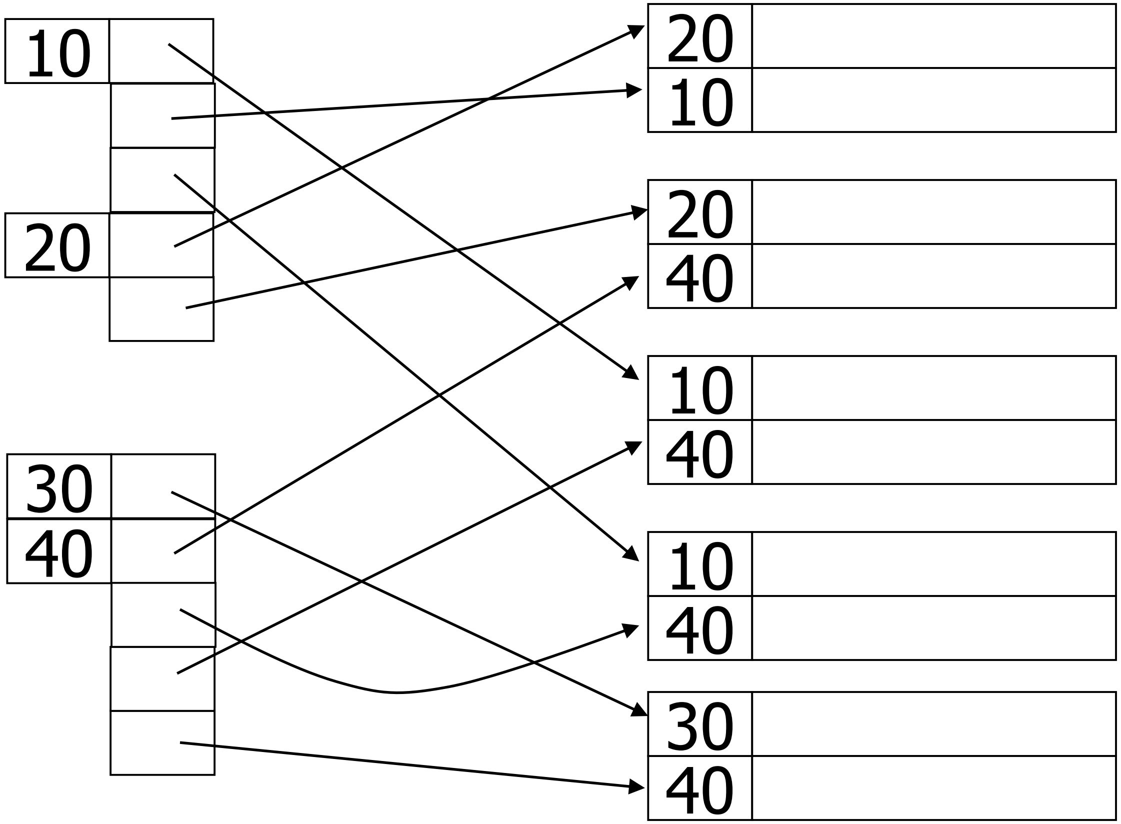


Problem:
excess overhead!

- disk space
- search time

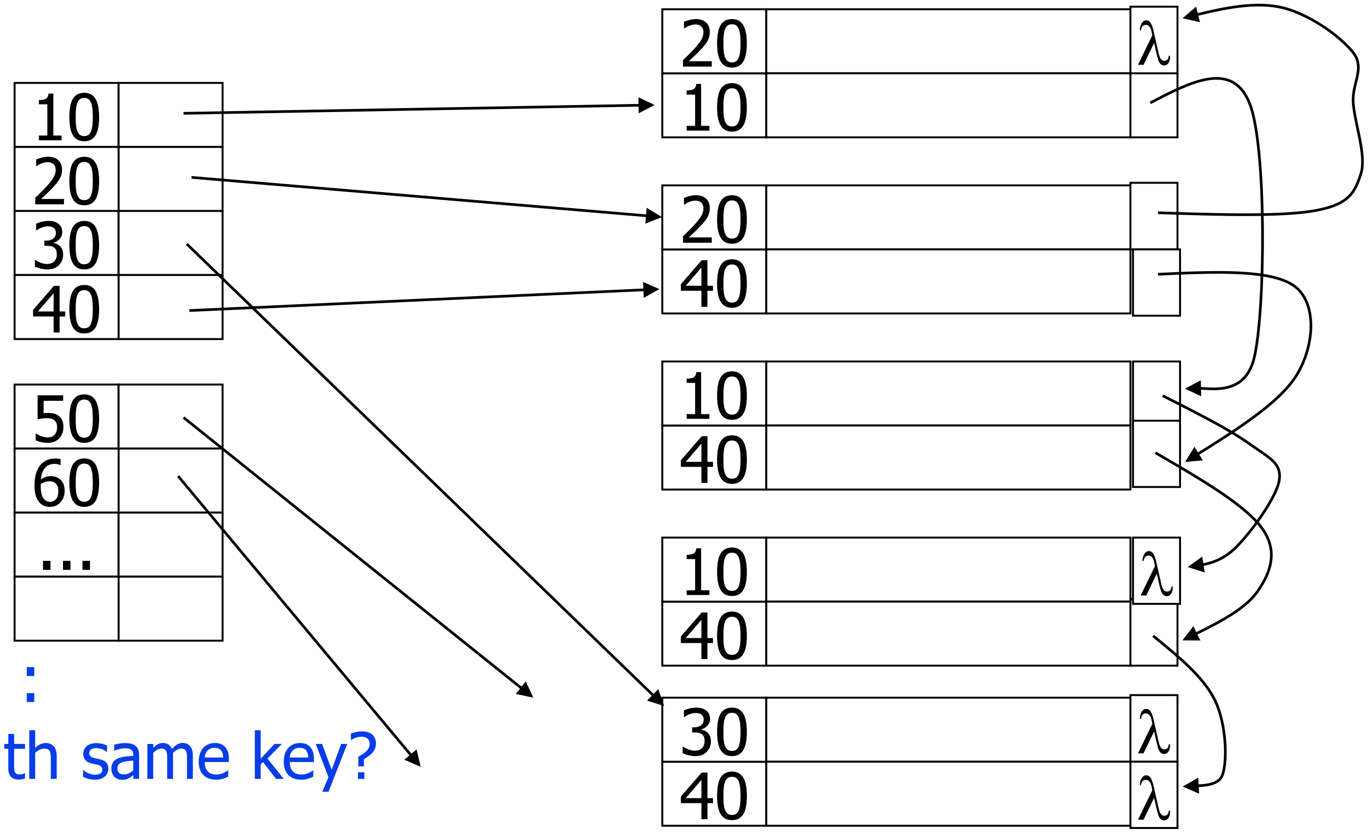
Duplicate Values & Secondary Indices

another option: lists of pointers



Problem: variable size records in index!

Duplicate Values & Secondary Indices

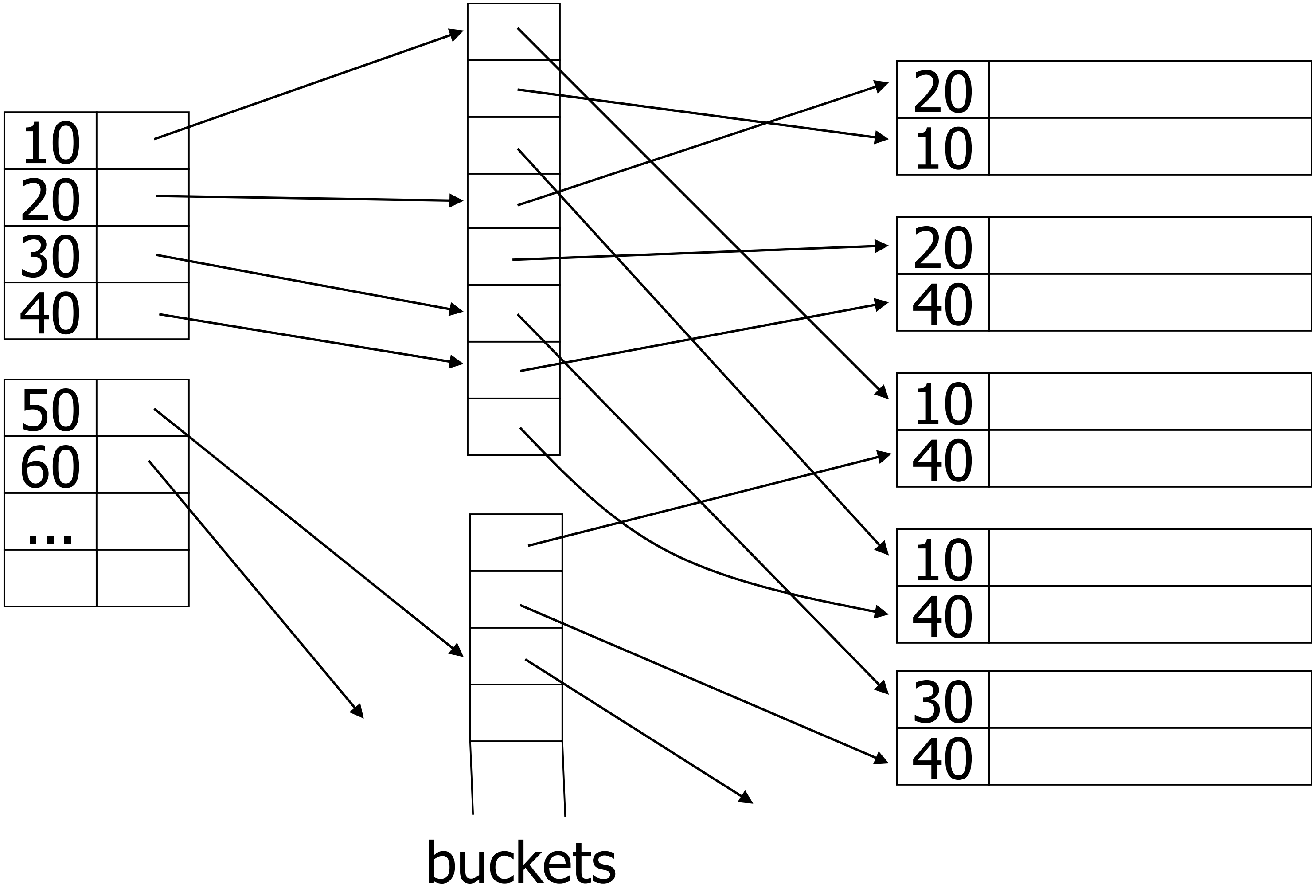


Yet another idea :
Chain records with same key?

Problems:

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

Duplicate Values & Secondary Indices



Index Update: Insertion

- Single-level index insertion
  - ◆Perform a lookup using the search-key value appearing in the record to be inserted
  - ◆Dense indices – if the search-key value does not appear in the index, insert it
  - ◆Sparse indices – if the index stores an entry for each block of the file, no change needs to be made to the index unless a new block is created
    - In the case where a new block is created, the first search-key value appearing in the new block is inserted into the index

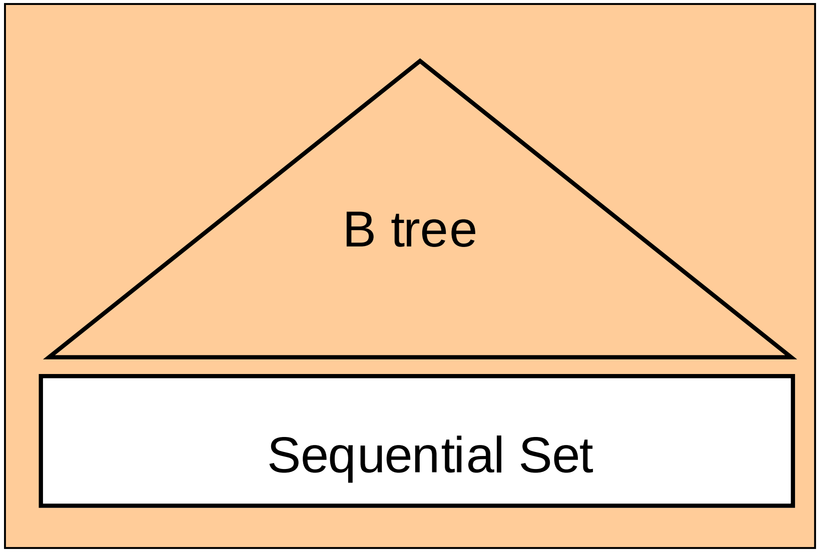
- Multilevel insertion and deletion algorithms are simple extensions of the single-level algorithms

Index Update: Deletion

- Single-level index deletion
  - If deleted record was the only record in the file with its particular search-key value, the search-key is deleted from the index
  - Dense indices – deletion of the search-key is similar to file record deletion
  - Sparse indices – if an entry for the search key exists in the index, it is deleted by replacing the entry in the index with the next search-key value in the file (in search-key order)
    - If the next search-key value already has an index entry, the entry is deleted instead of being replaced

Index Data Structures

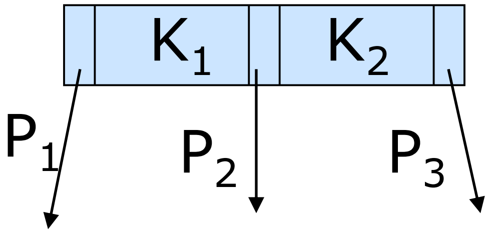
- Most index data structures can be viewed as trees
- In general, the root of this tree will always be in main memory, while the leaves will be located on disk
  - The performance of a data structure depends on the number of nodes in the average path from the root to the leaf
  - Data structures with high fan-out (maximum number of children of an internal node) are thus preferred
- B+ tree: Amendment to B-tree
  - addresses for data records are in the leaves and nowhere else
  - a key value is stored once, with the address of the associated data record



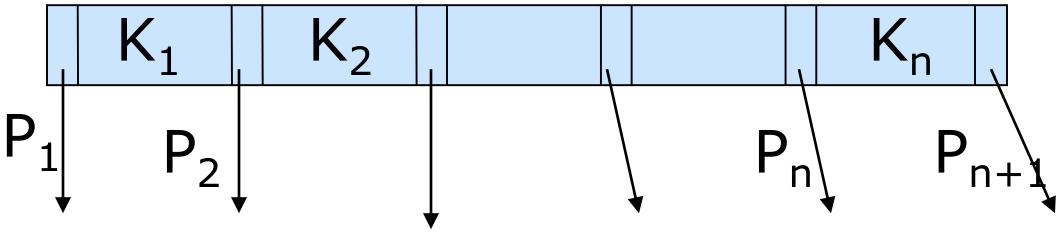
B+ Tree Properties

- A B+ Tree is a balanced tree whose leaves contain a sequence of key-pointer pairs
  - Same height for paths from root to leaf
  - Given a search-key K, nearly same access time for different K values
- A B+ Tree is constructed by **parameter n**, the maximum key capacity of a node. **n+1** is also called the B+ tree order or fan-out
  - Non-root nodes must always be at least half full (or half empty)
    - Each internal node (non-leaf and non-root) has $\lceil$**(n+1)/2**$\rceil$ to **n+1** pointers and $\lceil$**(n+1)/2**$\rceil$**–1** to **n** search-key values
    - The root node has at least **2** pointers or else is a leaf
    - Each leaf node has $\lfloor$**(n+1)/2**$\rfloor$ to **n** search-key values

Case for n=2

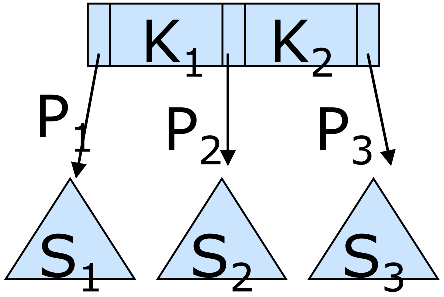


General case for n

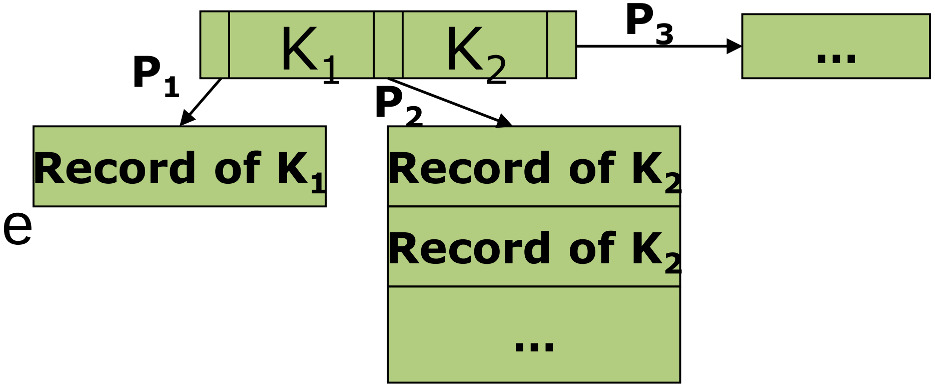


B+ Tree Properties

- Search keys are sorted in order
  - $K_1 < K_2 < \ldots < K_n$
- Non-leaf Node
  - For each search-key value K in sub-tree S_i pointed by P_i , $K_{i-1} \le K < K_i$
    - e.g. (key values in S_1) < K_1
    - e.g. $K_1 \le$ (key values in S_2) < K_2

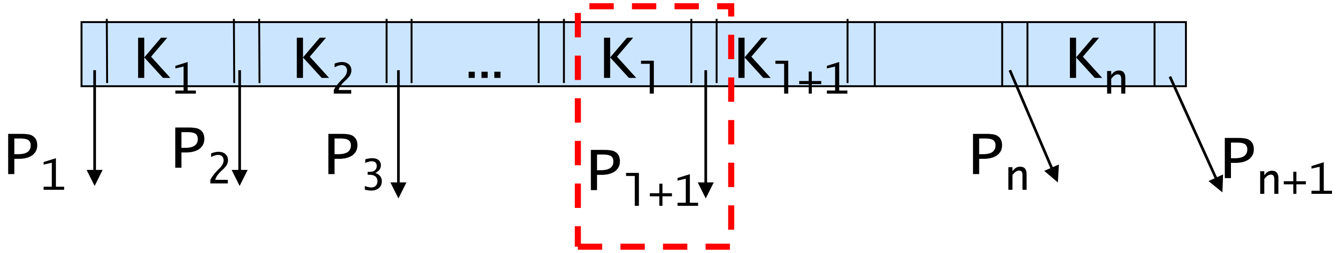


- Leaf Node
  - P_i points to record or bucket with search key value K_i
  - P_{i+1} points to the neighbor leaf node

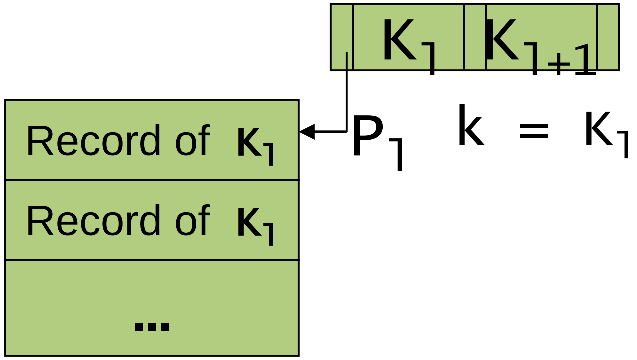


B+ Tree Searching

- Given a search-key value k
  - Start from the root, look for the largest search-key value (K_l) in the node such that $K_l \leq k < K_{l+1}$
  - Follow pointer P_{l+1} to next level, until you reach a leaf node



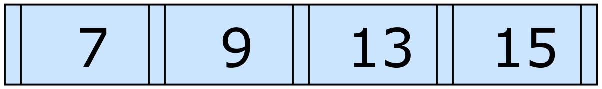
  - If k is found to be equal to K_l in the leaf, follow P_l to search the record or bucket



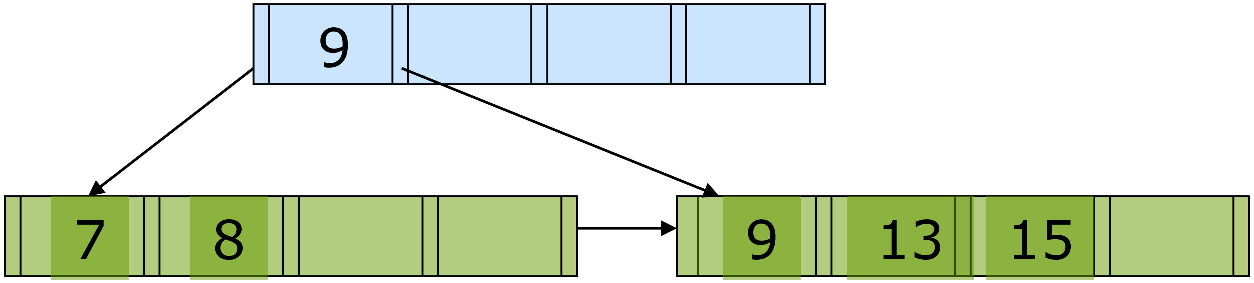
B+ Tree Insertion

- Overflow
  - When number of search-key values exceed **n**

Insert 8

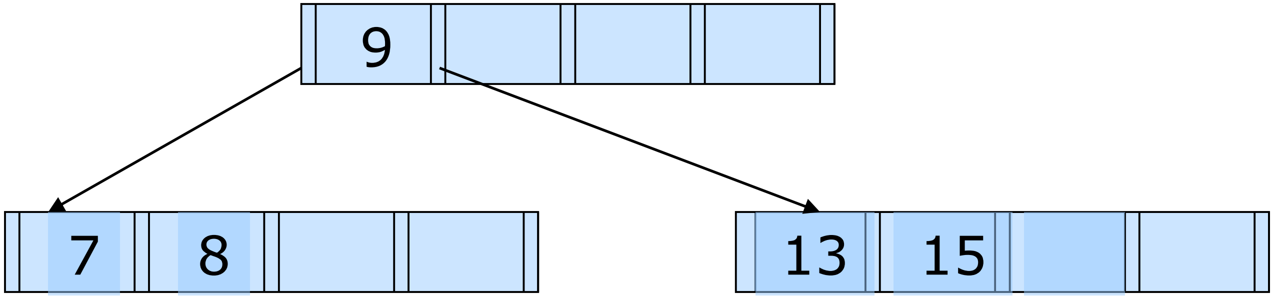


- Leaf Node
  - Split into two nodes:
    - 1st node contains (the first) $\lfloor \mathbf{(n+1)/2} \rfloor$ values
    - 2nd node contains the remaining values
    - Copy the smallest search-key value of the 2nd node to parent node



B+ Tree Insertion (cont.)

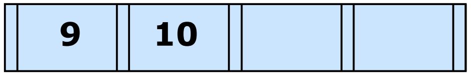
- Non-Leaf Node
  - Split into two nodes:
    - 1st node contains (the first) $\lceil (n+1)/2 \rceil - 1$ values
    - Move the smallest of the remaining values, together with pointer, to the parent
    - 2nd node contains the remaining values



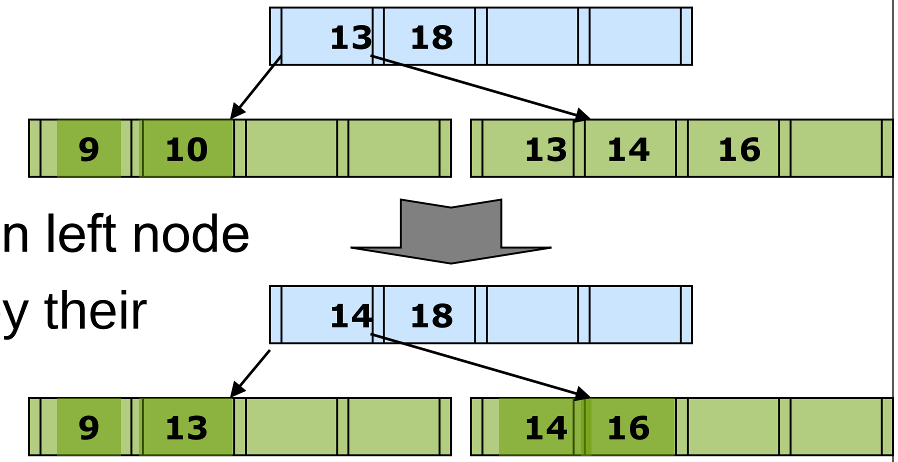
B+ Tree Deletion

- Underflow
  - When number of search-key values < $\lceil (n+1)/2 \rceil - 1$

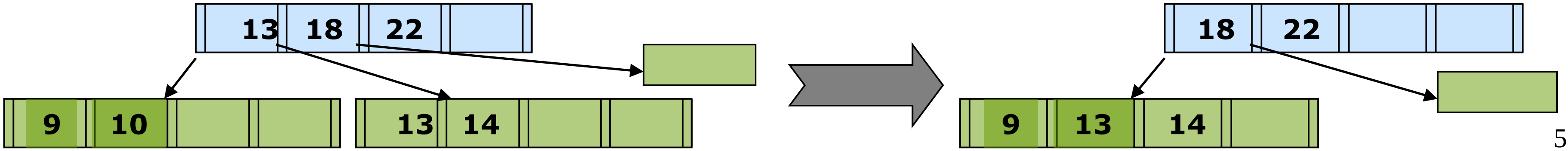
Delete 10



- Leaf Node
  - Redistribute to sibling
    - Still, right node shouldn’t have less than left node
    - Replace the between-value in parent by their smallest value of the right node

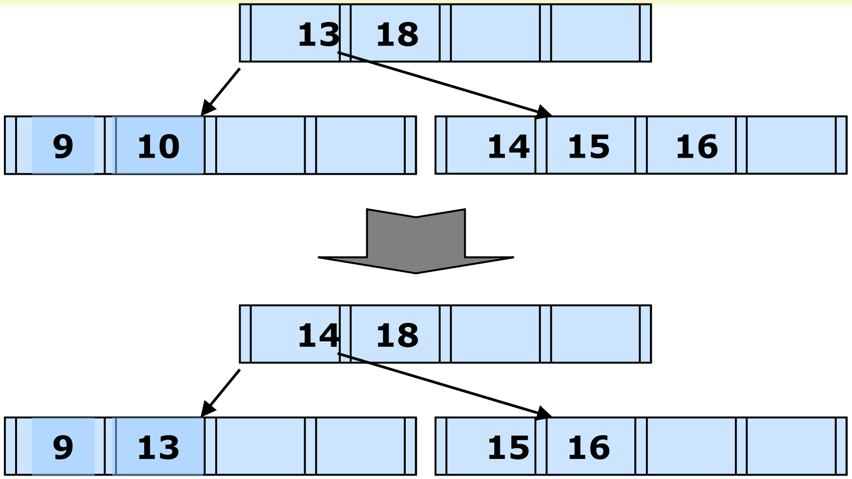


  - Merge (the two siblings contain too few entries)
    - Move all values, pointers to left node
    - Remove the between-value in parent

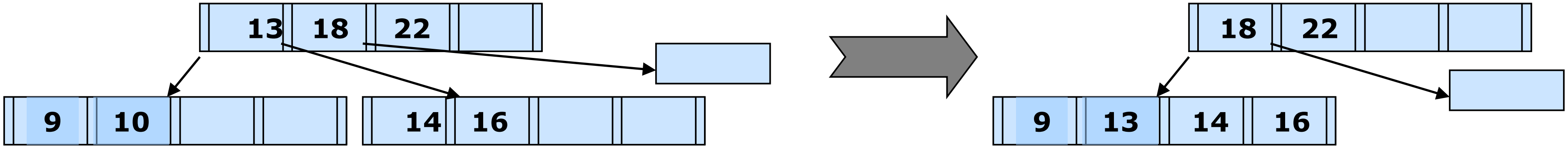


B+ Tree Deletion (cont.)

- Non-Leaf Node
  - Redistribute to sibling
    - Through parent
    - Right node not less than left node



  - Merge (the two siblings contain too few entries)
    - Bring down parent
    - Move all values, pointers to left node
    - Delete the right node, and pointers in parent

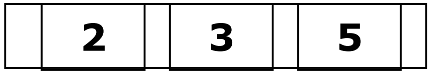


Example on B+ Tree

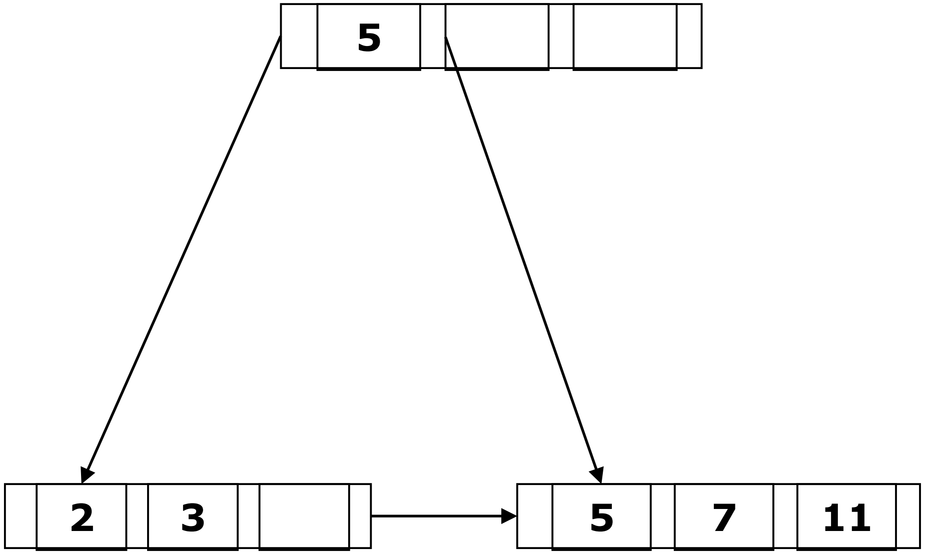
- Construct a B+ tree
  - Assume the tree is initially empty
  - All paths from root to leaf are of the same length
  - The number of pointers that will fit in one node (the order/fan-out of the tree) is four (**n**=3)
  - Leaf nodes must have between 2 and 3 values ($\lfloor$**(n+1)/2**$\rfloor$ and **n**)
  - Non-leaf nodes other than root must have between 2 and 4 children ($\lceil$**(n+1)/2**$\rceil$ and **n+1**)
  - Root must have at least 2 children

Insertions

- Insert 2, 3, 5

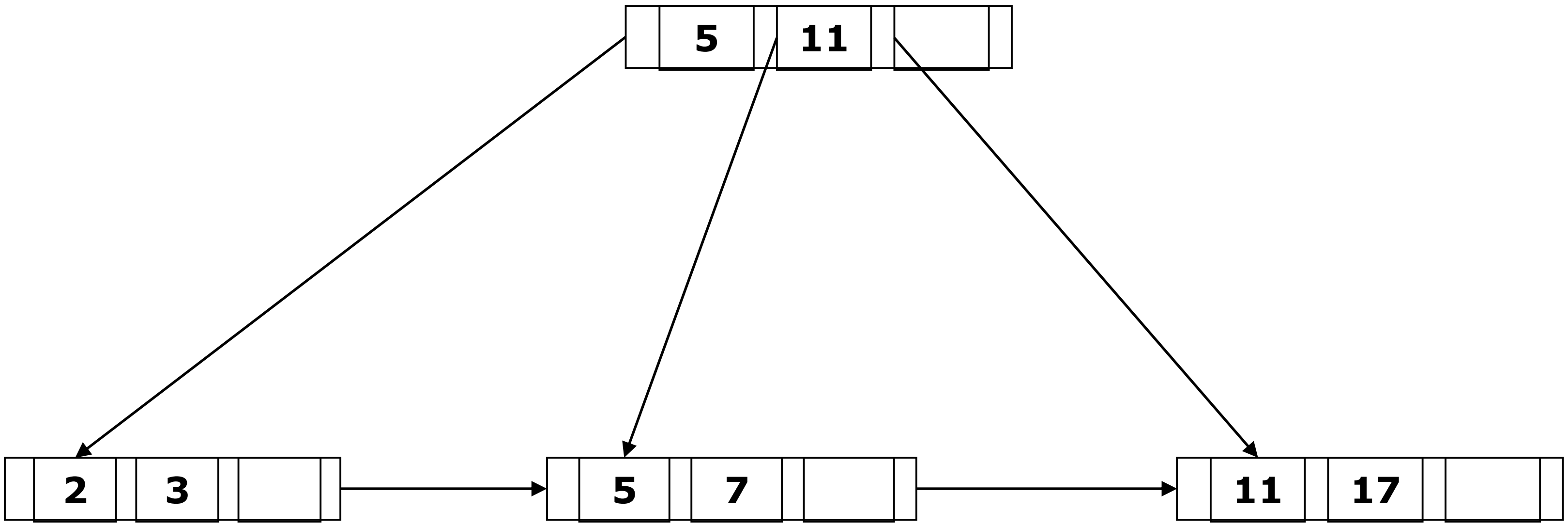


- Insert 7, 11



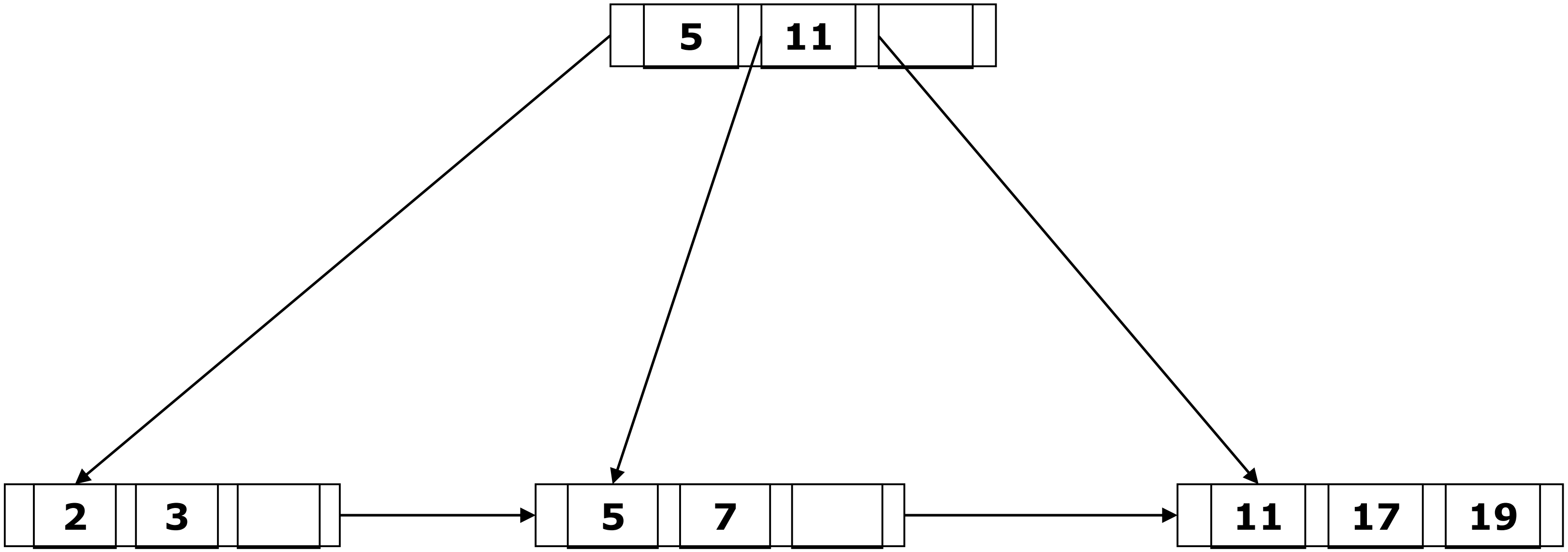
Next: Insert 17

Insert 17



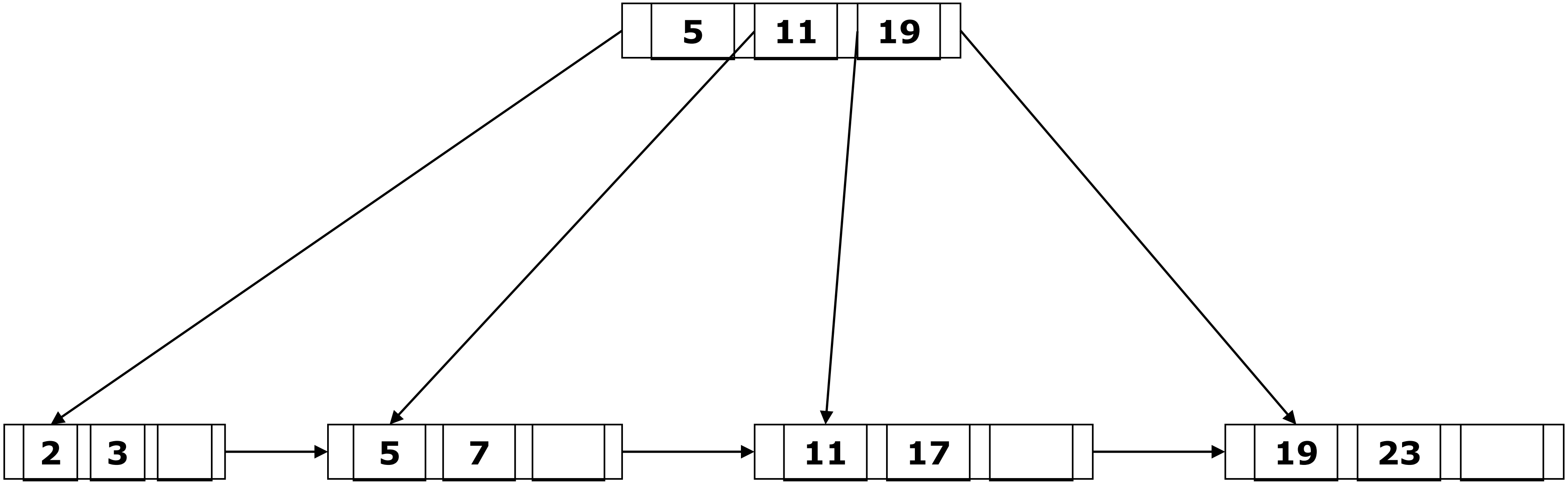
Next: Insert 19

Insert 19



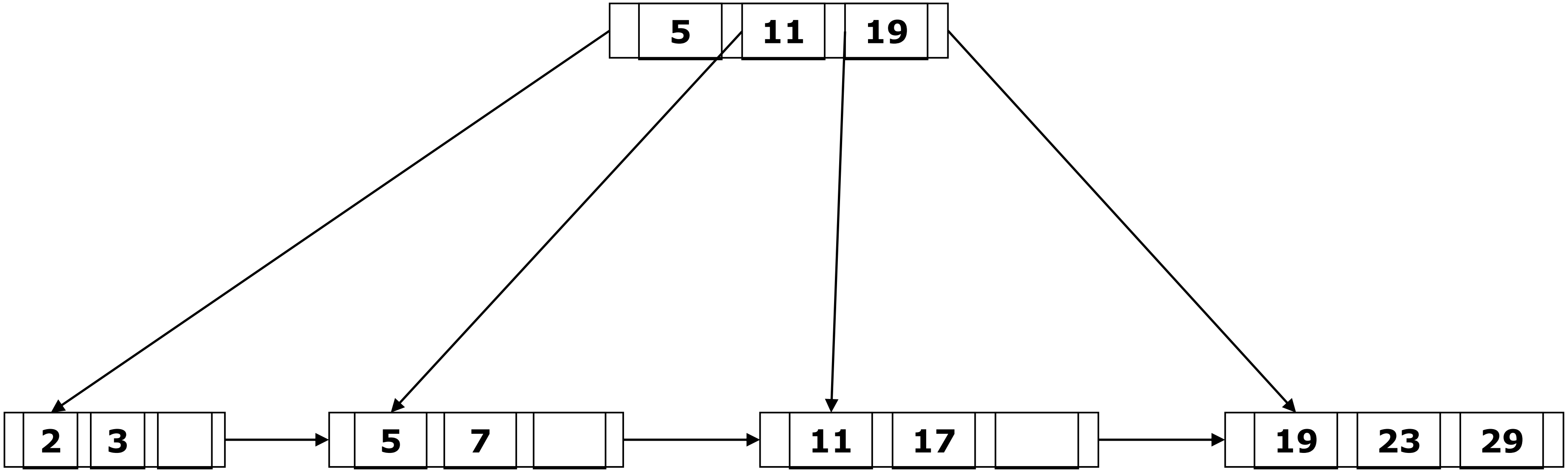
Next: Insert 23

Insert 23



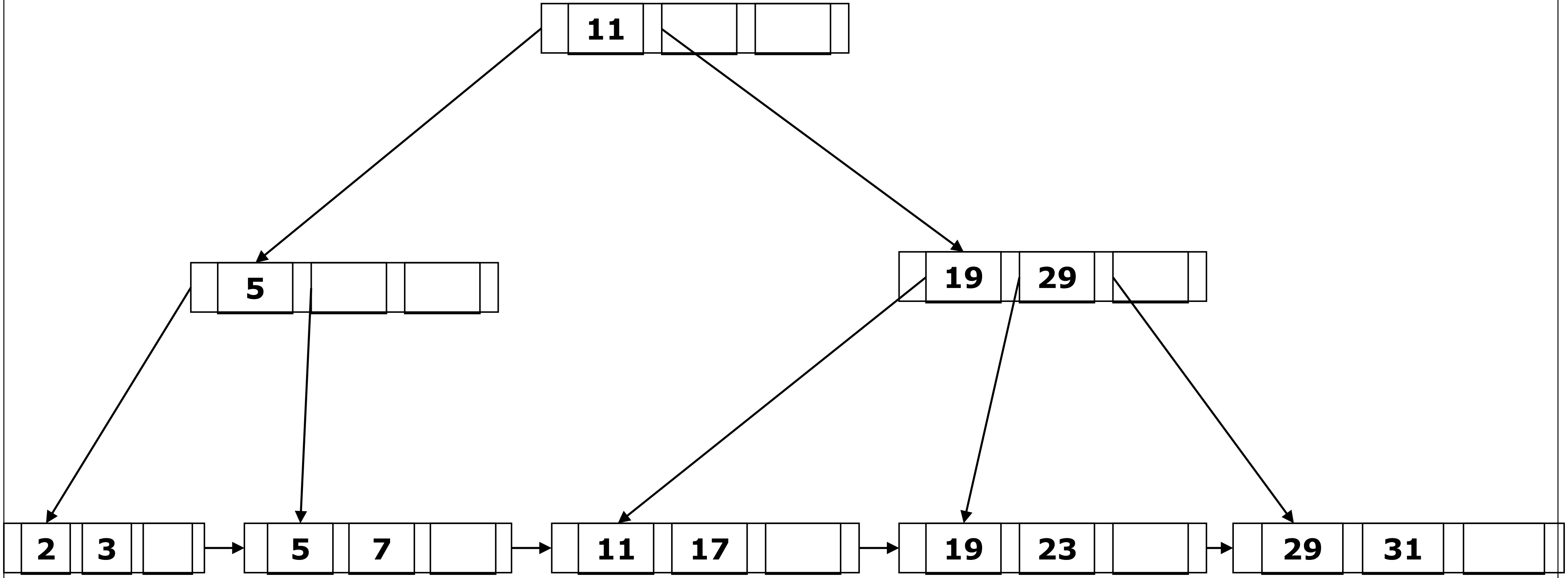
Next: Insert 29

Insert 29



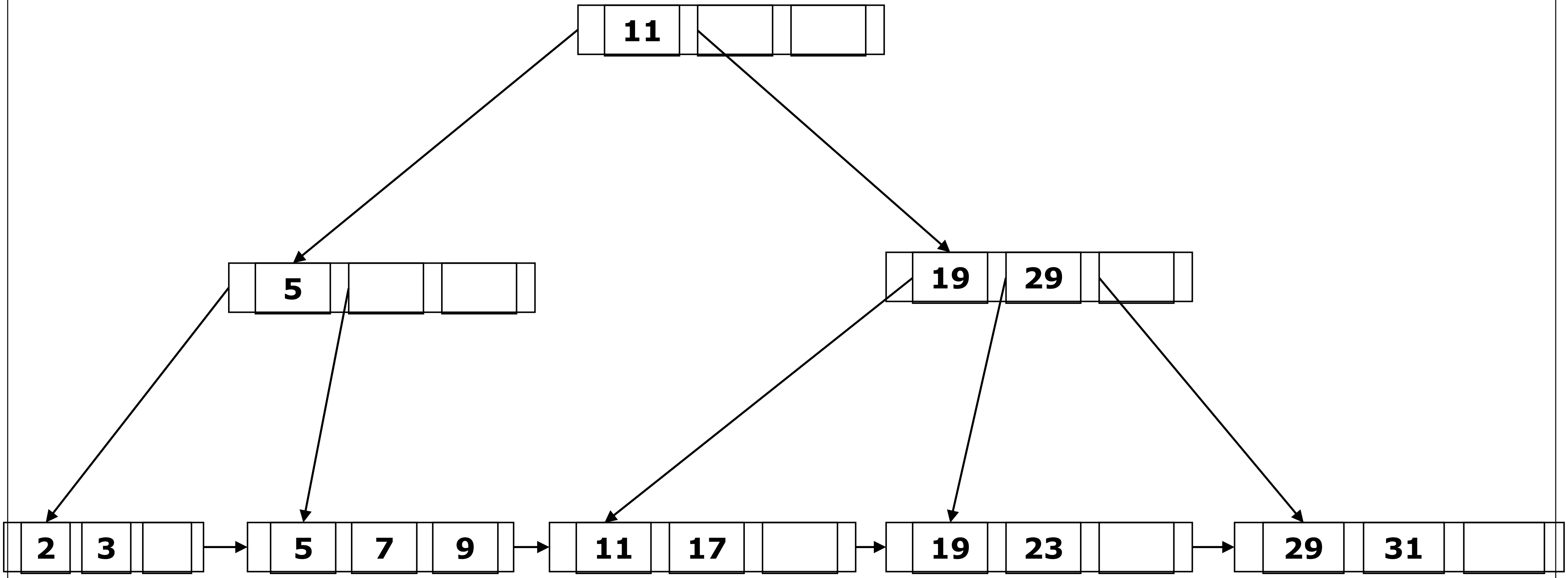
Next: Insert 31

Insert 31



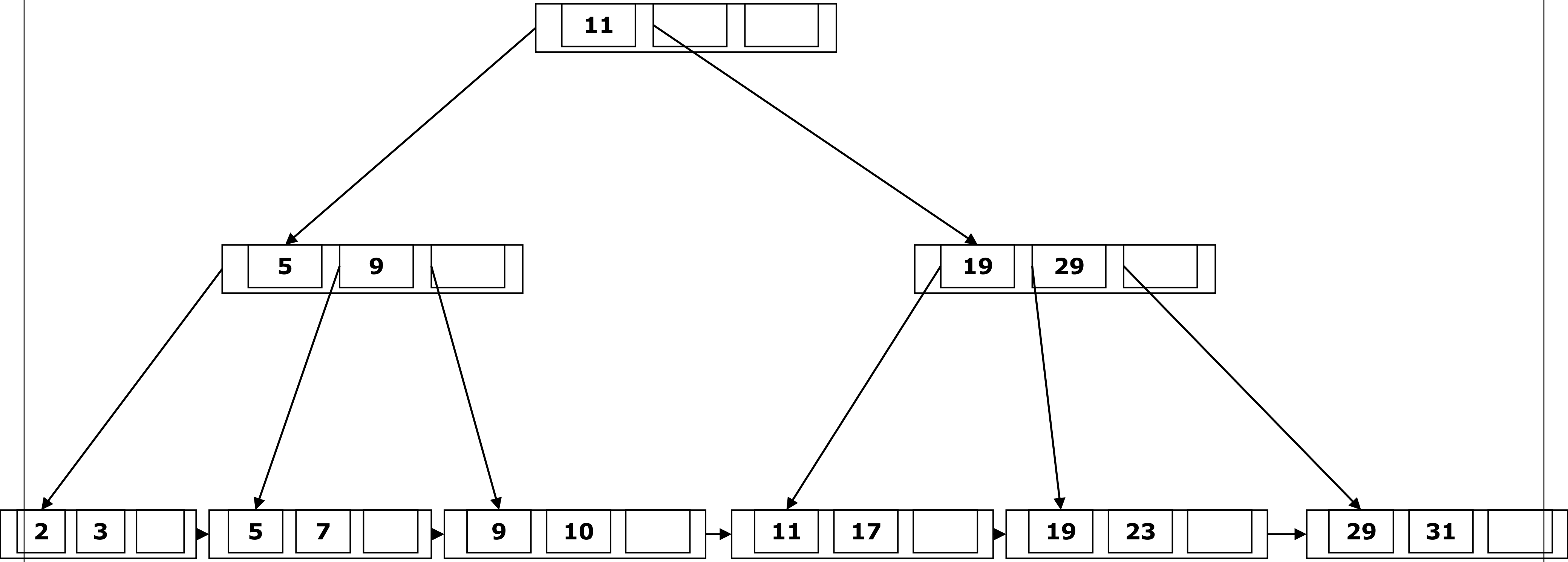
Next: Insert 9

Insert 9



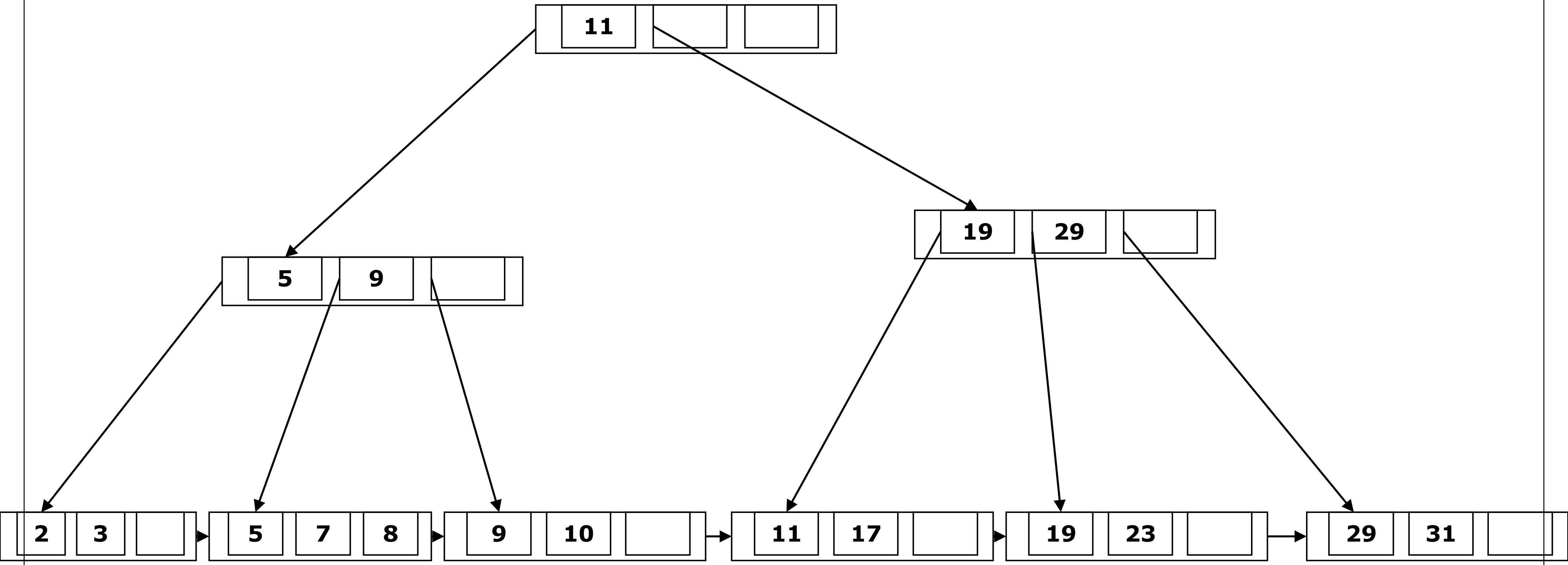
Next: Insert 10

Insert 10



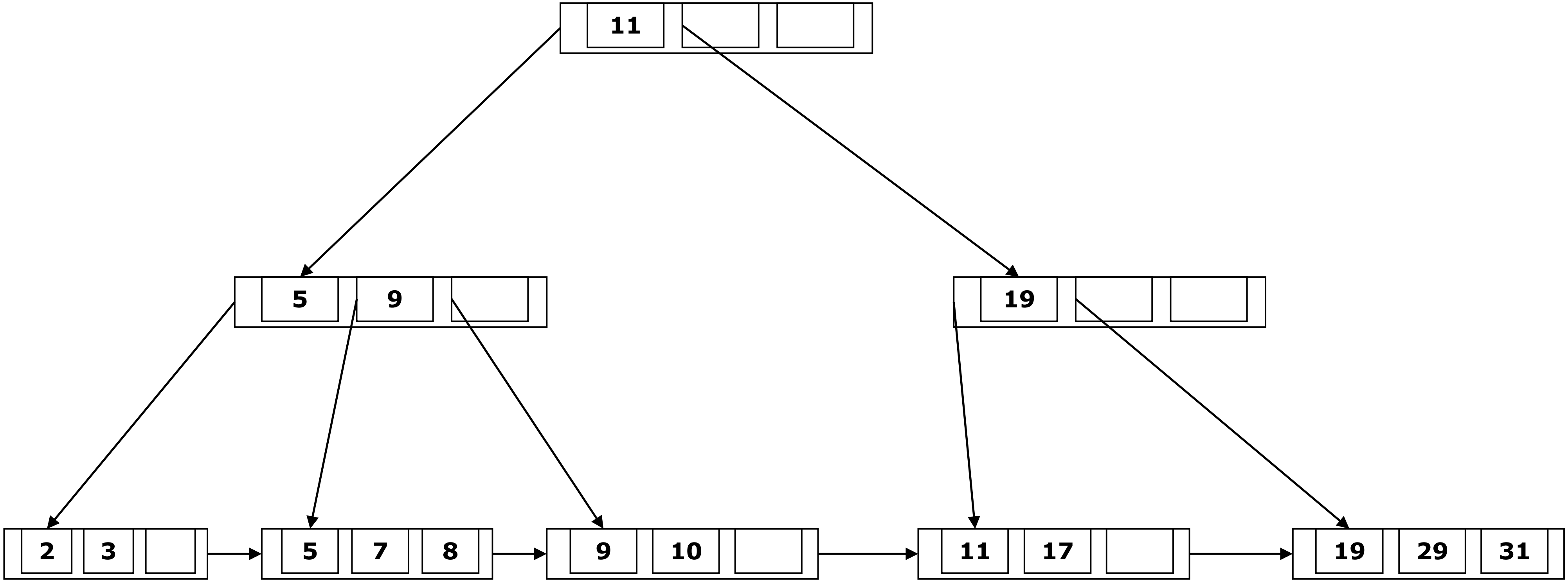
Next: Insert 8

Insert 8



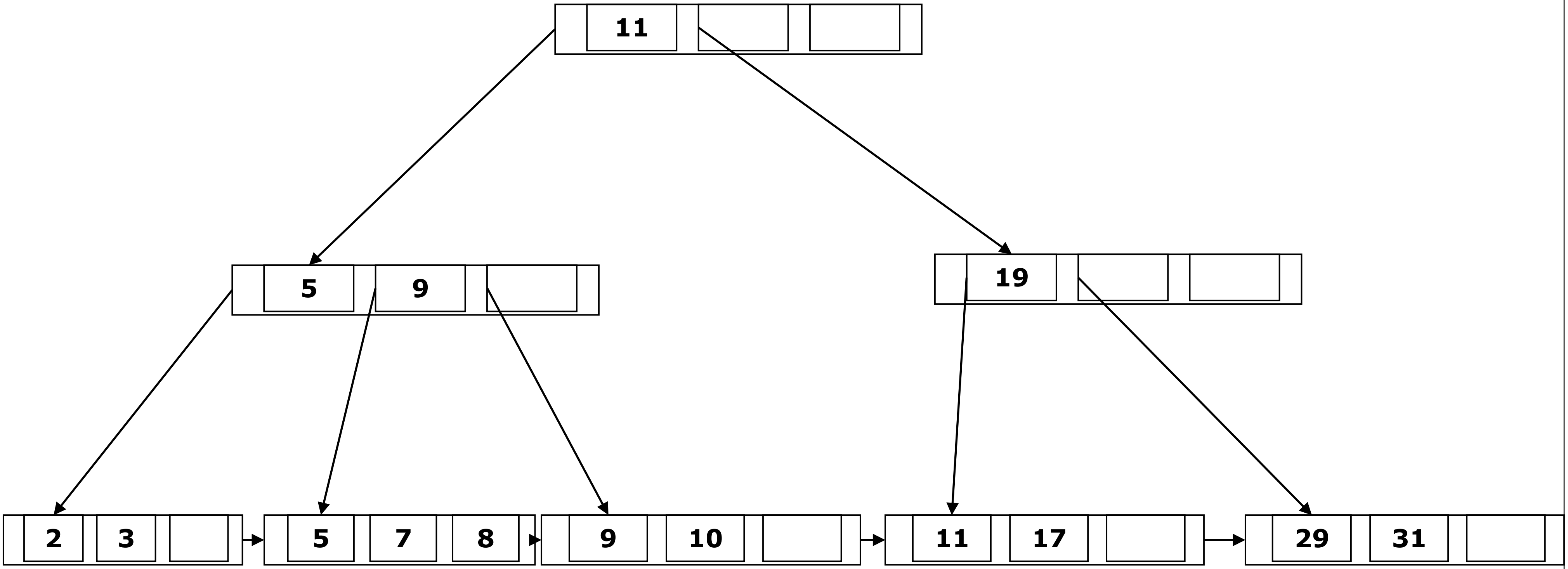
Next: Delete 23

Delete 23



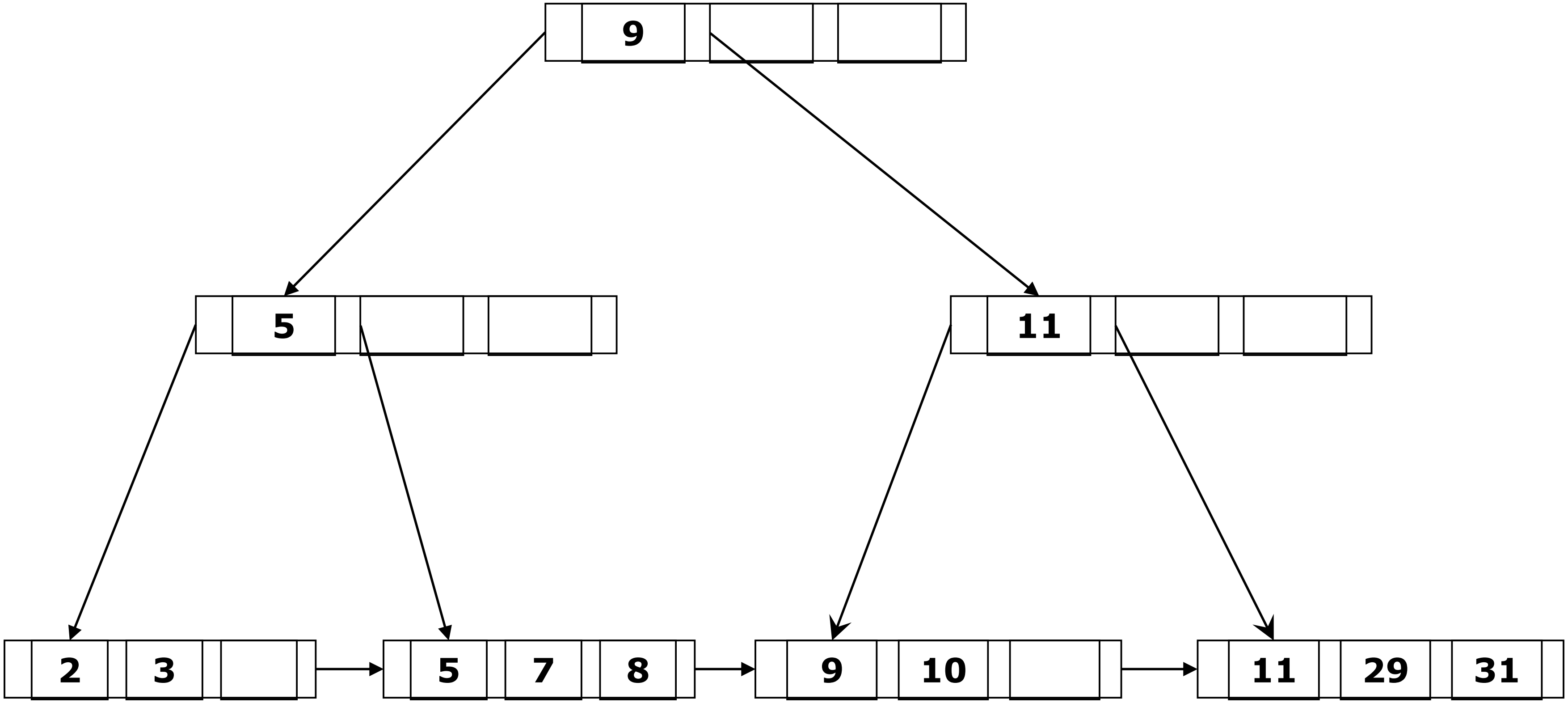
Next: Delete 19

Delete 19



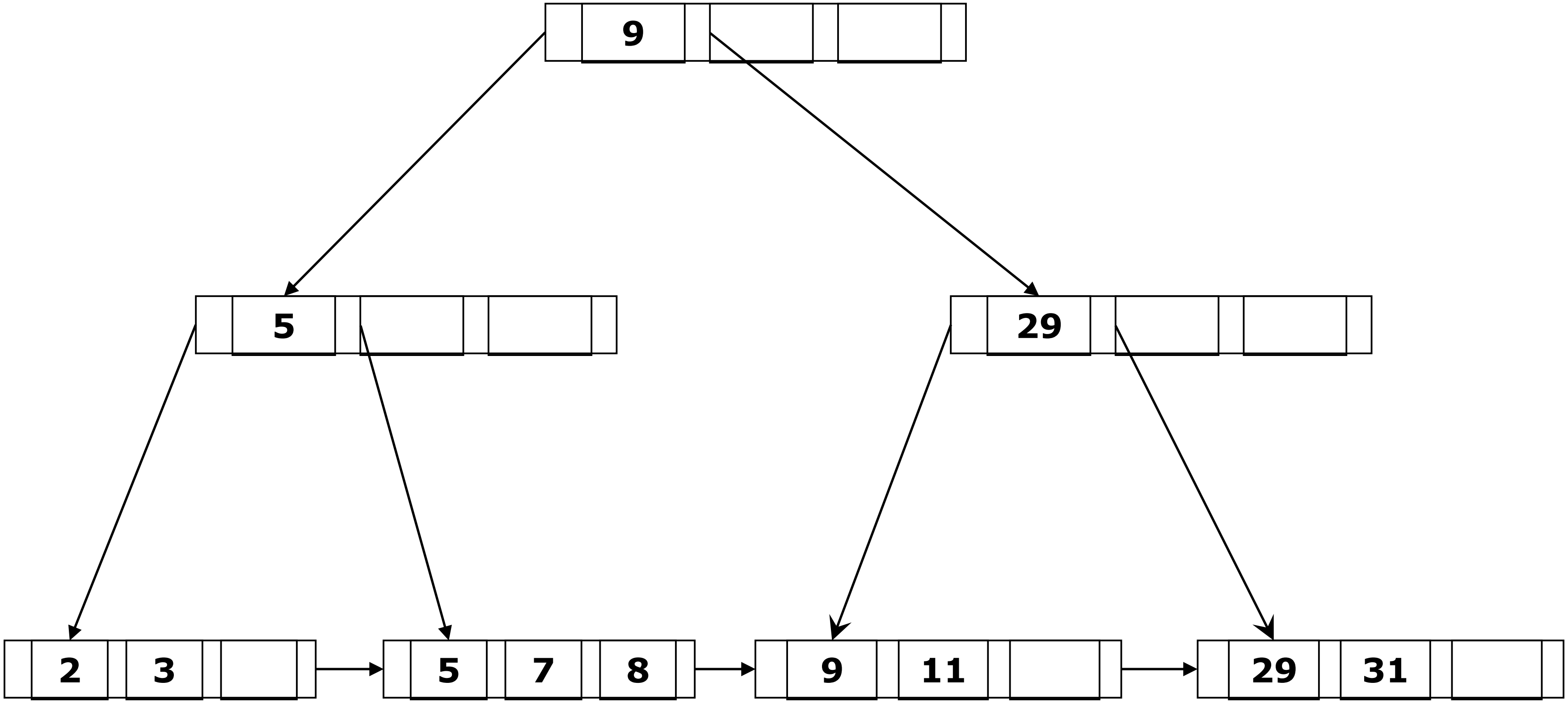
Next: Delete 17

Delete 17



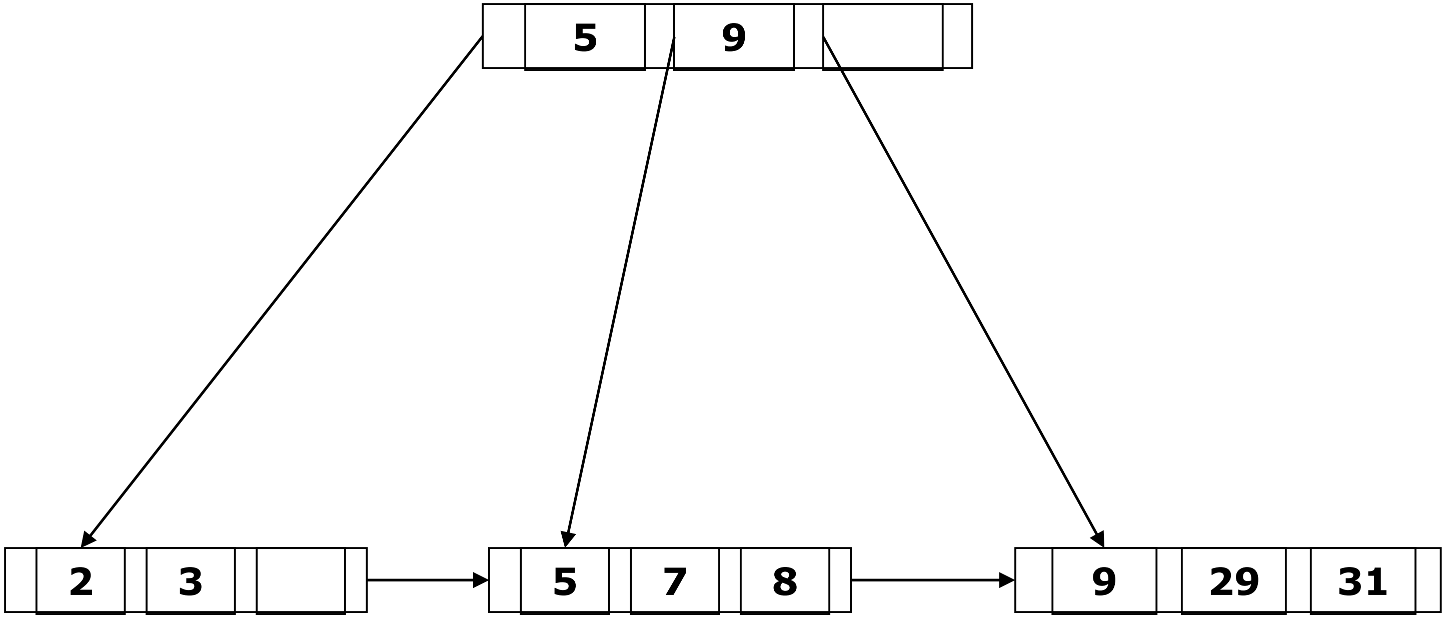
Next: Delete 10

Delete 10



Next: Delete 11

Delete 11



B+ Trees: Second Example

- Assume a student table: `Student(name, age, gpa, major)`

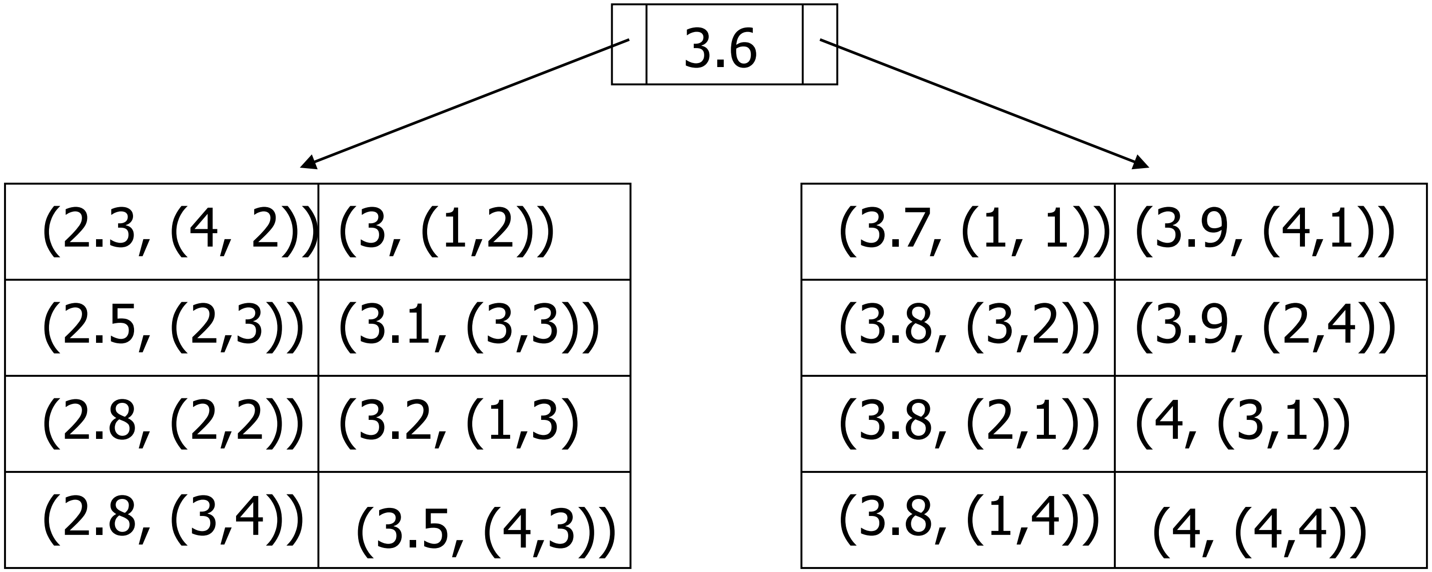
  tuples(`Student`) = 16

  pages(`Student`) = 4

Bob, 21, 3.7, CS	Kane, 19, 3.8, ME	Louis, 32, 4, LS	Chris, 22, 3.9, CS
Mary, 24, 3, ECE	Lam, 22, 2.8, ME	Martha, 29, 3.8, CS	Chad, 28, 2.3, LS
Tom, 20, 3.2, EE	Chang, 18, 2.5, CS	James, 24, 3.1, ME	Leila, 20, 3.5, LS
Kathy, 18, 3.8, LS	Vera, 17, 3.9, EE	Pat, 19, 2.8, EE	Shideh, 16, 4, CS

Non-Clustered Secondary B+ Tree

- A non-clustered secondary B+ tree on the gpa attribute



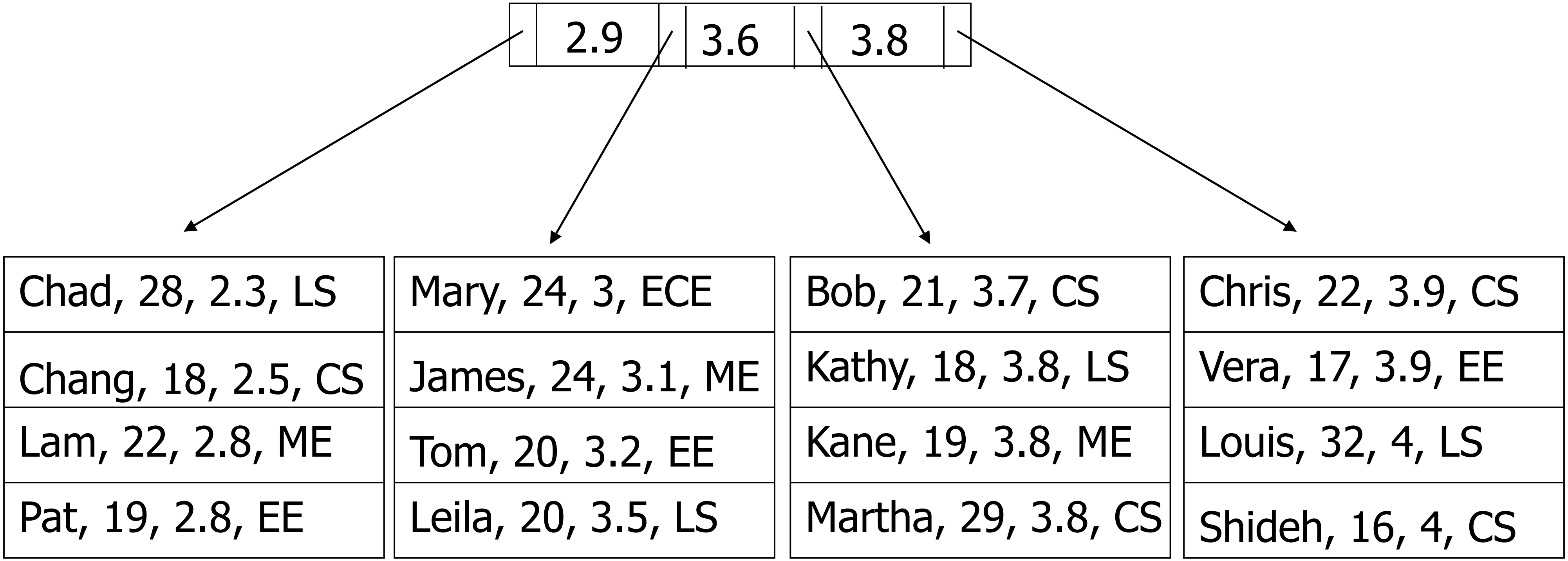
(2.3, (4, 2))	(3, (1,2))
(2.5, (2,3))	(3.1, (3,3))
(2.8, (2,2))	(3.2, (1,3)
(2.8, (3,4))	(3.5, (4,3))

(3.7, (1, 1))	(3.9, (4,1))
(3.8, (3,2))	(3.9, (2,4))
(3.8, (2,1))	(4, (3,1))
(3.8, (1,4))	(4, (4,4))

Bob, 21, 3.7, CS	Kane, 19, 3.8, ME	Louis, 32, 4, LS	Chris, 22, 3.9, CS
Mary, 24, 3, ECE	Lam, 22, 2.8, ME	Martha, 29, 3.8, CS	Chad, 28, 2.3, LS
Tom, 20, 3.2, EE	Chang, 18, 2.5, CS	James, 24, 3.1, ME	Leila, 20, 3.5, LS
Kathy, 18, 3.8, LS	Vera, 17, 3.9, EE	Pat, 19, 2.8, EE	Shideh, 16, 4, CS

Clustered B+ Tree

- A clustered B+ tree on the gpa attribute



2.9	3.6	3.8

Chad, 28, 2.3, LS	Mary, 24, 3, ECE	Bob, 21, 3.7, CS	Chris, 22, 3.9, CS
Chang, 18, 2.5, CS	James, 24, 3.1, ME	Kathy, 18, 3.8, LS	Vera, 17, 3.9, EE
Lam, 22, 2.8, ME	Tom, 20, 3.2, EE	Kane, 19, 3.8, ME	Louis, 32, 4, LS
Pat, 19, 2.8, EE	Leila, 20, 3.5, LS	Martha, 29, 3.8, CS	Shideh, 16, 4, CS

- It is impossible to have a clustered secondary B+ tree on an attribute

Clarifications on B+ Trees

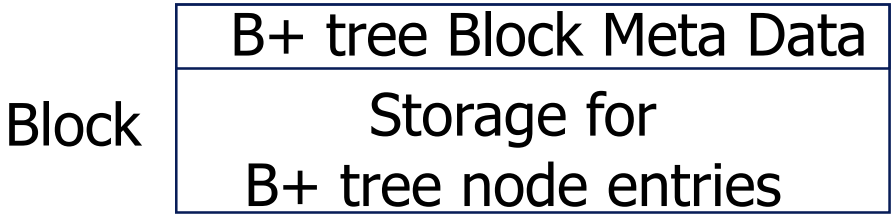
- B+ trees can be used to store relations as well as index structures
- In the drawn B+ trees we assume (this is not the only scheme) that an internal node with q pointers stores the maximum keys of each of the first q-1 sub-trees it is pointing to; that is, it contains q-1 keys
- Before a B+ tree can be generated the following parameters have to be chosen (based on the available block size; it is assumed one node is stored in one block):
  - the order of the tree (**n+1**) (n+1 is the maximum number of pointers an intermediate node might have; a non-root node must have between $\lceil (n+1)/2 \rceil$ and n+1 pointers)
  - the maximum number **m** of entries the leaf node can hold (in general leaf nodes (except the root) must hold between $\lfloor (m+1)/2 \rfloor$ and m entries)
- Internal nodes usually store more entries than leaf nodes

Why $\lfloor (n+1)/2 \rfloor$ and not $\lfloor n/2 \rfloor + 1$??

- If n is even: Assume n=10:
  - $\lfloor (n+1)/2 \rfloor$: then the number of pointers is between 5 and 10; in the case of underflow without borrowing, 4 pointers have to be merged with 5 pointer yielding a node with 9 pointers
  - $\lfloor n/2 \rfloor+1$: then it is between 6 and 10; in the case of underflow without borrowing, 5 pointers have to be merged with 6 pointer yielding 11 pointers which is one too many
- If n is odd: Assume n=11; then in both cases the number of pointers is between 6 and 11; in the case of an underflow without borrowing a 5 pointer node has to be merged with a 6 pointer node yielding an 11 pointer node
- Conclusion: We infer from the discussion that the minimum vs. maximum numbers of entries for a tree
  - of height 3 is: $2*\lceil (n+1)/2) \rceil * \lceil (n+1)/2) \rceil * \lfloor (m+1)/2 \rfloor$ vs. $n*n*n*m$
  - of height h+1 is: $2*(\lceil (n+1)/2) \rceil)^{h} *(\lfloor (m+1)/2) \rfloor$ vs. $n^{h+1}*m$

Choosing Order n+1 and Leaf Entry Maximum m

- Idea: One B+ tree node is stored in one block; choose maximal m and n+1 without exceeding block size!!
- Example 1: Want to store tuples of a relation `E(ssn,name,salary)` in a B+ tree using ssn as the search key (alternative ❶); ssn and salary take 4 bytes; name takes 12 bytes. B+ pointers take 2 bytes; the block size is 2.048 bytes and the available space inside a block for B+tree entries is 2.000 bytes. Choose n and m:
    - $(n+1)*2 + n*4 \leq 2.000 \rightarrow n \leq 1998/6=333$
    - $m \leq 2000/(4+12+4) = 2000/20=100$
    - Answer: Choose n=333 and m=100



B+ tree Block Meta Data:
Neighbor pointers, #entries,
Parent pointer, sibling bits,…

Choosing Order n+1 and Leaf Entry Maximum m

- Example 2: Want to store an index for a relation `E(ssn,name,salary)` in a B+ tree using `ssn` as the search key; storing `ssn`'s takes 4 bytes; index pointers take 4 bytes; the block size is 2.048 bytes and the available space inside the block for B+ tree entries is 2.000 bytes. Choose n and m:
  - $(n+1)*4 + n*4 \leq 2.000 \rightarrow n \leq 1996/8=249$
  - $m \leq 2.000/(4 + 4) = 2.000/8 = 250$
  - Answer: Choose n=249 and m=250

Coping with Duplicate Keys in B+ Trees

- Possible Approaches:

❶ Just allow duplicate keys

Consequences:

- Search is still efficient
- Insertion is still efficient (but could create "hot spots")
- Deletion faces a lot of problems: We have to follow the leaf pointers to find the entry to be deleted, and then updating the intermediate nodes might get quite complicated (can partially be solved by creating two-way node pointers)

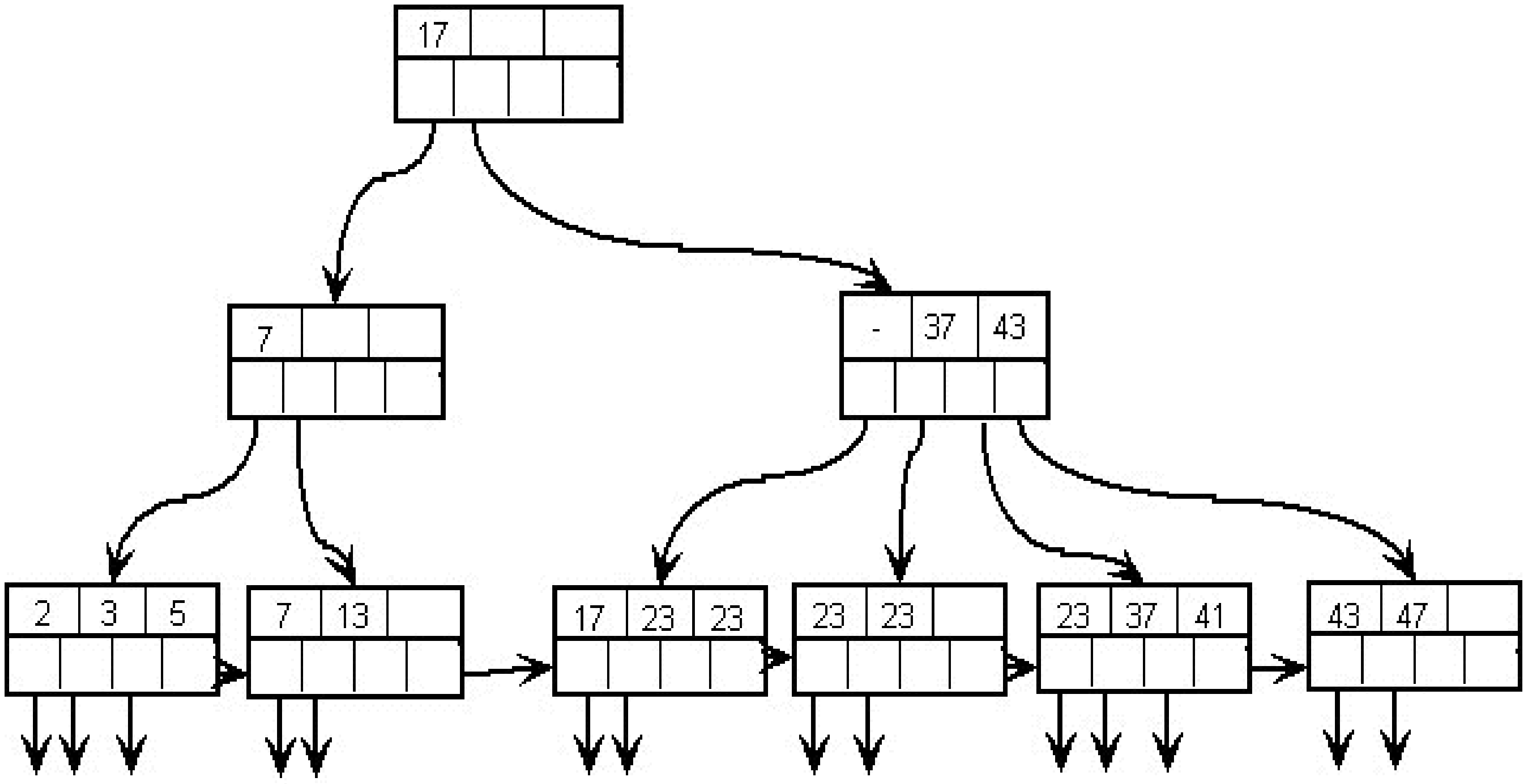
❷ Just create unique keys by using `key+data` (`key*`)

Consequences:

- Deletion is no longer a problem
- n (because of the larger key size) is significantly lower, and therefore the height of the tree is likely higher

Duplicate Keys in B+ Trees (first approach)

- In a non-leaf node, K_i is the smallest new value appearing in subtree pointed to by (i+1) pointer. If there is no new value, K_i = NULL.



Duplicate Keys in B+ Trees (first approach)

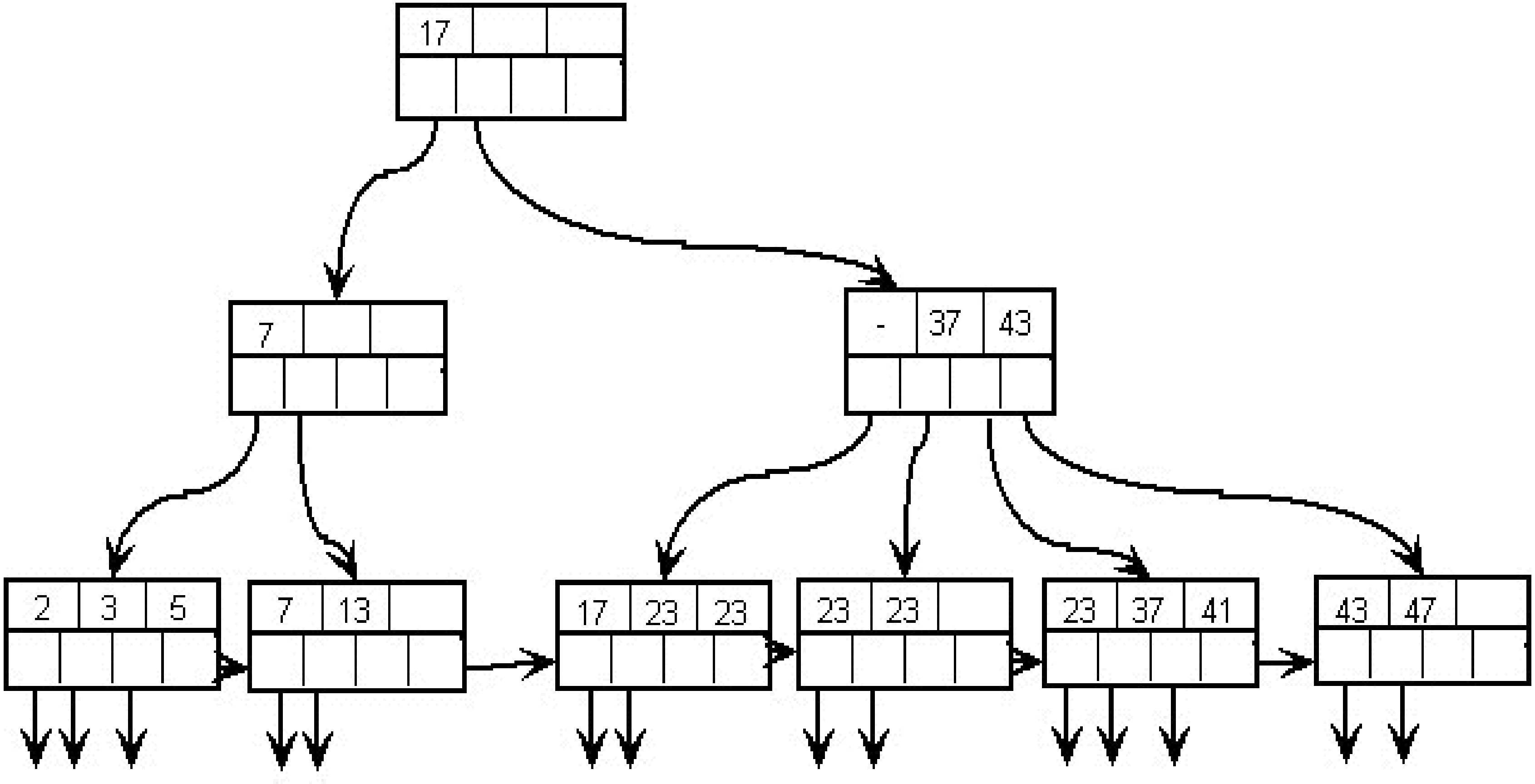
- Search algorithm the same but if you reach a leaf, you must be prepared to search the sibling if the last value of leaf equals the search value

- Insertion algorithm is the same, but attention must be kept when an old value is inserted in a leaf and we must split
  - The parent might get a `NULL` key value pointing to the second leaf of the split

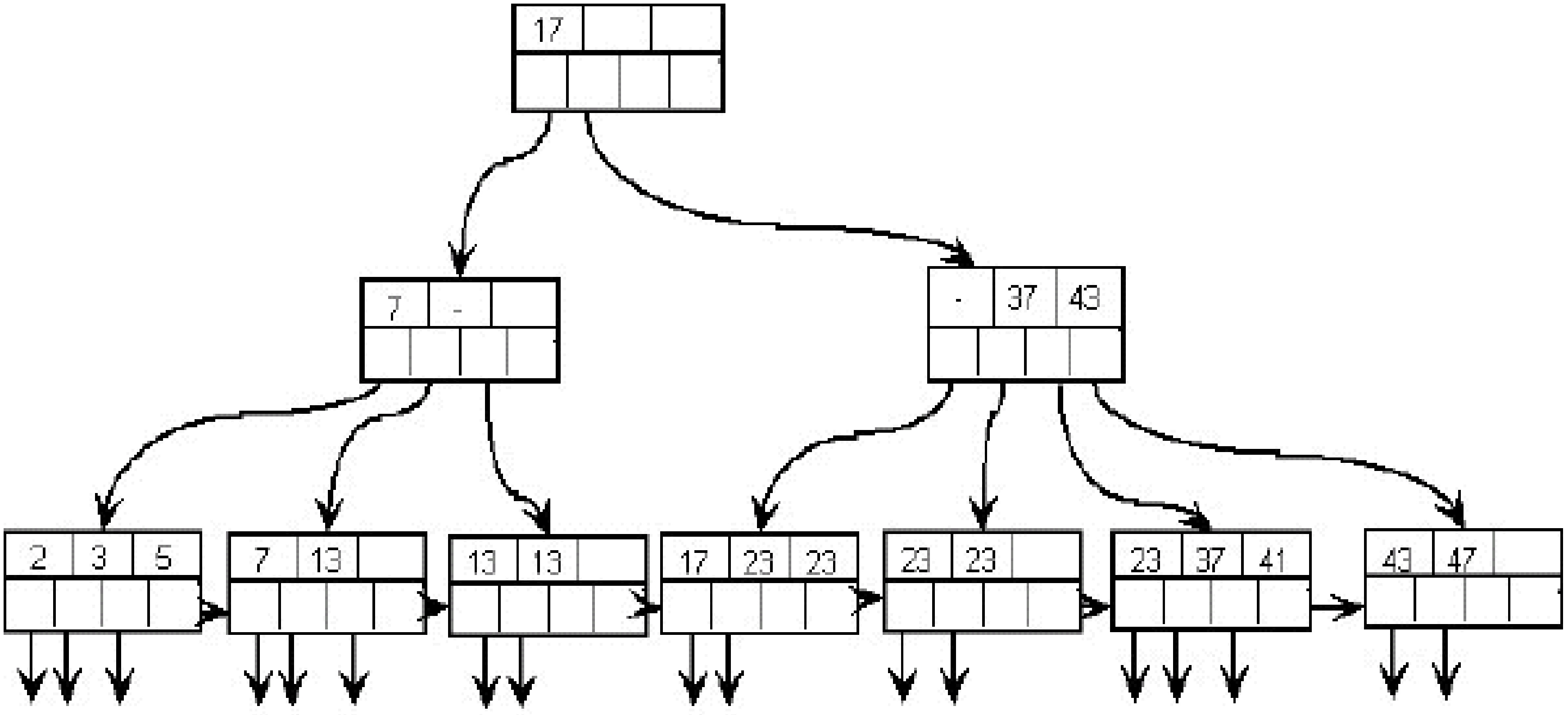
- Deletion algorithm is the same, but attention must be kept when there is underflow, because in this case the parent node must be updated

Duplicate Key Insertion in a B+ Tree

- Initial B+ Tree as below. Insertion of 13 two times.



Duplicate Key Insertion in a B+ Tree



B+ Tree Performance

- Tree levels
  - Tree Fanout
    - Size of key
    - Page utilization
- Tree maintenance
  - Online
    - On inserts
    - On deletes
  - Offline
- Tree locking
- Tree root in main memory

B+ Tree Performance

- Key length influences fanout
  - Choose small key when creating an index
  - Key compression
    - Prefix compression (Oracle 8, MySQL): only store that part of the key that is needed to distinguish it from its neighbors: Smi, Smo, Smy for Smith, Smoot, Smythe.
    - Front compression (Oracle 5): adjacent keys have their front portion factored out: Smi, (2)o, (2)y. There are problems with this approach:
      - Processor overhead for maintenance
      - Locking Smoot requires locking Smith too

Using Indexes: First Example

- `Executives(ename, title, dname, address)`
- Each attribute is 25 bytes, page size is 1.000 bytes; i.e., we store 10 records per page (record size = 100 bytes)
- Relation contains 100.000 records, 10.000 pages
- Consider query:

```
SELECT E.title, E.ename
FROM Executives E
WHERE E.title = ‘MANAGER’
```

- Assume only 10% of tuples meet the selection condition
- Selectivity S = 0,1
- Pointer size = 5 bytes; index entry size = 30 (5+25); index entries per page = 33
- Assume that there are 1.000 job titles. Index on job title contains only two levels (assuming that all index nodes are full)

Using Indexes: First Example (cont.)

- Sequential scan of files
  - Cost = File size = 10.000
- Binary search for file (sorted on title)
  - Cost = $\log_2$(File size) + # file pages containing qualifying records = $\log_2$(10.000) + 1.000= 1.013
  - Here we could also say $\log_2$(10.000) + 999, since the first page containing Manager records is retrieved by binary search
- Clustered B+ tree index on title (file sorted on title)
  - Cost = index lookup + (File size) *S = 2 + (10.000)*0.1=1.002
  - We use the index to retrieve the first record - since the file is clustered, the remaining Manager records are stored sequentially after the first one
- Clustered B+ tree index on ename
  - No use of index. File Scan

Using Indexes: First Example (cont.)

- Unclustered B+ tree index on title
  - ◆Cost = index lookup + # of leaf node blocks + # of records * S
  - ◆Cost = 2 + (100.000*0,1)/200 + 100.000*0,1 + = 2+50+10.000=10.052 (we can put 200 pointers per block i.e., 1.000/5)
- Clustered B+ tree index on <title, ename>
  - ◆Cost = index lookup + # of index entries * S (The index now has larger size, since each entry is 55 bytes)
- Clustered B+ tree index on <ename, title>
  - ◆Index only scan, Cost = # of index entries

Using Indexes: Second Example

- Assume that relation `R(A,B,C)` has the following properties:
  - R has 54.000 tuples
  - 20 tuples of R can fit in 1 block
  - Attribute A is an integer with range from 1 to 270
  - Attribute values are uniformly distributed in the relation
  - Tuples with the same attribute value of A are put in the same block and blocks with the same attribute value of A are chained by pointers
  - There is a B+ Tree index on A, which stores 10 pointers per node

- What is the minimum height of the B+ Tree? How many block accesses for the following query using B+ Tree with minimum height?

```
SELECT A
FROM r
WHERE A = 150
```

Using Indexes: Second Example (cont.)

- The height of the tree would be minimum when space of each node is fully occupied

  => Each node has 10 pointers to the next level node

  => Height = $\log_{10}(270) = 2{,}43$

  => Minimum Height is 3

- Number of data blocks = 54.000 tuples / 20 = 2.700
- Number of data blocks for each value of A = 2.700 / 270 = 10

- To implement the query, we give A=150 as the search-key value to the B+ Tree and we need to access 3 levels (3 index blocks) to find out the blocks storing tuples with A=150

- Total no. of block accesses = index block accesses + data block accesses = 3 + 10 = 13

Tree-Structured Indexing Summary

- Sequential and direct access
- Straightforward insertion and deletion maintaining ordering
- Grows as required—only as big as needs to be
- Predictable scanning pattern
- Predictable and constant search time

but .....

- maintenance overhead
- overkill for small static files
- duplicate keys?
- relies on random distribution of key values for efficient insertion

Τέλος Ενότητας

Ευρωπαϊκή Ένωση
Ευρωπαϊκό Κοινωνικό Ταμείο

ΕΠΙΧΕΙΡΗΣΙΑΚΟ ΠΡΟΓΡΑΜΜΑ
ΕΚΠΑΙΔΕΥΣΗ ΚΑΙ ΔΙΑ ΒΙΟΥ ΜΑΘΗΣΗ
επένδυση στην κοινωνία της γνώσης
ΥΠΟΥΡΓΕΙΟ ΠΑΙΔΕΙΑΣ & ΘΡΗΣΚΕΥΜΑΤΩΝ, ΠΟΛΙΤΙΣΜΟΥ & ΑΘΛΗΤΙΣΜΟΥ
ΕΙΔΙΚΗ ΥΠΗΡΕΣΙΑ ΔΙΑΧΕΙΡΙΣΗΣ
Με τη συγχρηματοδότηση της Ελλάδας και της Ευρωπαϊκής Ένωσης

ΕΣΠΑ
2007-2013
πρόγραμμα για την ανάπτυξη
ΕΥΡΩΠΑΪΚΟ ΚΟΙΝΩΝΙΚΟ ΤΑΜΕΙΟ

Χρηματοδότηση

- Το παρόν εκπαιδευτικό υλικό έχει αναπτυχθεί στα πλαίσια του εκπαιδευτικού έργου του διδάσκοντα.
- Το έργο «**Ανοικτά Ακαδημαϊκά Μαθήματα στο Πανεπιστήμιο Κρήτης**» έχει χρηματοδοτήσει μόνο τη αναδιαμόρφωση του εκπαιδευτικού υλικού.
- Το έργο υλοποιείται στο πλαίσιο του Επιχειρησιακού Προγράμματος «Εκπαίδευση και Δια Βίου Μάθηση» και συγχρηματοδοτείται από την Ευρωπαϊκή Ένωση (Ευρωπαϊκό Κοινωνικό Ταμείο) και από εθνικούς πόρους.

Ευρωπαϊκή Ένωση
Ευρωπαϊκό Κοινωνικό Ταμείο

ΕΠΙΧΕΙΡΗΣΙΑΚΟ ΠΡΟΓΡΑΜΜΑ
ΕΚΠΑΙΔΕΥΣΗ ΚΑΙ ΔΙΑ ΒΙΟΥ ΜΑΘΗΣΗ
επένδυση στην κοινωνία της γνώσης
ΥΠΟΥΡΓΕΙΟ ΠΑΙΔΕΙΑΣ ΚΑΙ ΘΡΗΣΚΕΥΜΑΤΩΝ
ΕΙΔΙΚΗ ΥΠΗΡΕΣΙΑ ΔΙΑΧΕΙΡΙΣΗΣ

ΕΣΠΑ
2007-2013
πρόγραμμα για την ανάπτυξη
ΕΥΡΩΠΑΪΚΟ ΚΟΙΝΩΝΙΚΟ ΤΑΜΕΙΟ

Με τη συγχρηματοδότηση της Ελλάδας και της Ευρωπαϊκής Ένωσης

Σημειώματα

Σημειώματα

Σημείωμα αδειοδότησης

Σημείωμα Αναφοράς

Copyright Πανεπιστήμιο Κρήτης, Δημήτρης Πλεξουσάκης. «**Συστήματα Διαχείρισης Βάσεων Δεδομένων. Φροντιστήριο 3: Tutorial on Indexing part 1 – Properties and Tree-Structured Indexing**». Έκδοση: 1.0. Ηράκλειο/Ρέθυμνο 2015. Διαθέσιμο από τη δικτυακή διεύθυνση: http://www.csd.uoc.gr/~hy460/