CS 540
Database Management Systems

Storage& indexing

Database System Implementation

```
┌─────────────────────┐
│  User Requirements  │
└─────────────────────┘
          │
          ▼
┌─────────────────────┐        ┌─────────────┐        ┌──────────────┐
│     Conceptual      │ ─────► │   Schema    │ ─────► │   Physical   │
│       Design        │        │             │        │   Storage    │
└─────────────────────┘        └─────────────┘        └──────────────┘
          ▲                           ▲                       ▲
          ┊                           ┊                       ┊
      Entity                   Relational Model        Files and Indexes
  Relationship(ER)
       Model
```
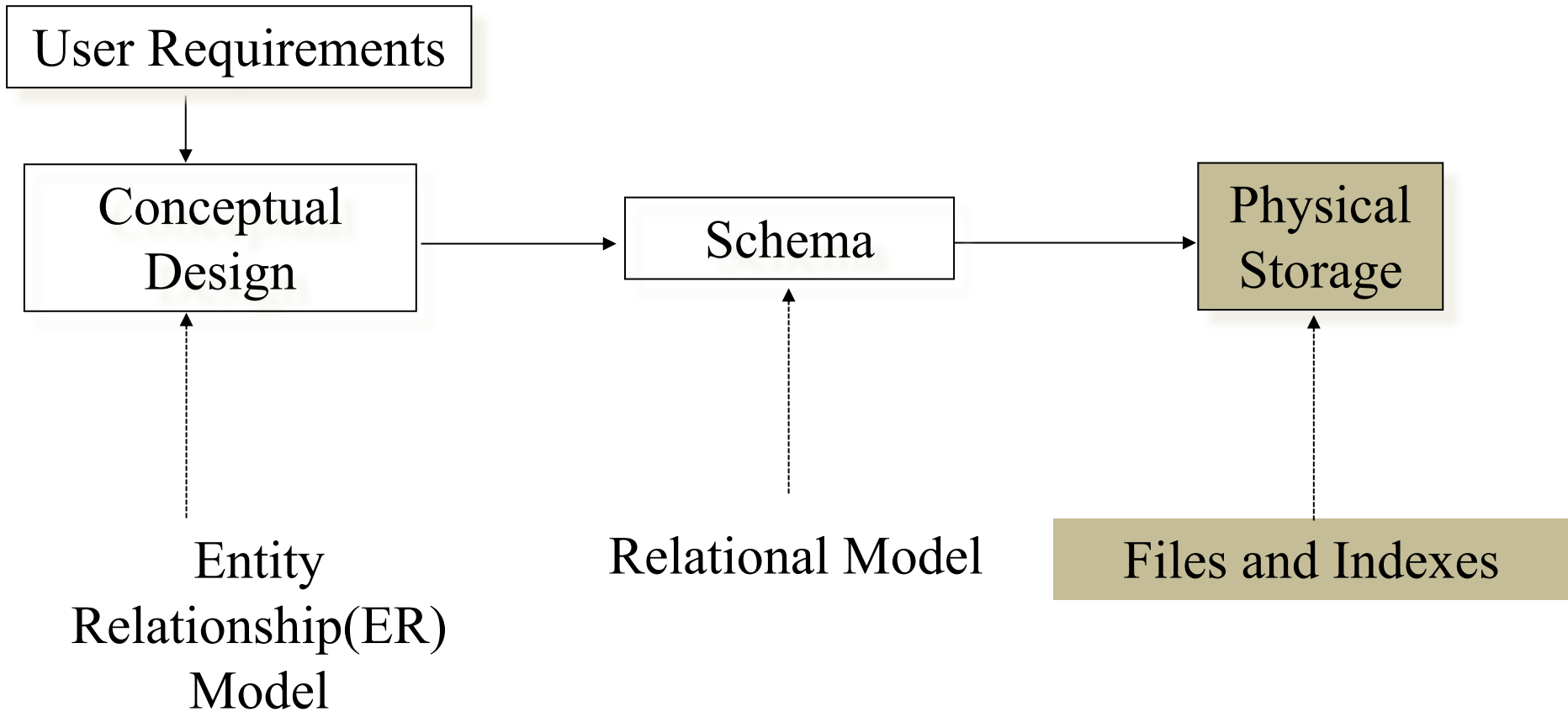
The advantage of data systems

- It separates logical level (schema) from physical level (implementation).

- Physical data independence
  - Users do not worry about data storage and processing.
  - It is all SQL!
  - Most queries work over all relational data systems.

DBMS Architecture

User/Web Forms/Applications/DBA

query   transaction

| Query Parser | Transaction Manager | Process manager |

Query Rewriter

| Query Optimizer | Lock Manager | Logging & Recovery |

Query Executor

Files & Access Methods

Buffer Manager

Storage Manager

Buffers   Lock Tables

Main Memory

Storage

Challenges in physical level

- Processor:
  - 10000 – 100000 MIPS
- Main memory (RAM)
  - around 10 Gb/ sec.
  - expensive; volatile
- Secondary storage (HDD, SSD, …)
  - cheap; high capacity; durability
  - data is stored and retrieved in *blocks (pages)*
  - access is orders of magnitude slower than main memory

Components of a disk

- The platters spin (say, 90rps).

- The arm assembly is moved in or out to position a head on a desired track. Tracks under heads make a *cylinder* (imaginary!).

- Only one head reads/writes at any one time.

- *Block size* is a multiple of *sector size* (which is fixed).

Spindle

Tracks

Disk head

Sector

Arm movement

Platters

Arm assembly
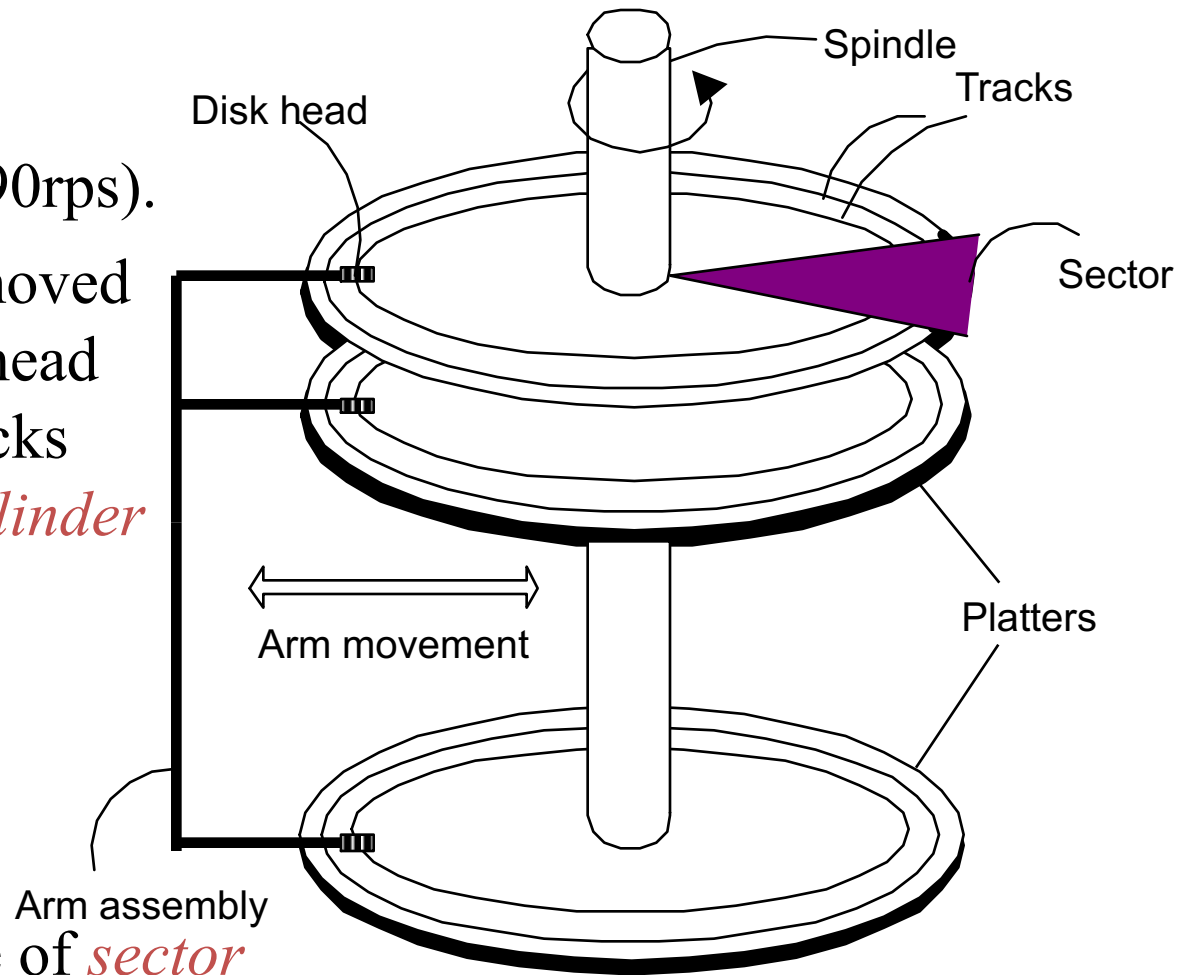
Slow access in secondary storage

- seek time + rotational latency + transfer time
  - seek time: about 4 ms - 15 ms!
  - rotational latency: about 2 ms – 7 ms!
  - transfer time: about 1000 Mb/ sec
  - seek time and rotational delay dominate.

- #I/O access dominates the time of Big data analysis and processing!

General approach to storing Big data

- Storage hierarchy:
  - main memory (RAM) for currently used data.
  - secondary storage (disk) for the main database.

- Key to lower I/O cost:
  - reduce seek/rotation delays!

- Recently main memory is getting cheaper
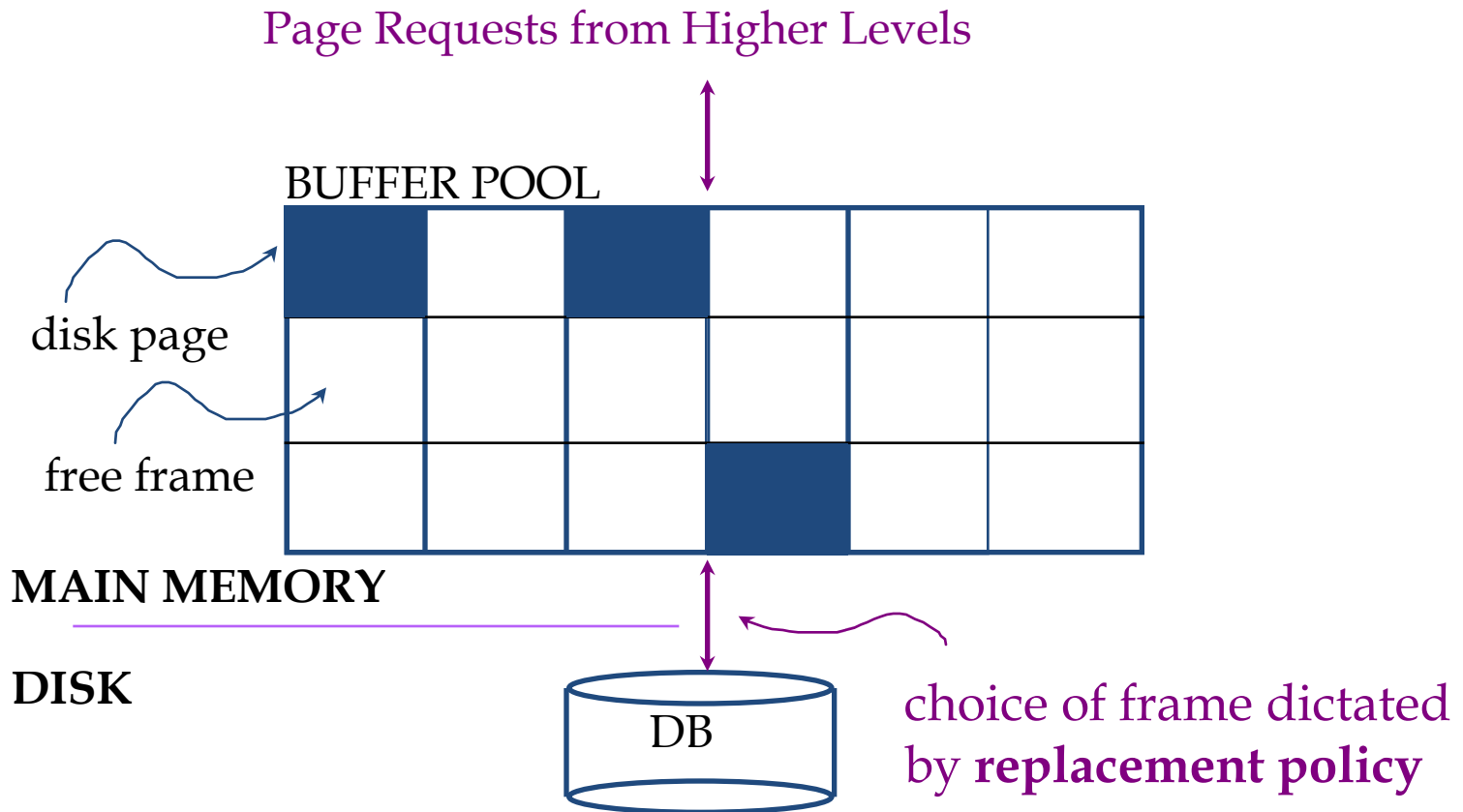 - in-memory data systems
  - different storage hierarchy?

Random versus sequential access

- Disk random access (example ?)
  - seek time + rotational latency + transfer time.
- Disk sequential access (example ?)
  - `*next*` block concept:
    - blocks on same track, followed by
    - blocks on same cylinder, followed by
    - blocks on adjacent cylinder
  - blocks in a file should be arranged sequentially on disk (by `next'), to minimize seek and rotational delay.
- For a sequential scan, *pre-fetching* several pages at a time is a big win!

Storage management

- Lowest layer of DBMS software manages space on disk.

- Higher levels call upon this layer to:
  - allocate/de-allocate a page
  - read/write a page

- Request for a *sequence* of pages must be satisfied by allocating the pages sequentially on disk!
  - higher levels don't need to know how this is done, or how free space is managed.

Buffer management

Page Requests from Higher Levels

BUFFER POOL

disk page

free frame

MAIN MEMORY

DISK

DB

choice of frame dictated by **replacement policy**

- *Data must be in RAM for DBMS to operate on it!*
- *Table of <frame#, pageid> pairs is maintained.*

When a page is requested ...

- If requested page is not in pool:
  - Choose a frame for *replacement*
  - If frame is dirty, write it to disk
  - Read requested page into chosen frame
- *Pin* the page and return its address.

☞ *If requests can be predicted (e.g., sequential scans) pages can be <u>pre-fetched</u> several pages at a time!*

More on buffer management

- Requestor of page must unpin it, and indicate whether page has been modified:
    - *dirty* bit is used for this.
- Page in pool may be requested many times,
    - a *pin count* is used. A page is a candidate for replacement iff *pin count* = 0.
- CC & recovery may entail additional I/O when a frame is chosen for replacement. (*Write-Ahead Log* protocol; more later.)

Buffer replacement policy

- Frame is chosen for replacement by a *replacement policy:*
  - Least-recently-used (LRU), Clock, MRU etc.
- Policy can have big impact on # of I/O's; depends on the *access pattern.*
- *Sequential flooding*: nasty situation caused by LRU + repeated sequential scans.
  - # buffer frames < # pages in file means each page request causes an I/O. MRU much better in this situation (but not in all situations, of course).

DBMS vs. OS File System

OS does disk space & buffer mgmt: why not let OS manage these tasks?

- Differences in OS support: portability issues
- Buffer management in DBMS requires ability to:
  - pin a page in buffer pool, force a page to disk (important for implementing CC & recovery),
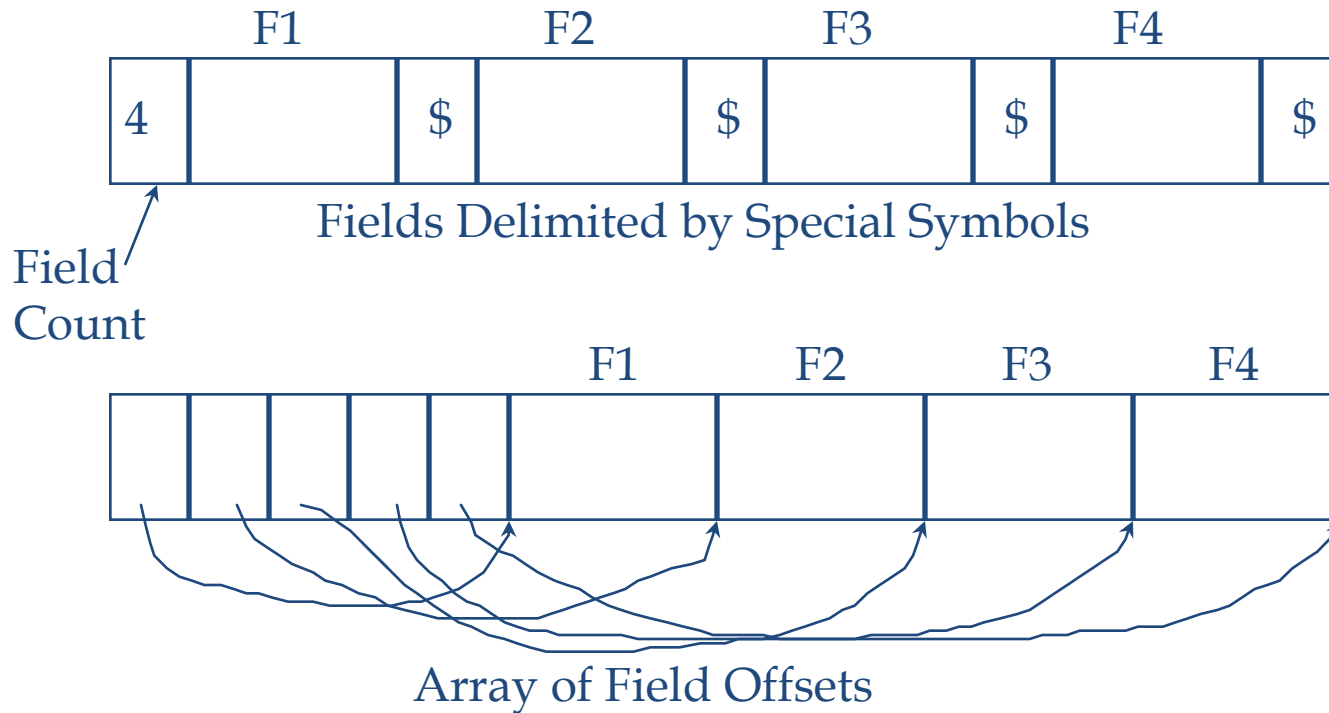 - adjust *replacement policy,* and pre-fetch pages based on access patterns in typical DB operations.

Record formats: fixed length

F1	F2	F3	F4
←— L1 —→	L2	L3	L4

Base address (B)   Address = B+L1+L2
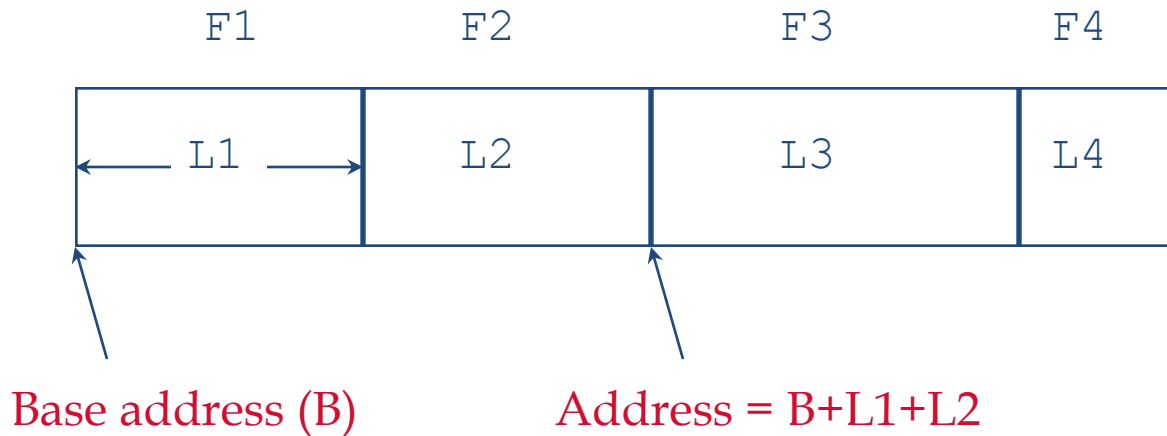
- Information about field types same for all records in a file; stored in *system catalogs.*
- Finding *i'th* field does not require scan of record.

Record formats: variable length
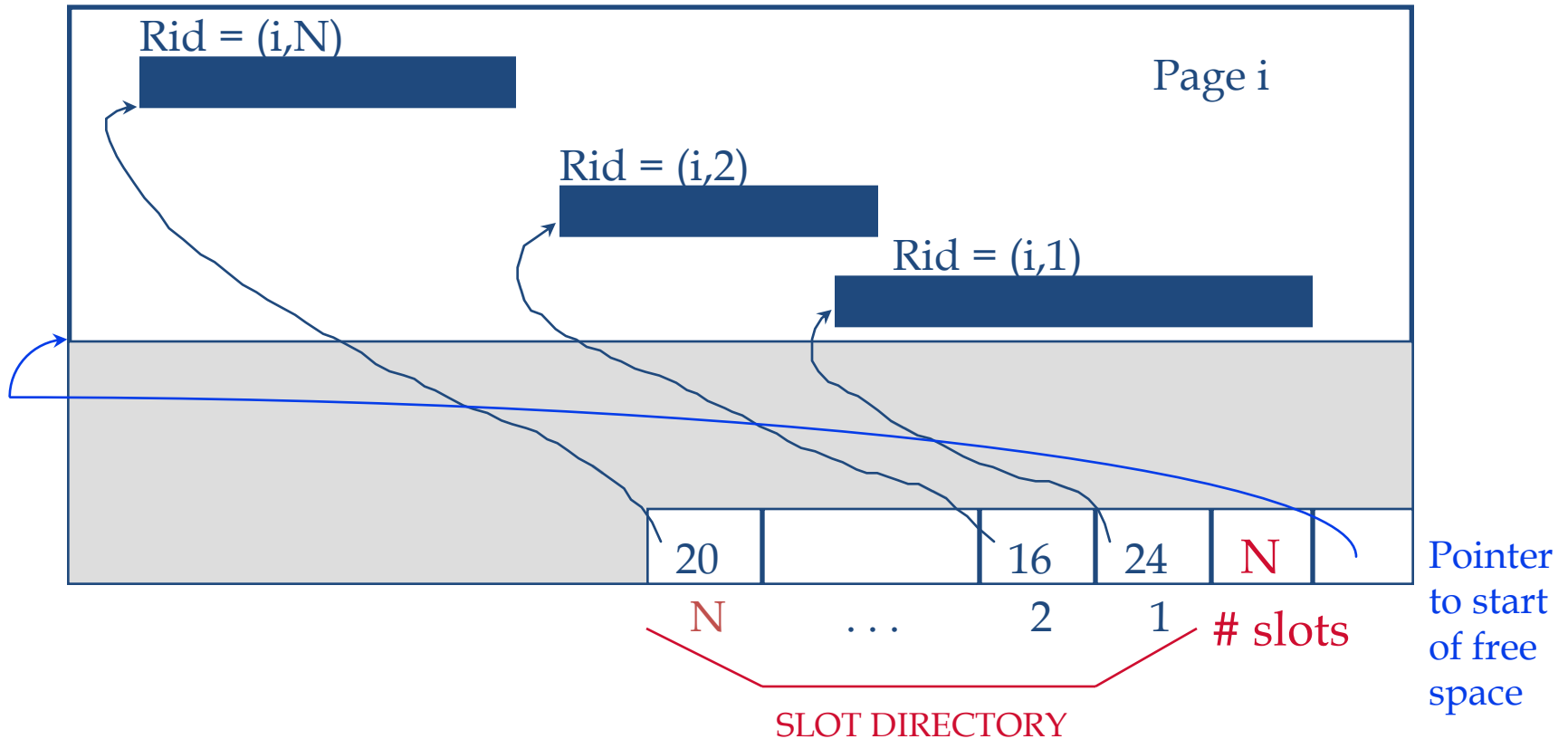
- Two alternative formats (# fields is fixed):

	F1		F2		F3		F4	
4		$		$		$		$

Fields Delimited by Special Symbols

Field
Count

						F1	F2	F3	F4

Array of Field Offsets

☛ Second offers direct access to i'th field, efficient storage
of *nulls* (special *don't know* value); small directory overhead.

Page formats: fixed length records

Slot 1	Slot 1
Slot 2	Slot 2
. . .	Free Space
Slot N	. . .
	Slot N
N	Slot M
	1 . . . 0 1 1 M

Free Space

number of records

M ... 3 2 1

number of slots

PACKED

UNPACKED, BITMAP
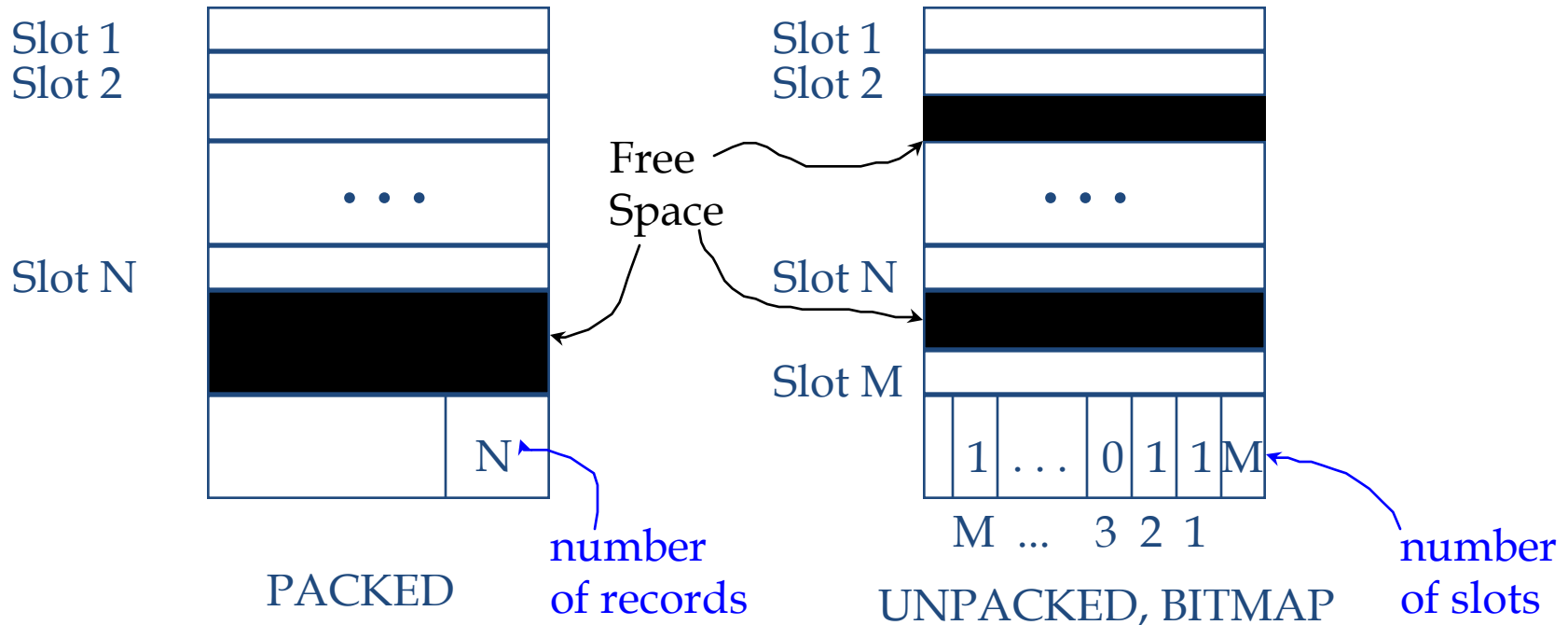
☛ *Record id* = *<page id, slot #>. In first alternative, moving records for free space management changes rid; may not be acceptable.*

Page formats: variable length records

Rid = (i,N)

Page i

Rid = (i,2)

Rid = (i,1)

20		16	24	N	
N	...	2	1	# slots	

Pointer to start of free space

SLOT DIRECTORY

☞*Can move records on page without changing rid; so, attractive for fixed-length records too.*

Spanned versus un-spanned

- Un-spanned
  - Each record belongs to only one block
- Spanned
  - Records may be stored across multiple blocks
  - Saves space
  - The only way to deal with large records and fields: blob, image, …

Files of records

- Page or block is OK when doing I/O, but higher levels of DBMS operate on *records*, and *files of records*.

- FILE: A collection of pages, each containing a collection of records. Must support:
  - insert/delete/modify record
  - read a particular record (specified using *record id*)
  - scan all records (possibly with some conditions on the records to be retrieved)

System catalogs

- For each relation:
  - name, file name, file structure (e.g., Heap file)
  - attribute name and type, for each attribute
  - index name, for each index
  - integrity constraints
- For each view:
  - view name and definition
- Plus statistics, authorization, buffer pool size, etc.

☛ *Catalogs are themselves stored as relations*!

Access paths

- The methods that RDBMS uses to retrieve data.
- Attribute value(s) ➜ Tuple(s)
- Point query over *Coffee(<u>cname</u>, producer)*

  Select *

  From coffee

  Where cname = 'Costa';

- Range query over *Sells(<u>sname</u>, <u>cname</u>, price)*

  Select *

  From Sells

  Where price > 2 AND price < 10;

Types of access paths

- Heap files
  - there is not any order in the file
  - new blocks are inserted at the end of the file.
- Sorted files
  - order blocks (and records) based on some key.
  - physically contiguous or using links
- Average cost of heap versus sorted files
  - search?
  - insertion/update?
  - deletion?
- Middle ground?

Indexing

- An old idea

Index

- A data structure that speeds up selecting tuples in a relation based on some **search keys**.

- Search key
  - A subset of the attributes in a relation
  - May not be the same as the (primary) key

- Entries in an index
  - (k, r)
  - k is the search key.
  - r is the pointer to a record (record id).

Index

- **Data file** stores the table data.
- **Index file** stores the index data structure.

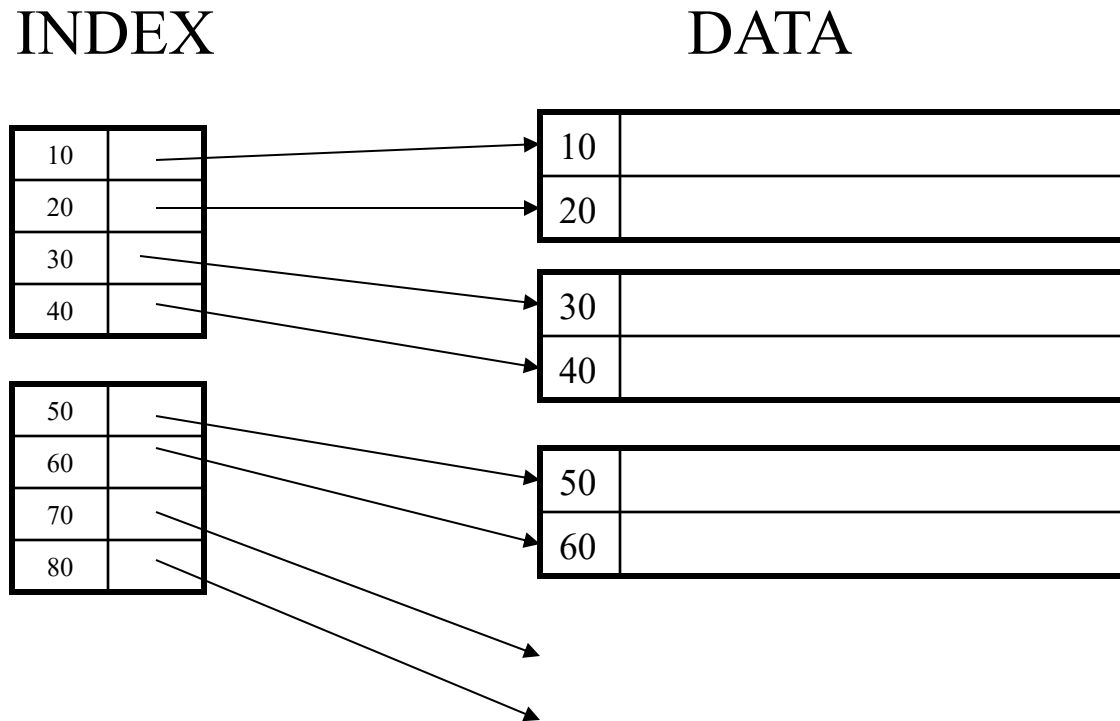Index File   Data File
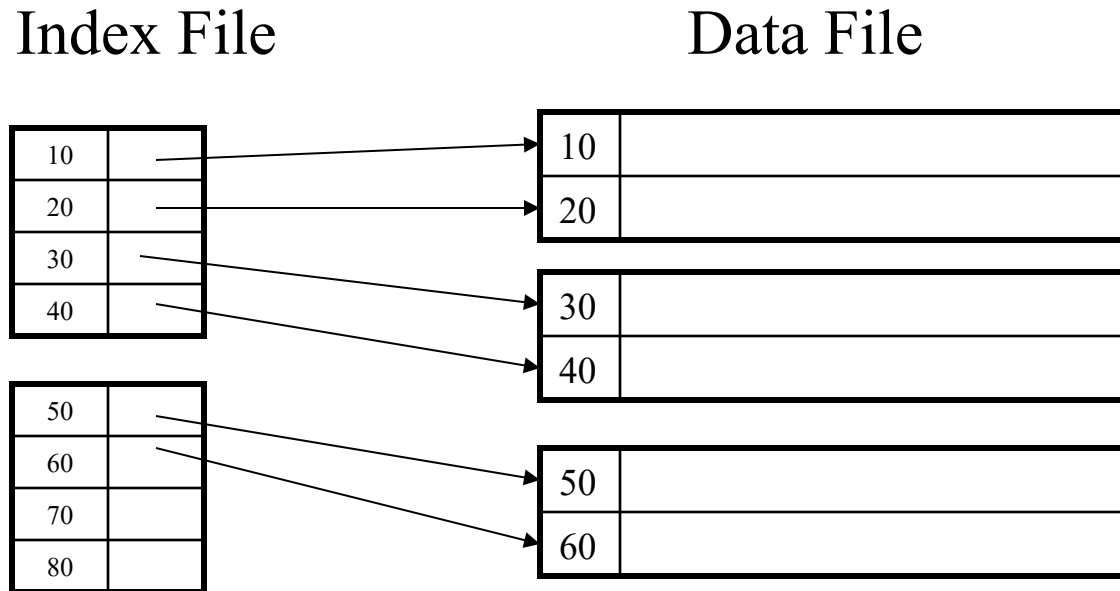
- Index file is smaller than the data file.
- Ideally, the index should fit in the main memory.

Index categorizations

- Clustered vs. unclustered
  - Records are stored according to the index order.
  - Records are stored in another order, or not any order.

- Dense vs. sparse
  - Each record is pointed by an entry in the index.
  - Each block has an entry in the index.
  - Size versus time tradeoff.

- Primary vs. secondary
  - Primary key is the search key
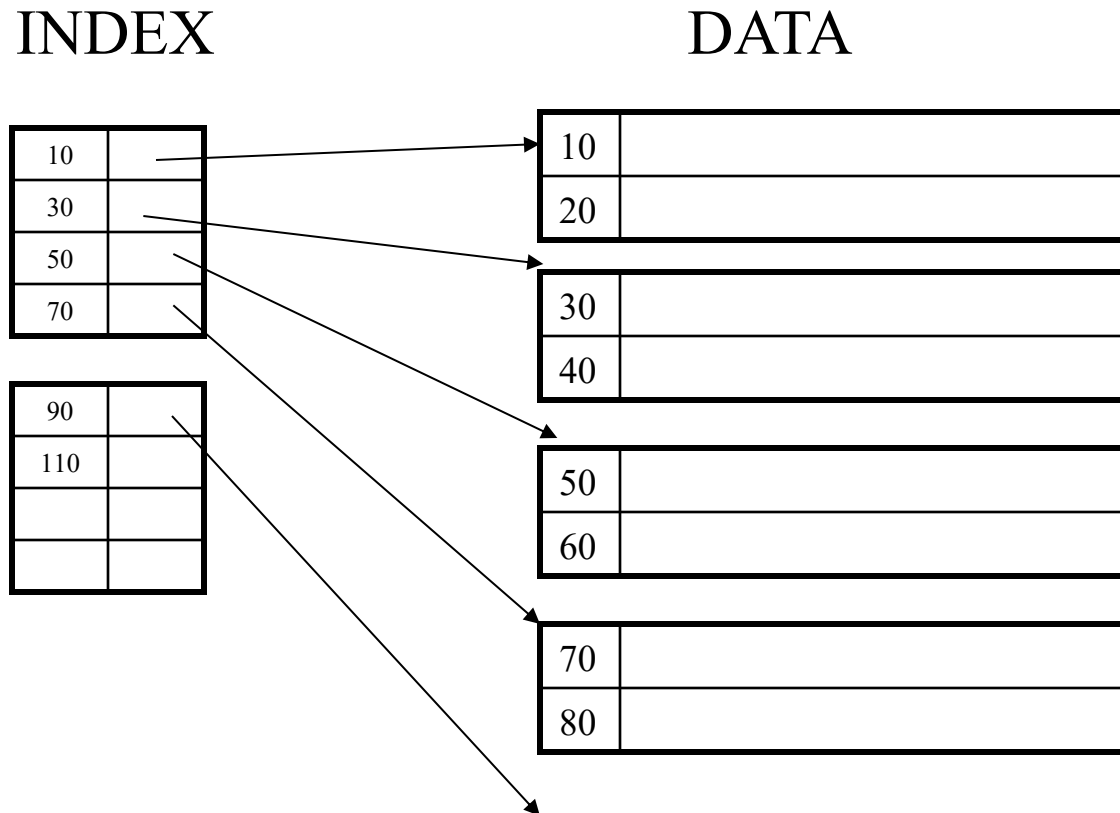 - Other attributes.

Index categorizations

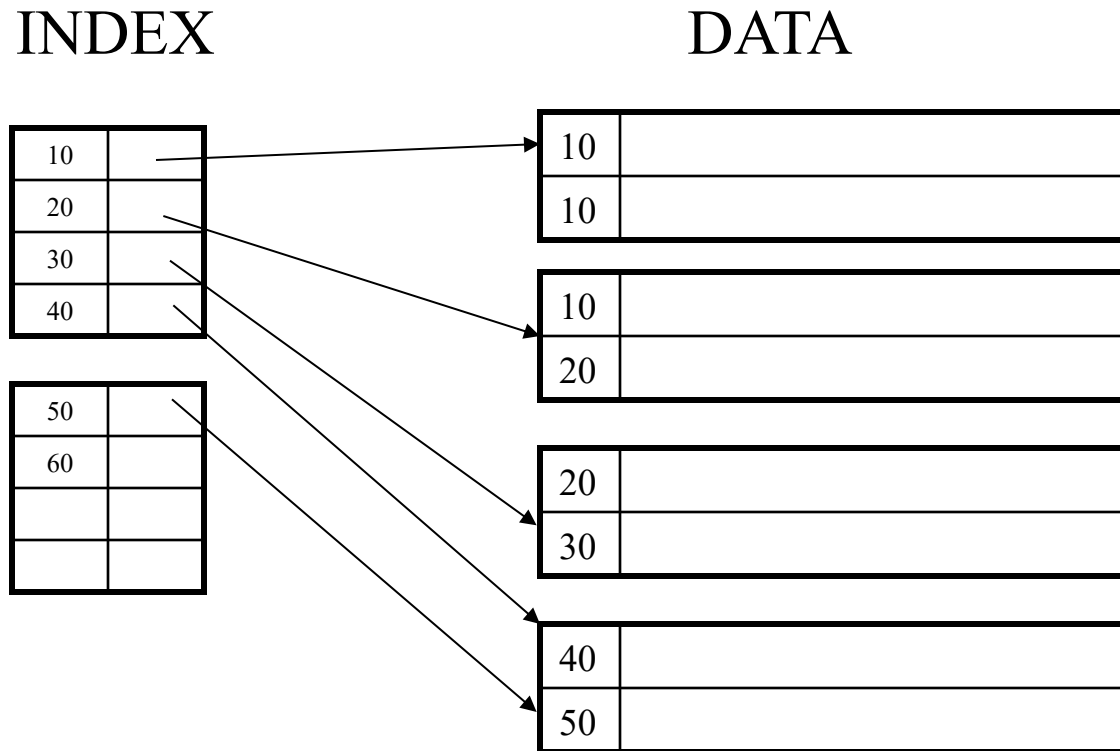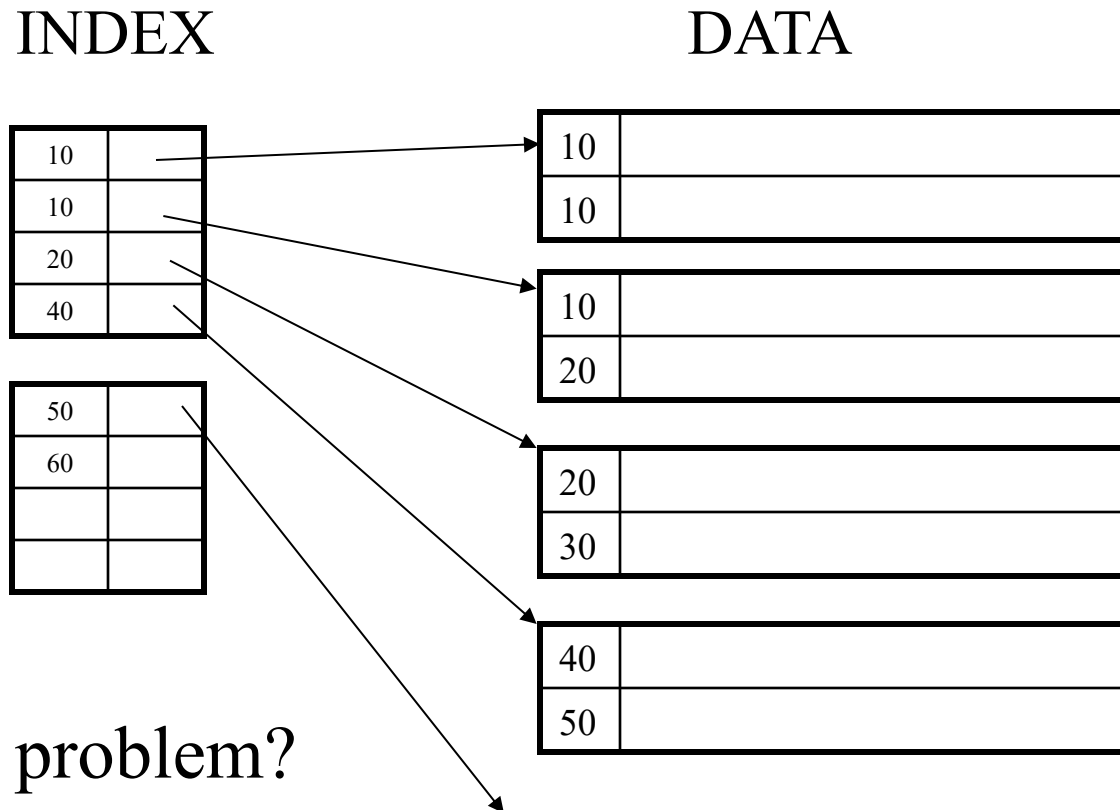- Clustered and dense

INDEX   DATA

10			10	
20			20	

30			30	
40			40	

50			50	
60			60	
70				
80				

Index categorizations

- Clustered and sparse

INDEX   DATA

10	
30	
50	
70	

90	
110	

10	
20	

30	
40	

50	
60	

70	
80	

Duplicate search keys

- Clustered and dense

INDEX   DATA

10	
20	
30	
40	

| 10 |
| 10 |

| 10 |
| 20 |

50	
60	

| 20 |
| 30 |

| 40 |
| 50 |

Duplicate search keys

- Clustered and sparse:

INDEX   DATA

10	
10	
20	
40	

50	
60	

10	
10	

10	
20	

20	
30	

40	
50	

  – Any problem?

Duplicate search keys

- Clustered and sparse:
  - Point to the lowest new search key in every block

INDEX   DATA

10	
20	
30	
40	

50	

10	
10	

10	
20	

20	
30	

40	
50	
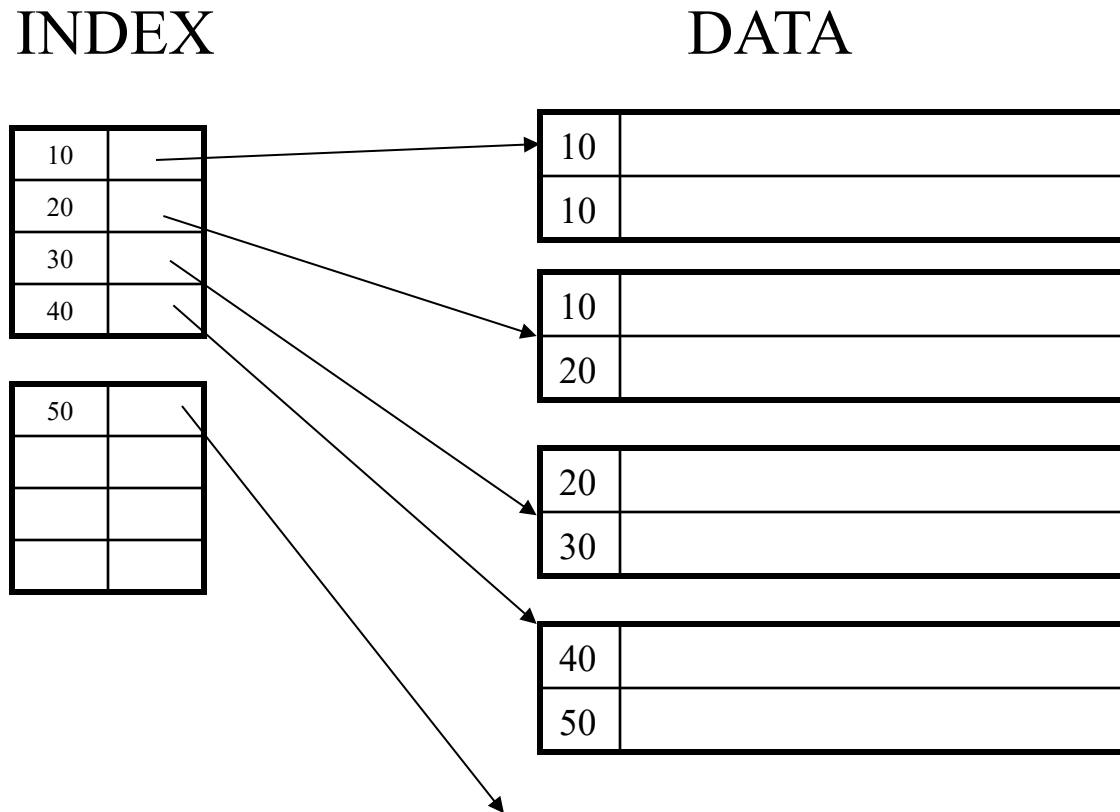
Unclustered Index

- Dense / sparse?

INDEX   DATA

10	
10	
10	
20	

| 30 | |
| 10 | |

| 20 | |
| 30 | |

20	
30	
30	
40	

| 10 | |
| 20 | |

| 10 | |
| 40 | |
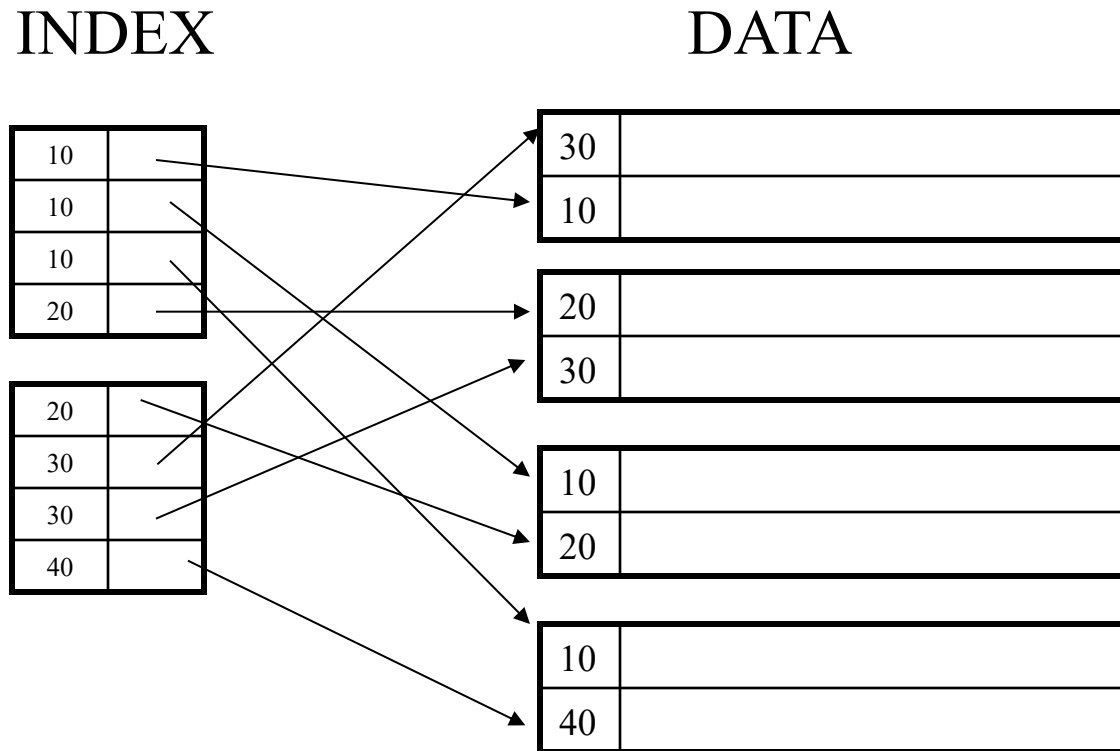
Index structures

- Tree indexing
  - extends the idea of search trees in main memory
  - popular
  - most frequently used one is B+ tree
- Hash indexing
  - extends the idea of hash tables in main memory
  - less frequently used
- Other structures based on the type of data/ query
  - bitmap indexes, learned indexes, …

B+ trees

- The index of a very large data file gets too large.
- How about building an *index* for the index file?
  - A multi-level index, or a tree
- Insert/delete at $\log_F N$ cost; keep tree *height-balanced.* (F = fanout, N = # leaf pages)
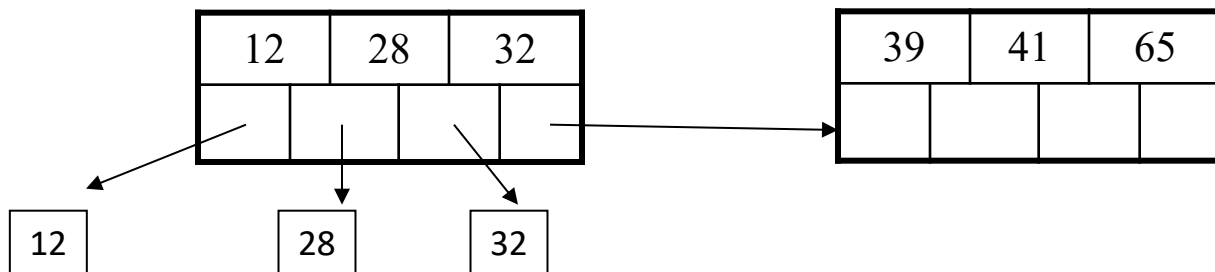- Supports point (equality) and range queries efficiently.

B+ trees

- Degree (order) of the tree: d
- Each node (except root) stores [d, 2d] keys
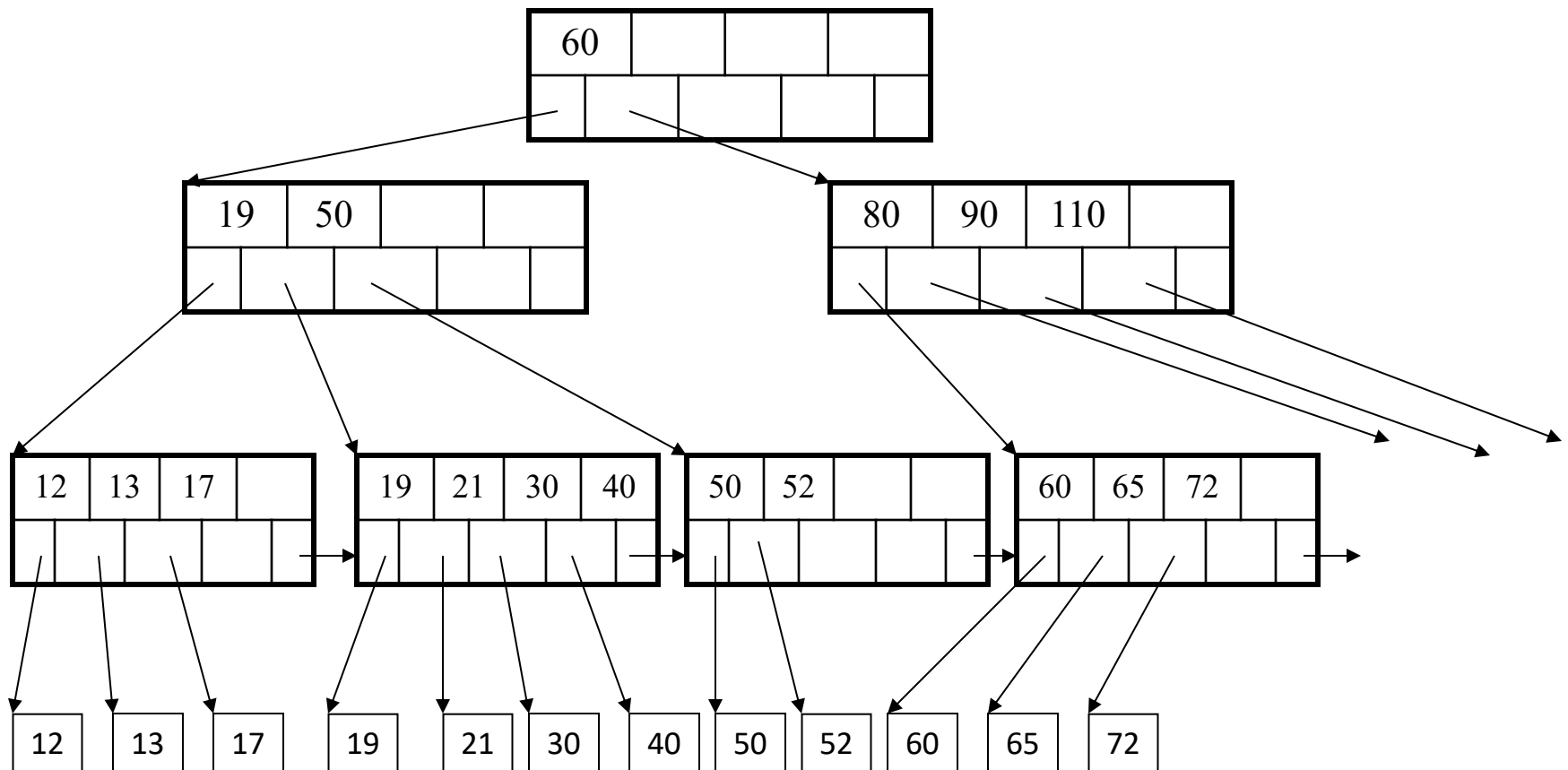 - minimum 50% occupancy (except for root).

Non-leaf nodes

10	32	94

[A , 10)   [10, 32)   [32, 94)   [94, B)

Leaf nodes

12	28	32

39	41	65

Records

| 12 | | 28 | | 32 |
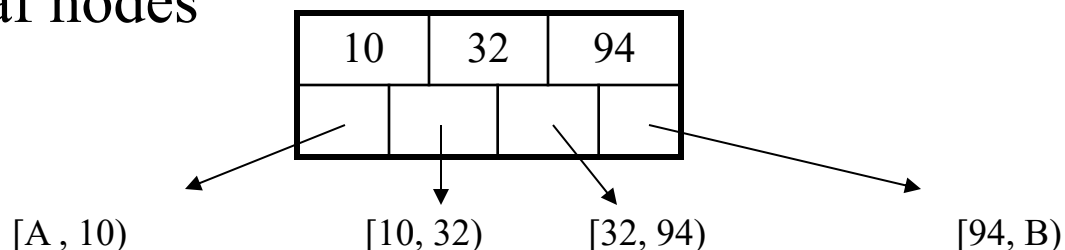
Example

d = 2

B+ tree tuning

- How to choose the value of d?
  - each node should fit in a block.

- Example
  - key value: 8 byte; record pointer: 16 bytes
  - block size: 4096 bytes
  - $2d * 8 + (2d + 1) * 16 <= 4096; d <= 85$

B+ trees in practice

- Typical order: 100. Typical fill-factor: 67%.
  - average fanout = 133
- Typical capacities:
  - Height 4: $133^4 = 312,900,700$ records
  - Height 3: $133^3 = \quad 2,352,637$ records
- Can often hold top levels in buffer pool:
  - Level 1 = $\qquad$ 1 page = $\quad$ 8 Kbytes
  - Level 2 = $\quad$ 133 pages = $\quad$ 1 Mbyte
  - Level 3 = 17,689 pages = 133 MBytes

Retrieving tuples

- Point queries
  - start from the root and follow the links to the leaf.

- Range queries
  - find the lowest point in the range; then, follow the links between the nodes.
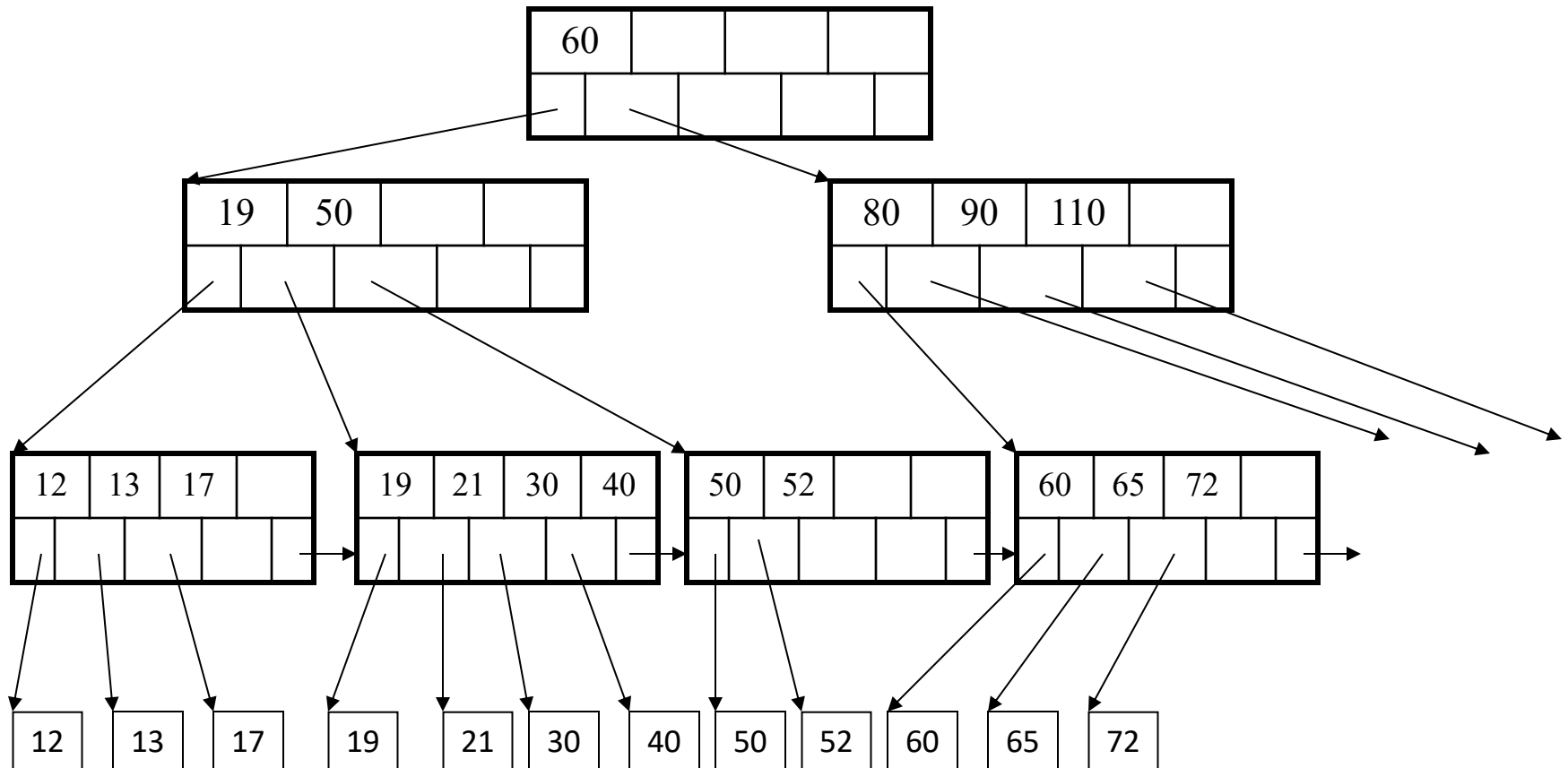
Inserting a new key

- Pick the proper leaf node and insert the key.
- If the node contains more than 2d keys, split the node and insert the extra node in the parent.

(K3,) parent

K1	K2	K3	K4	K5
R0	R1	R2	R3	R4	R5

→

K1	K2		
R0	R1	R2		

K4	K5		
R3	R4	R5		
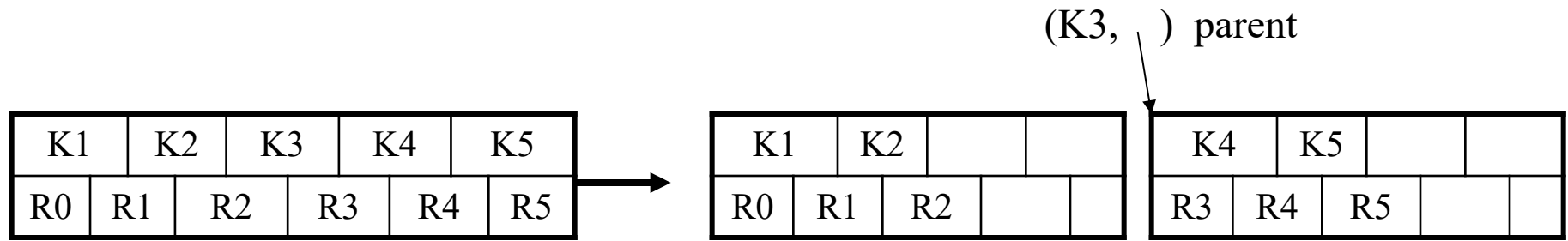
  – If leaf level, add K3 to the right node

Insertion
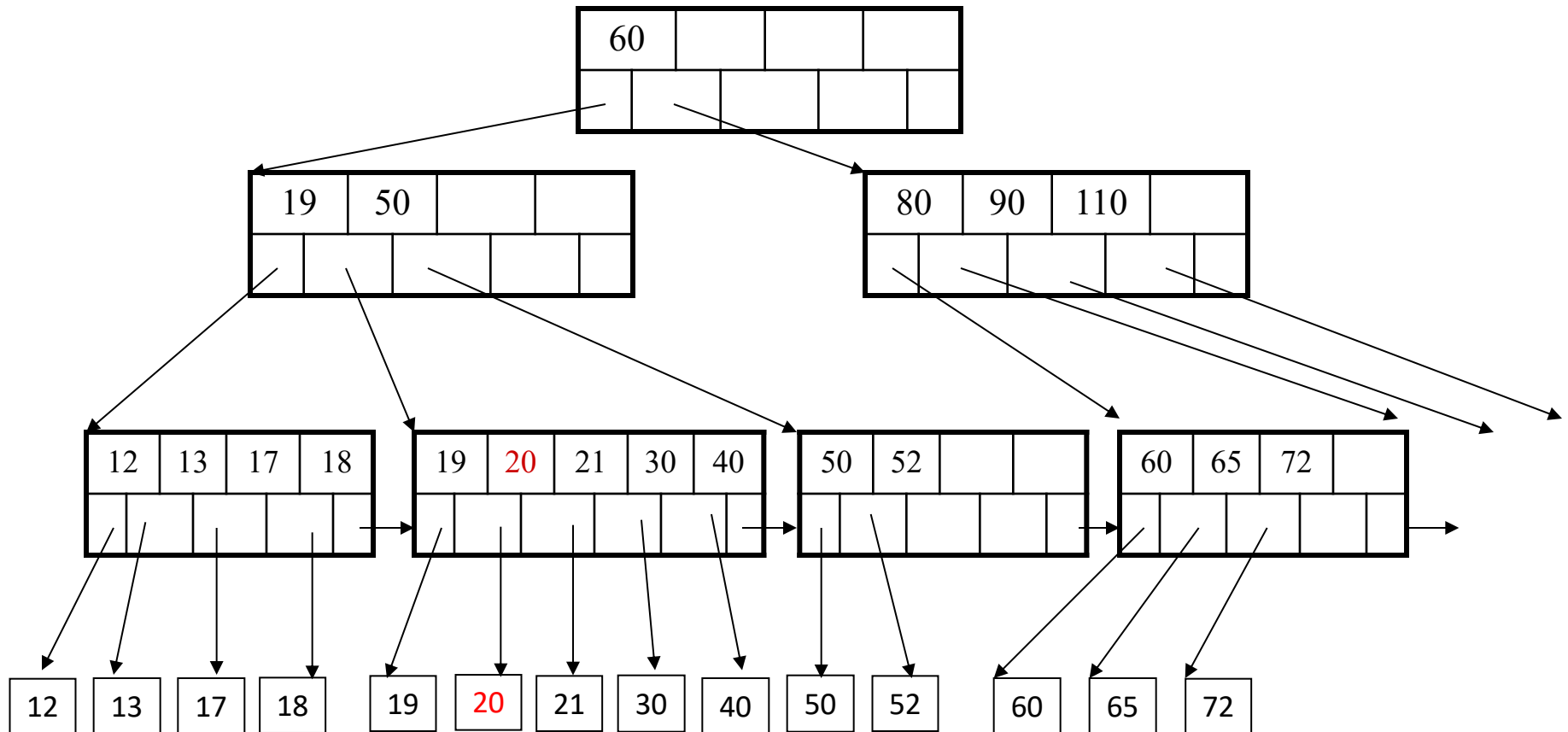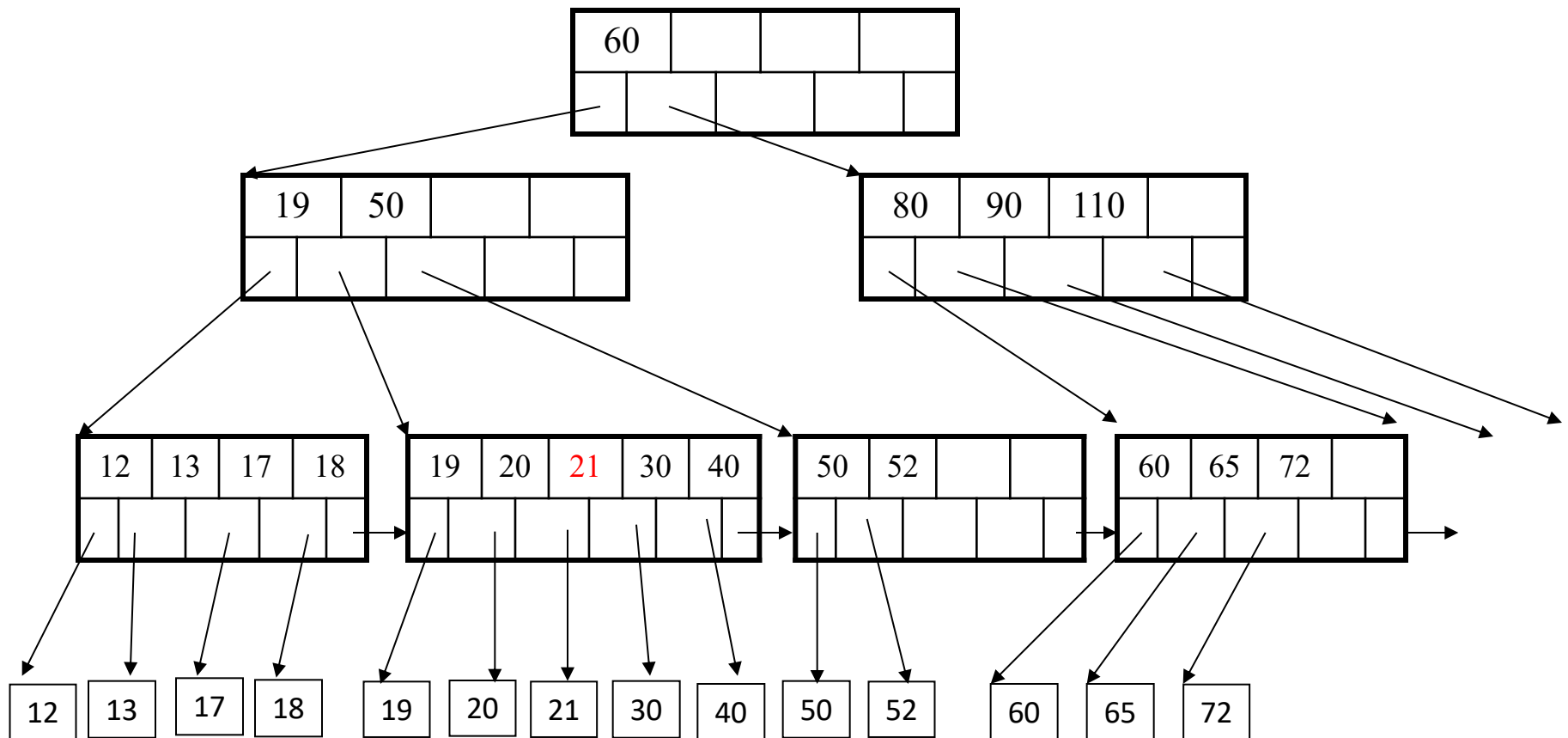
Insert K = 18

Insertion
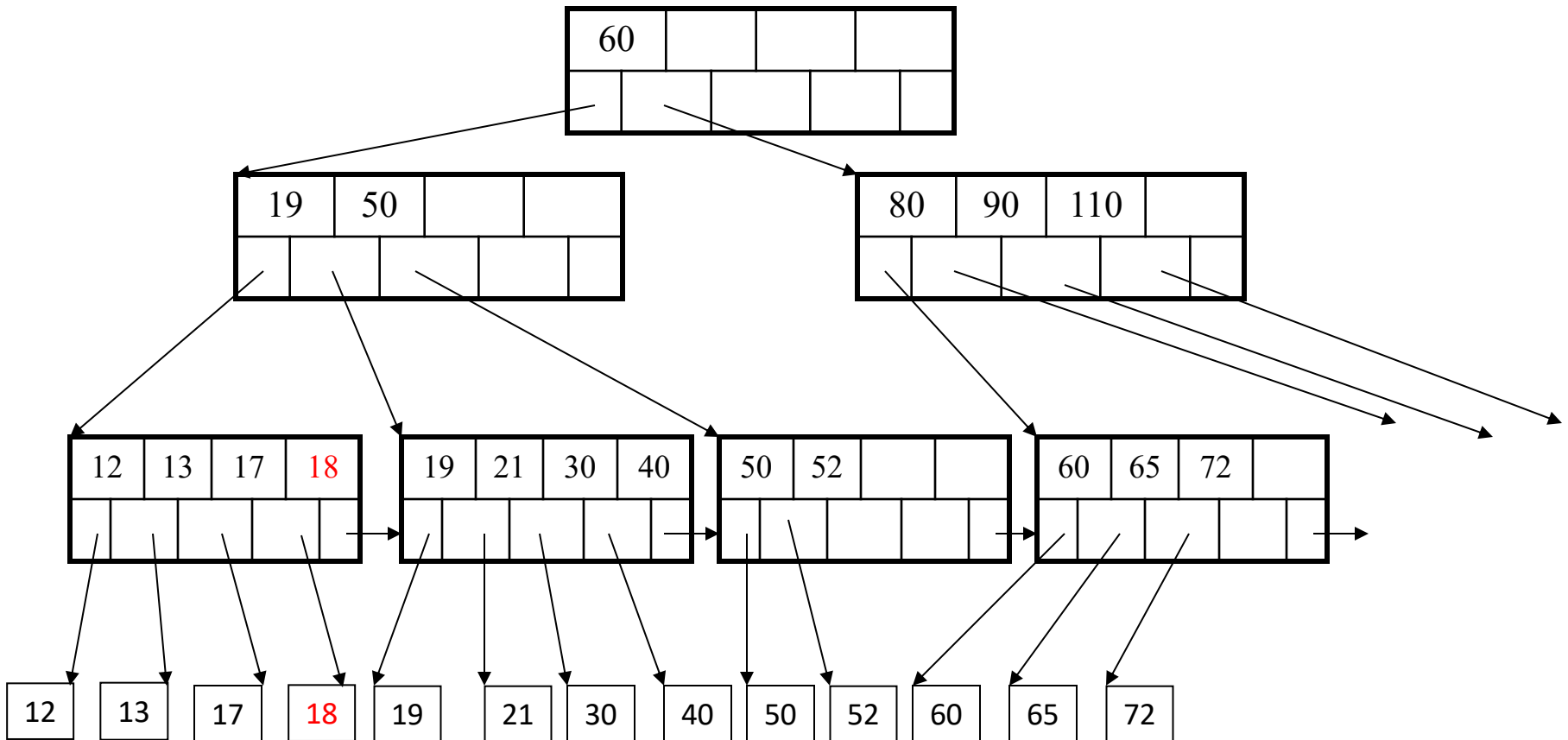
Insert K = 18
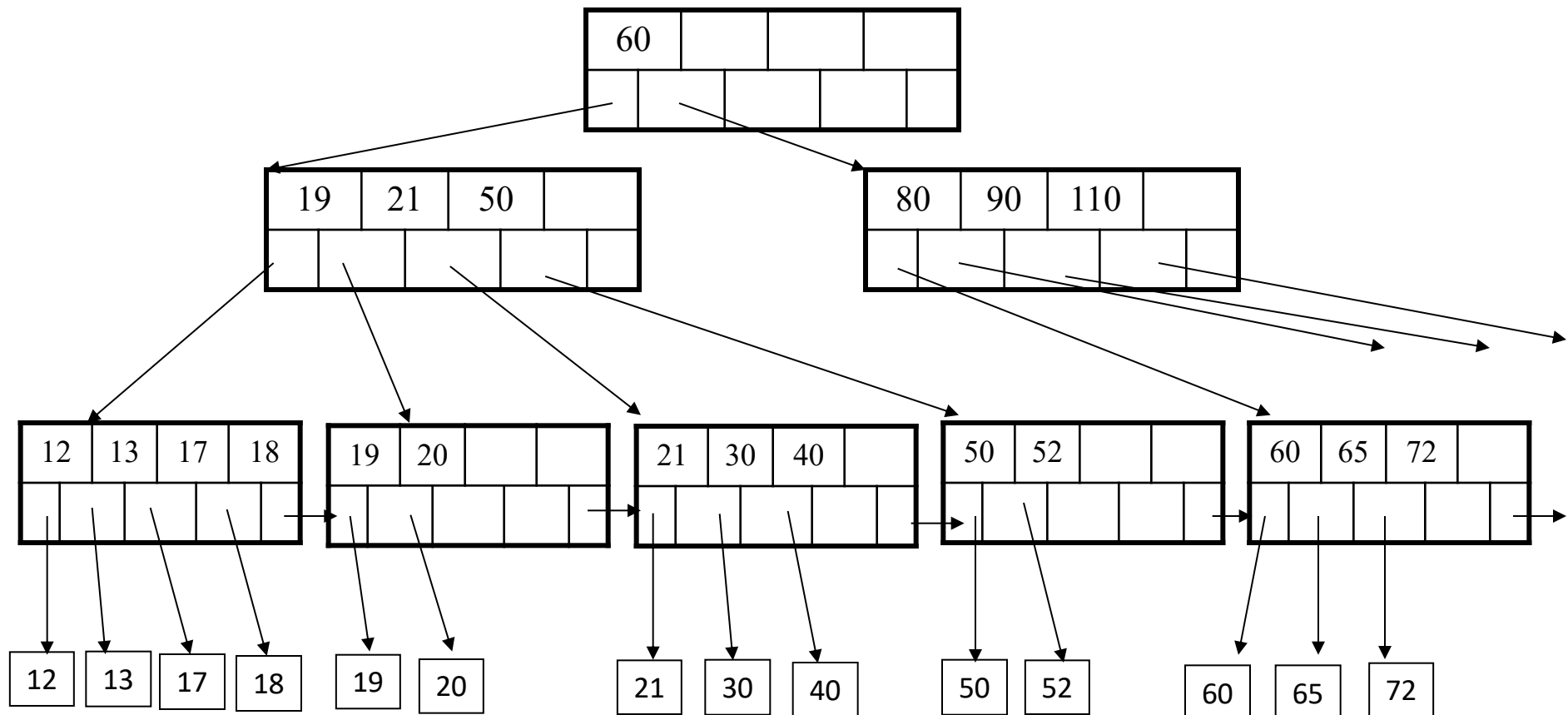
Insertion

Insert K= 20

Insertion

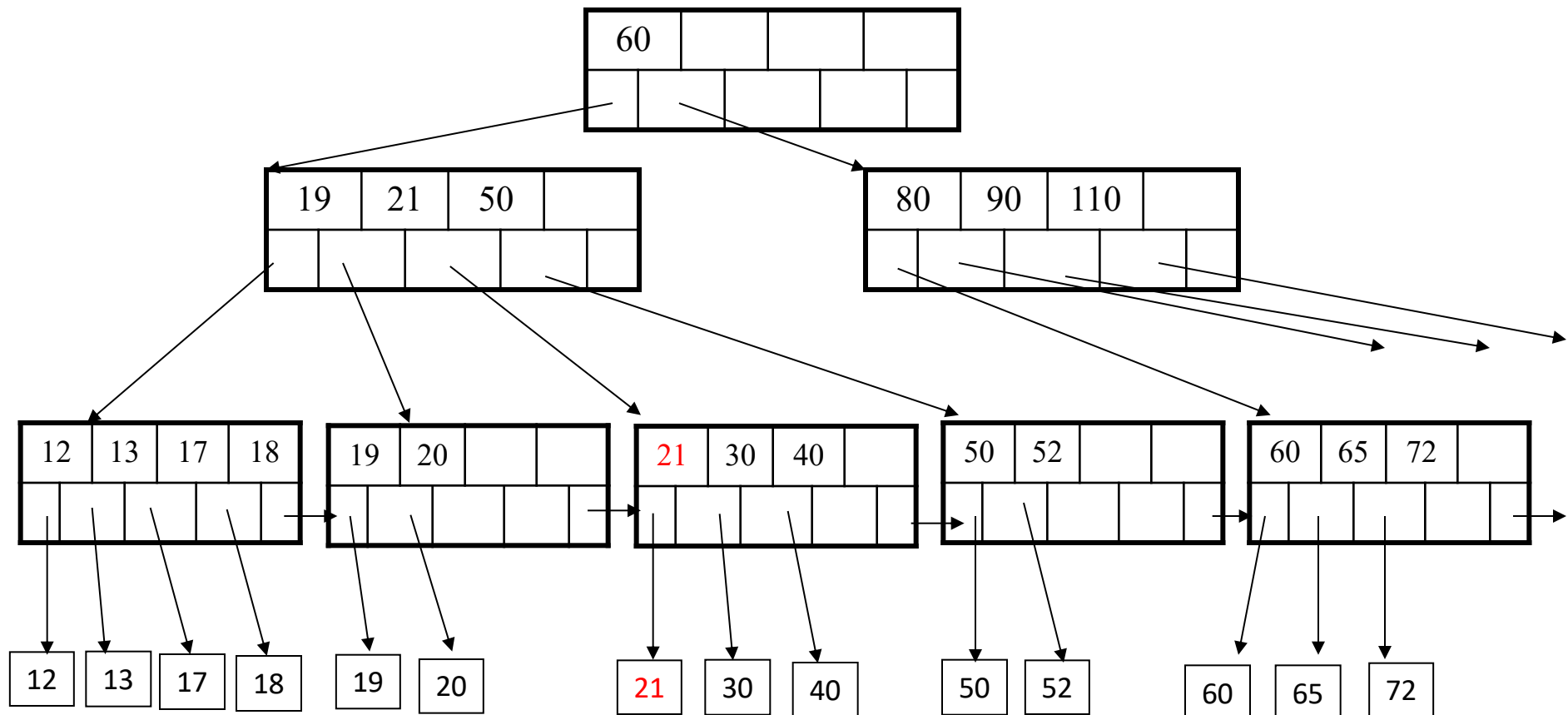Need to split the node

Insertion

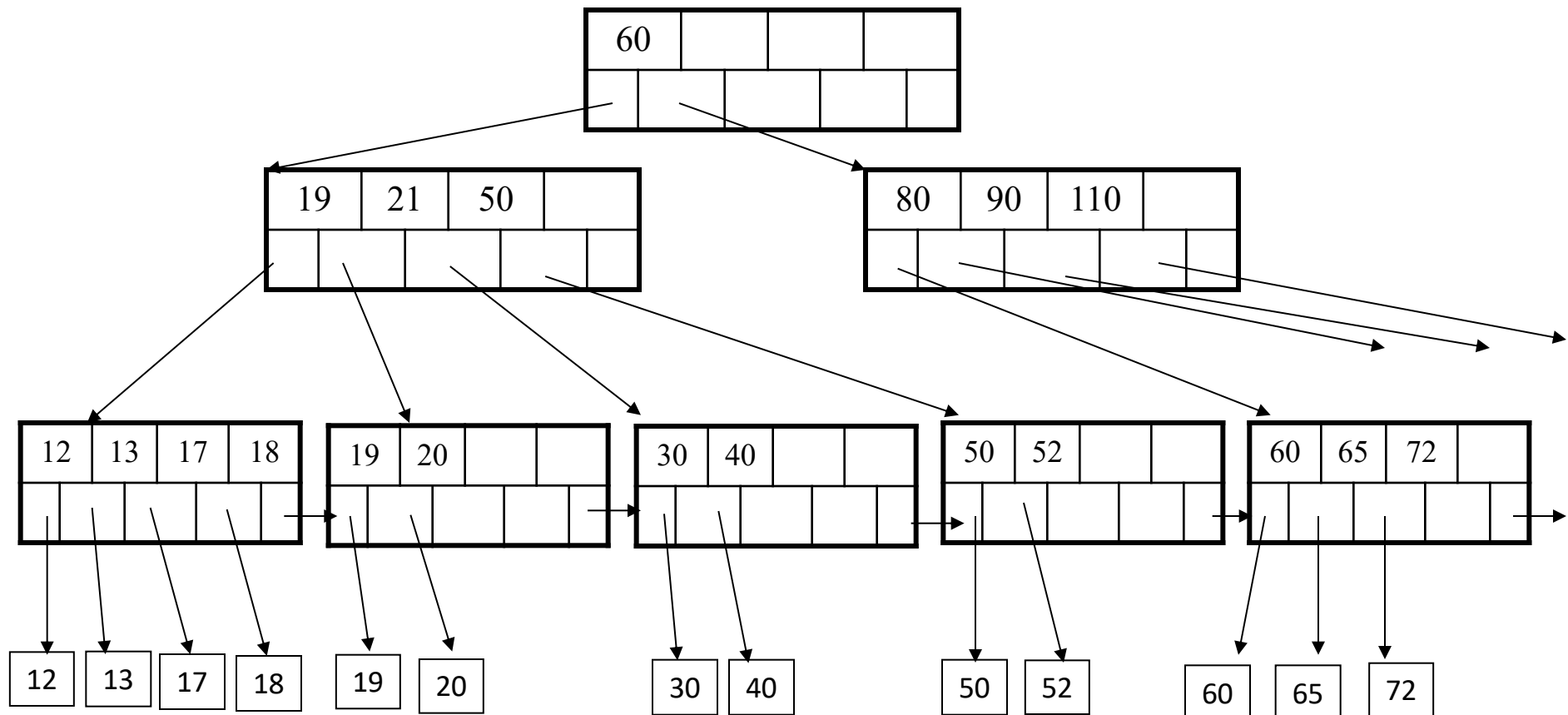Split and update the parent node.
What if we need to split the root?
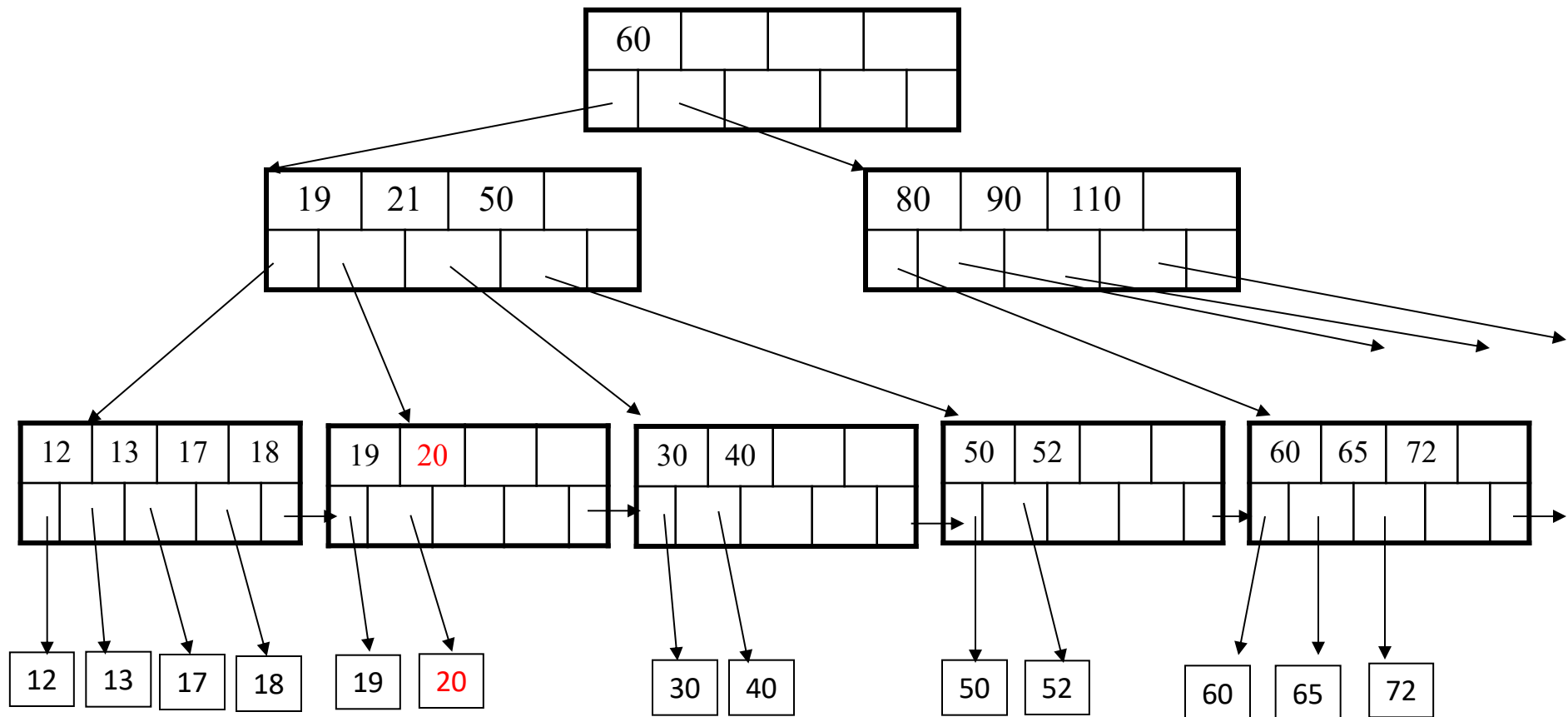
Deletion

Delete K = 21

Deletion

Note: K = 21 may still remain in the internal levels

Deletion

Delete K = 20

Deletion

We need to update the number of keys on the node:
 Borrow from siblings: rotate

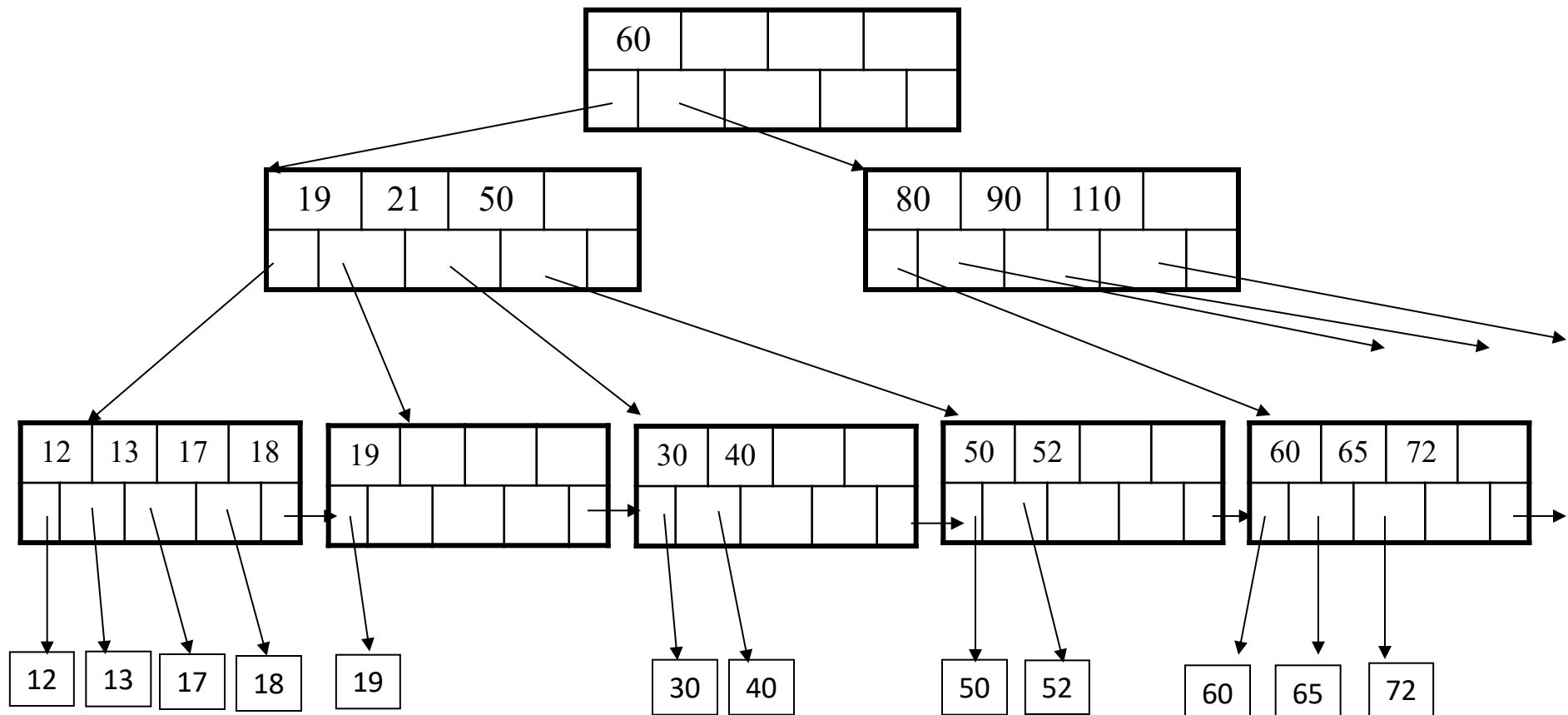# Deletion

We need to update the number of keys on the node:
Borrow from siblings: rotate
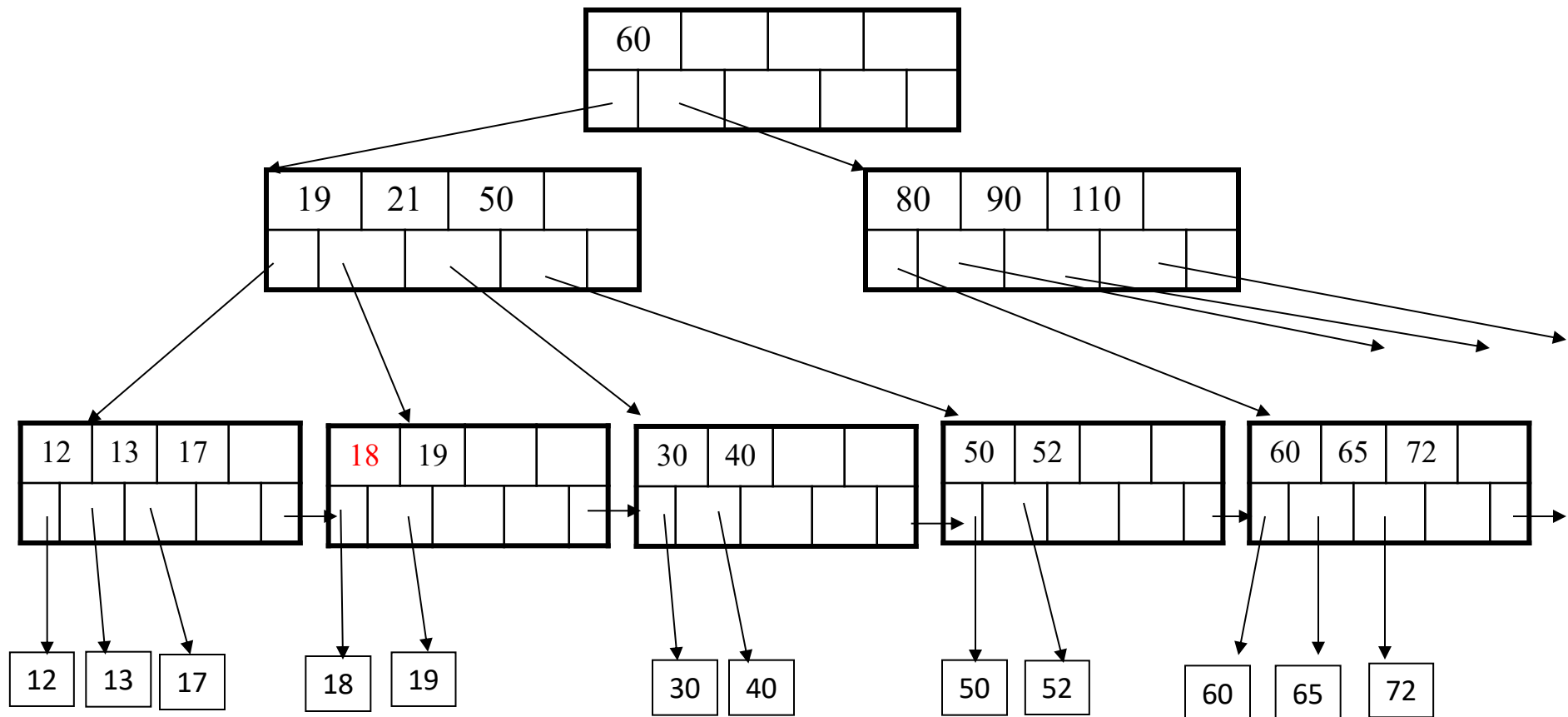
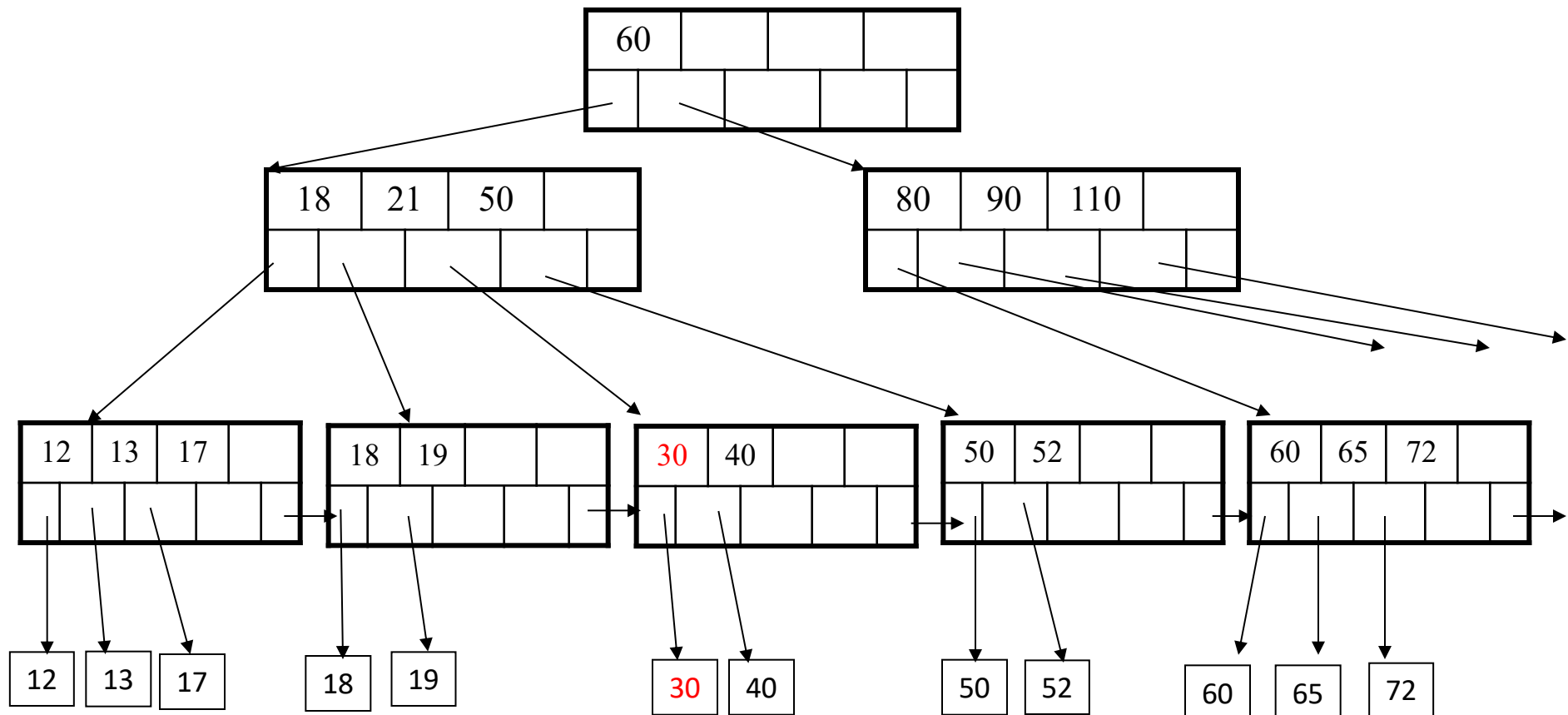# Deletion

We need to update the number of keys on the node:
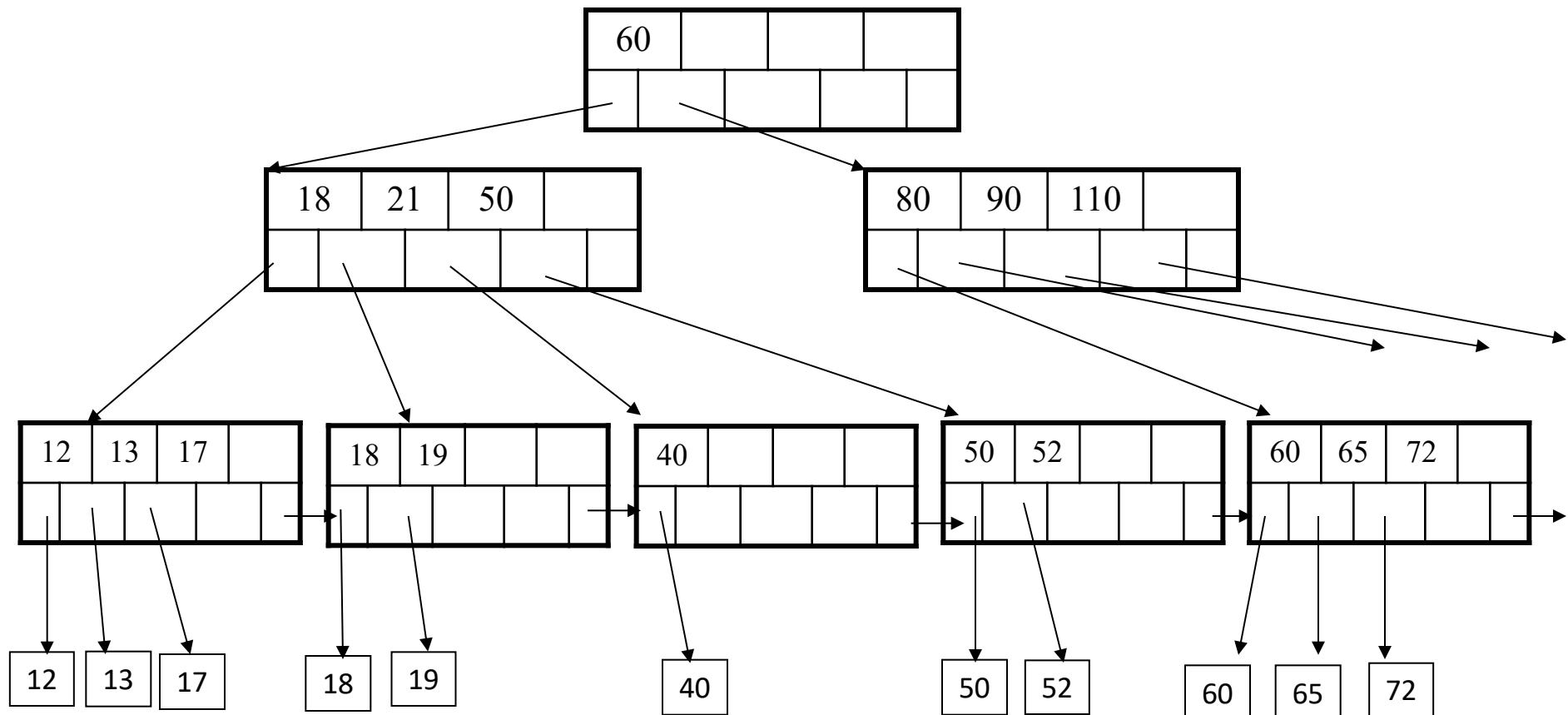Borrow from siblings: rotate
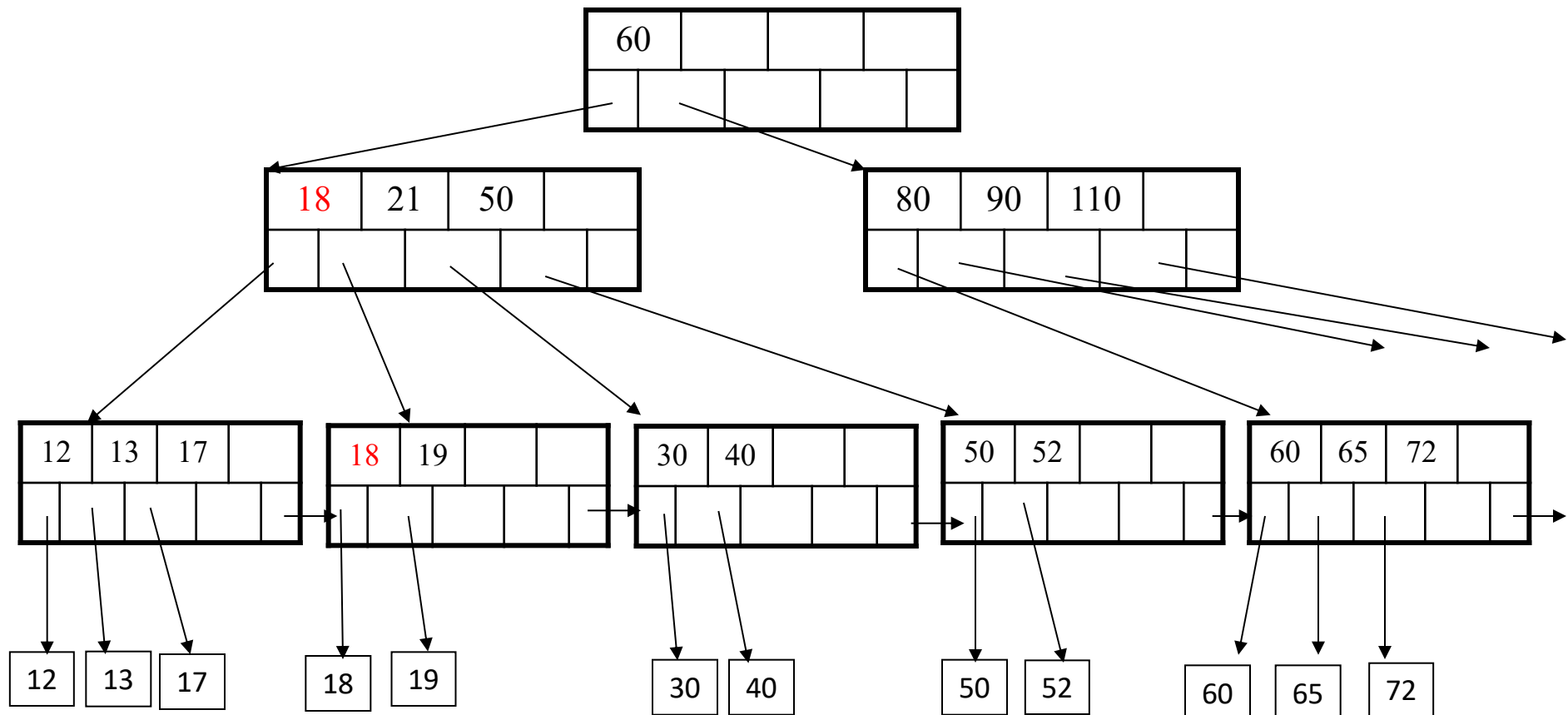
Deletion

What if we cannot borrow from siblings?
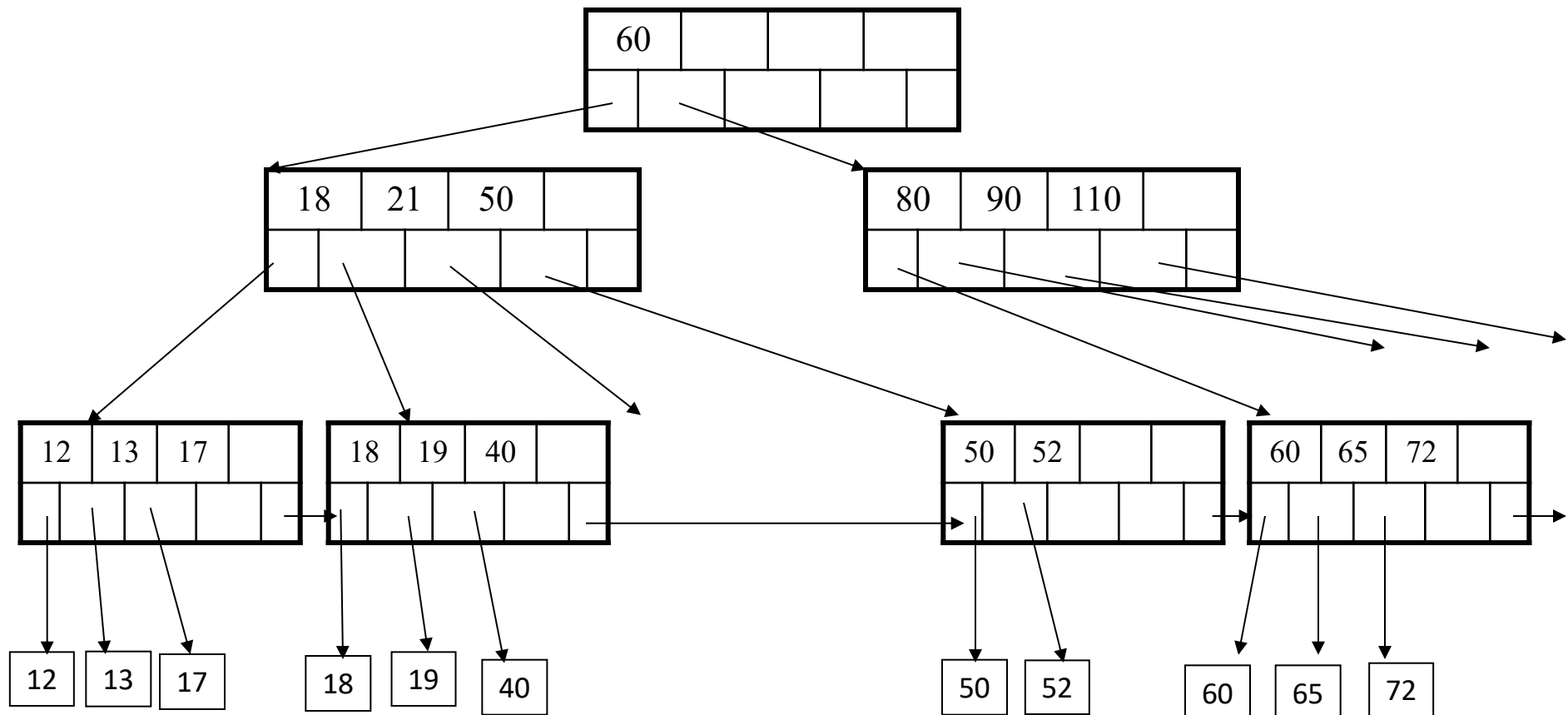Example: delete K = 30

Deletion

What if we cannot borrow from siblings?
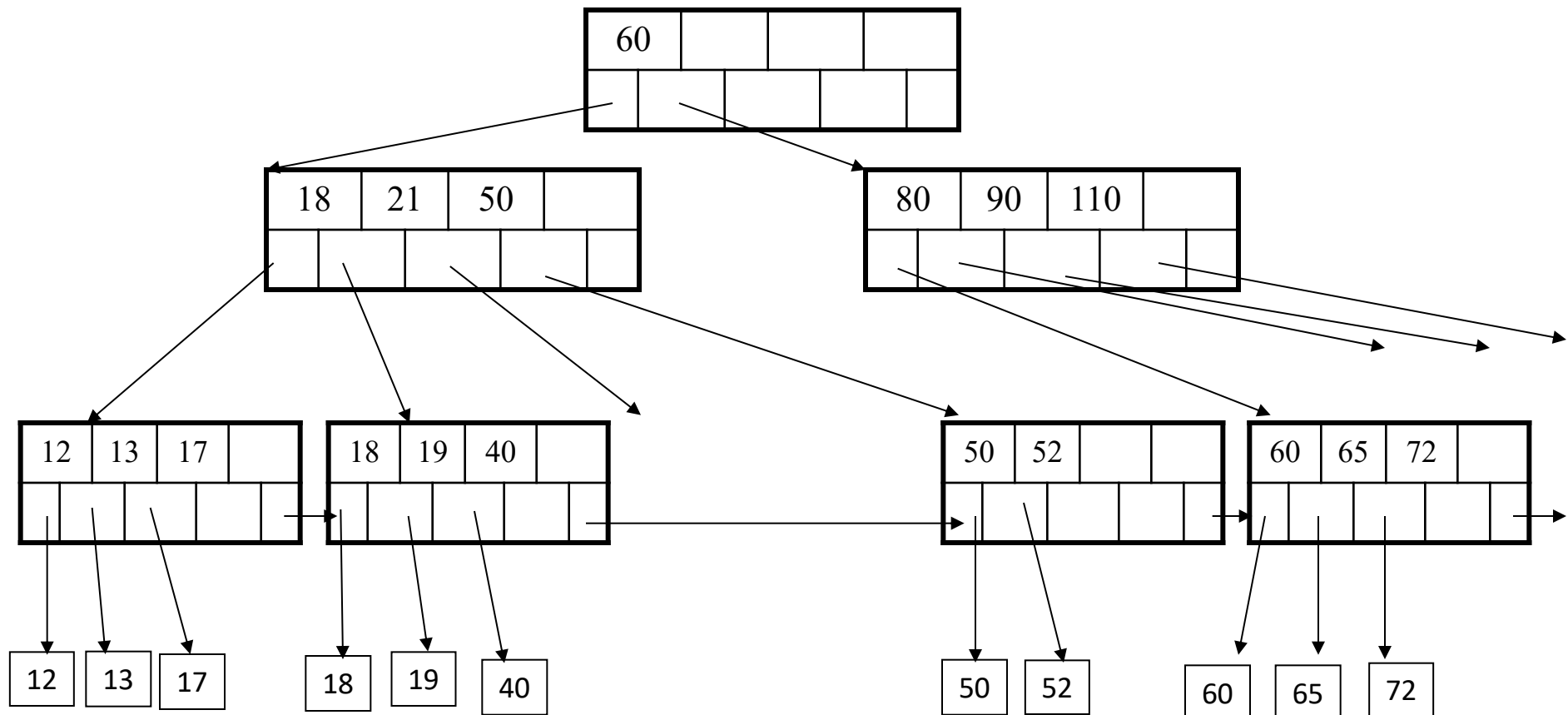Merge with a sibling.

Deletion

What if we cannot borrow from siblings?
Merge siblings!

Deletion

What to do with the dangling key and pointer? simply remove them

Deletion

Final tree

```
                              ┌──────┬──────┬──────┬──────┐
                              │  60  │      │      │      │
                              ├──────┼──────┼──────┼──────┤
                              │      │      │      │      │
                              └──────┴──────┴──────┴──────┘

        ┌──────┬──────┬──────┬──────┐          ┌──────┬──────┬──────┬──────┐
        │  18  │  50  │      │      │          │  80  │  90  │ 110  │      │
        ├──────┼──────┼──────┼──────┤          ├──────┼──────┼──────┼──────┤
        │      │      │      │      │          │      │      │      │      │
        └──────┴──────┴──────┴──────┘          └──────┴──────┴──────┴──────┘
```

```
┌────┬────┬────┬──┐  ┌────┬────┬────┬──┐   ┌────┬────┬──┬──┐  ┌────┬────┬────┬──┐
│ 12 │ 13 │ 17 │  │  │ 18 │ 19 │ 40 │  │   │ 50 │ 52 │  │  │  │ 60 │ 65 │ 72 │  │
├────┼────┼────┼──┤  ├────┼────┼────┼──┤   ├────┼────┼──┼──┤  ├────┼────┼────┼──┤
│    │    │    │  │  │    │    │    │  │   │    │    │  │  │  │    │    │    │  │
└────┴────┴────┴──┘  └────┴────┴────┴──┘   └────┴────┴──┴──┘  └────┴────┴────┴──┘
```

```
┌────┐ ┌────┐ ┌────┐  ┌────┐ ┌────┐ ┌────┐   ┌────┐ ┌────┐  ┌────┐ ┌────┐ ┌────┐
│ 12 │ │ 13 │ │ 17 │  │ 18 │ │ 19 │ │ 40 │   │ 50 │ │ 52 │  │ 60 │ │ 65 │ │ 72 │
└────┘ └────┘ └────┘  └────┘ └────┘ └────┘   └────┘ └────┘  └────┘ └────┘ └────┘
```

58
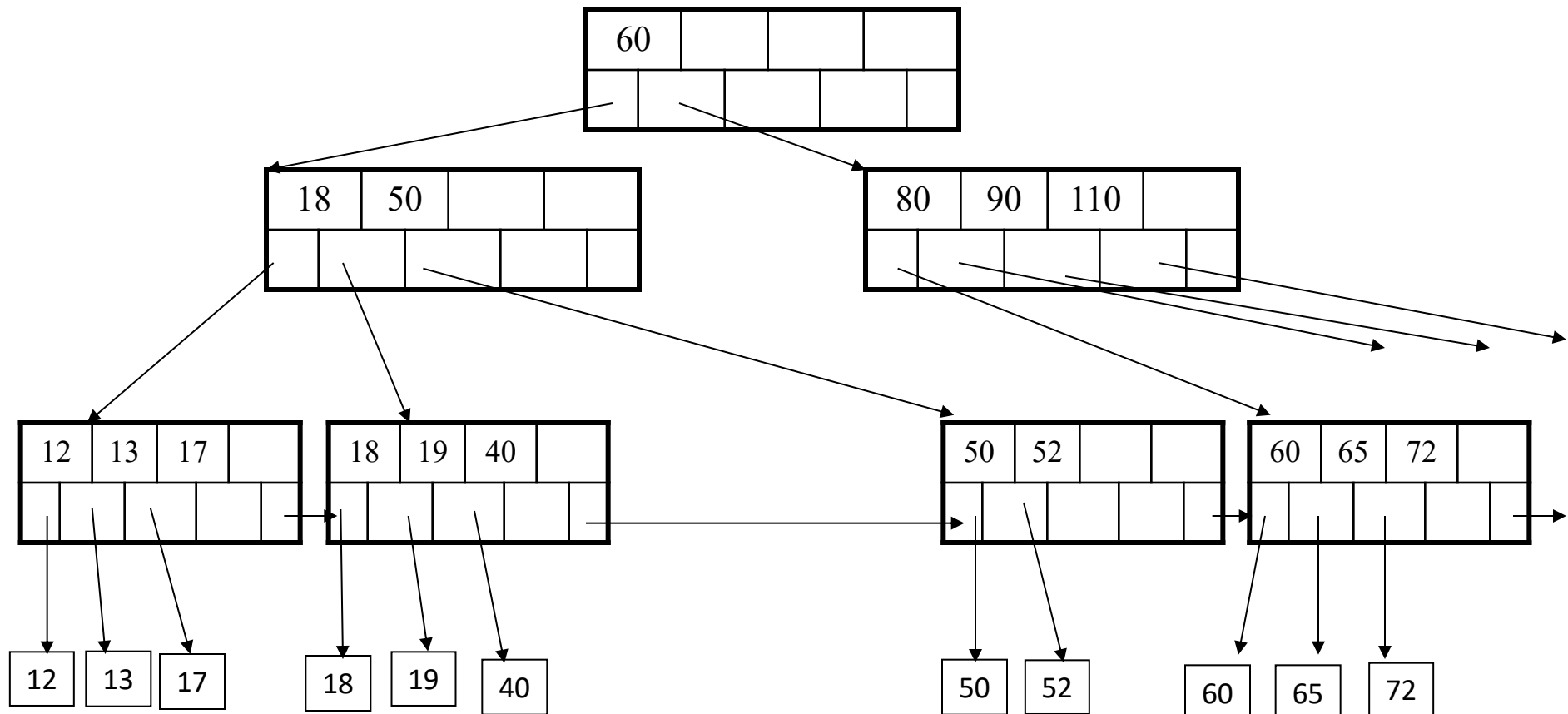
Prefix key compression

- Important to increase fan-out. (Why?)
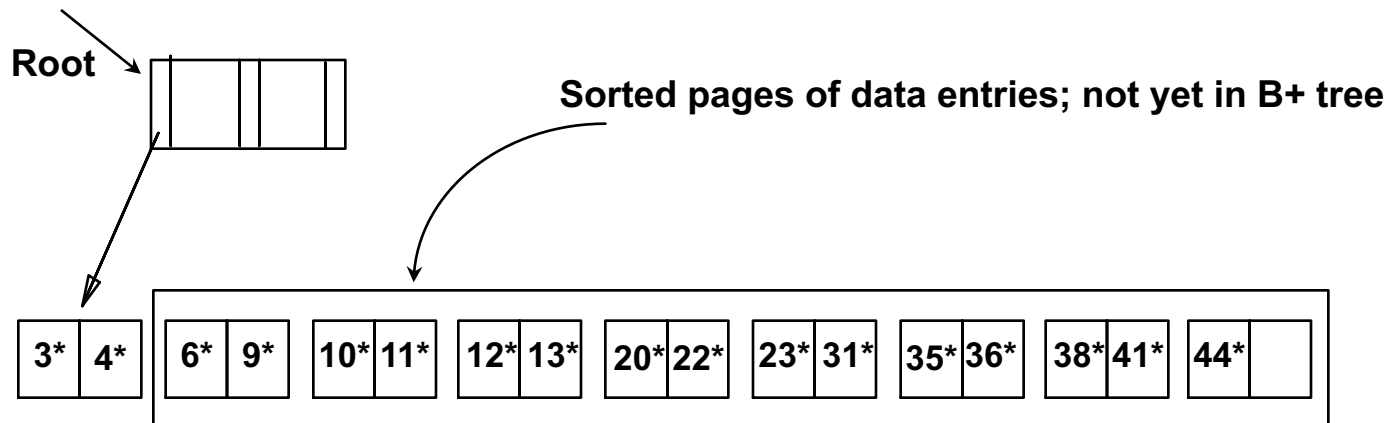- Key values in index entries only `direct traffic'; can often compress them.
  - E.g., If we have adjacent index entries with search key values *Dannon Yogurt*, *David Smith* and *Devarakonda Murthy*, we can abbreviate *David Smith* to *Dav*. (The other keys can be compressed too ...)
    - Is this correct? Not quite! What if there is a data entry *Davey Jones*? (Can only compress *David Smith* to *Davi*)
    - In general, while compressing, must leave each index entry greater than every key value (in any subtree) to its left.
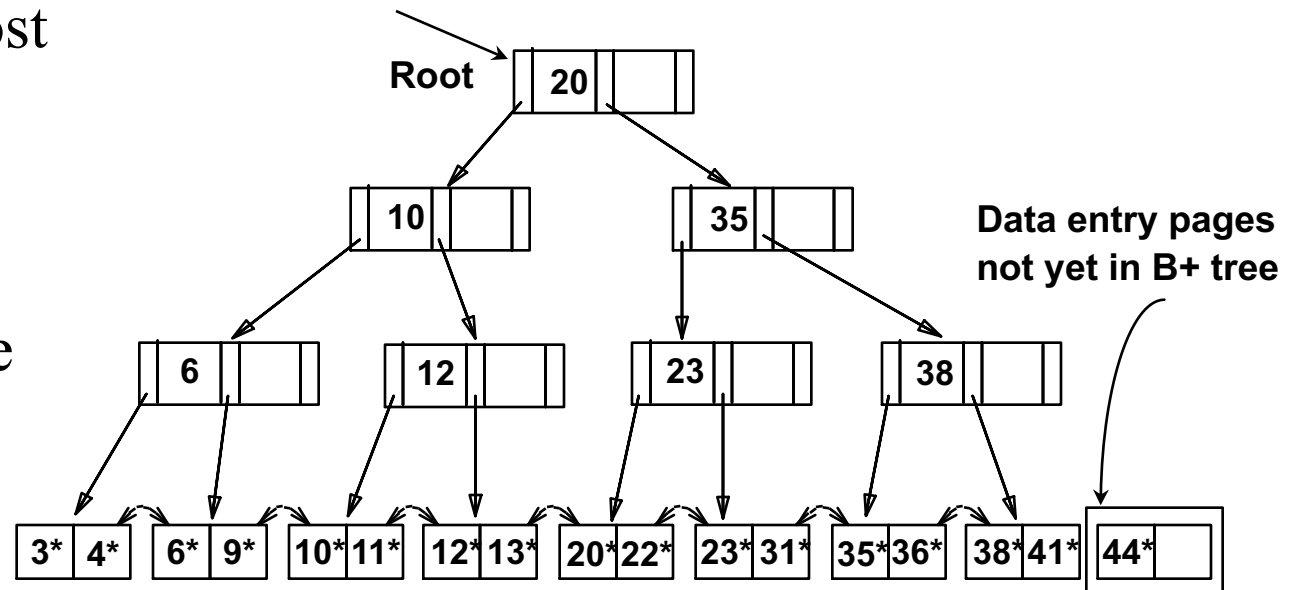- Insert/delete must be suitably modified.

Bulk loading of a B+ tree

- If we have a large collection of records, and we want to create a B+ tree on some field, doing so by repeatedly inserting records is very slow.

- *Bulk Loading* can be done much more efficiently.

- *Initialization*: Sort all data entries, insert pointer to first (leaf) page in a new (root) page.

Root

Sorted pages of data entries; not yet in B+ tree

| 3* | 4* | | 6* | 9* | | 10* | 11* | | 12* | 13* | | 20* | 22* | | 23* | 31* | | 35* | 36* | | 38* | 41* | | 44* | |

Bulk loading

- Index entries for leaf pages always entered into right-most index page just above leaf level. When this fills up, it splits. (Split may go up right-most path to the root.)

- Much faster than repeated inserts, especially when one considers locking!

Root

| 10 | 20 |

| 6 | | | 12 | | | 23 | 35 |

Data entry pages not yet in B+ tree

| 3* | 4* | | 6* | 9* | | 10* | 11* | | 12* | 13* | | 20* | 22* | | 23* | 31* | | 35* | 36* | | 38* | 41* | | 44* | |

Root

| 20 | |

| 10 | | | | 35 | | |

Data entry pages not yet in B+ tree

| 6 | | | 12 | | | 23 | | | 38 | |

| 3* | 4* | | 6* | 9* | | 10* | 11* | | 12* | 13* | | 20* | 22* | | 23* | 31* | | 35* | 36* | | 38* | 41* | | 44* | |
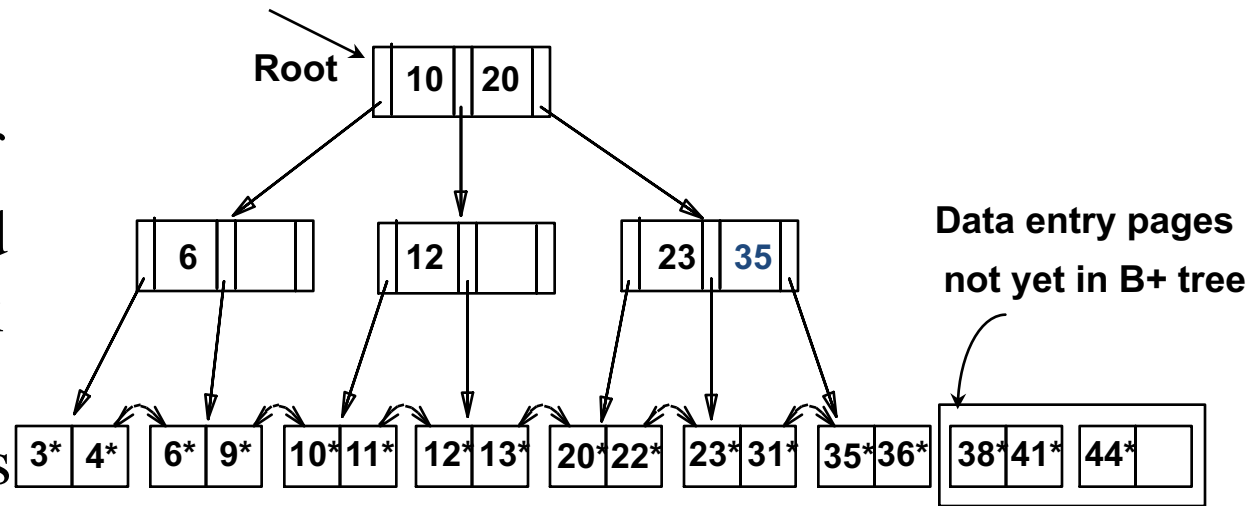
Hash Table

- Very similar to main memory hash tables, but much larger
- Main memory hash tables
  - n available buckets
  - A hash function f(k) maps a key k to {0, …, n-1}
  - Bucket f(k) points to the records whose keys are k.
- Secondary storage hash tables
  - Bucket = block
  - Block f(k) stores the records with key k.
  - Overflow blocks when needed.

Example

- Assume 1 bucket (block) stores 2 records
- $h(e)=0$
- $h(b)=h(f)=1$
- $h(g)=2$
- $h(a)=h(c)=3$

0	e
1	b
	f
2	g
3	a
	c

Searching a Hash Table

- Search for a:
- Compute h(a)=3
- Read bucket 3
- 1 disk access

Main memory may have an array of pointers (to buckets) accessible by bucket number.

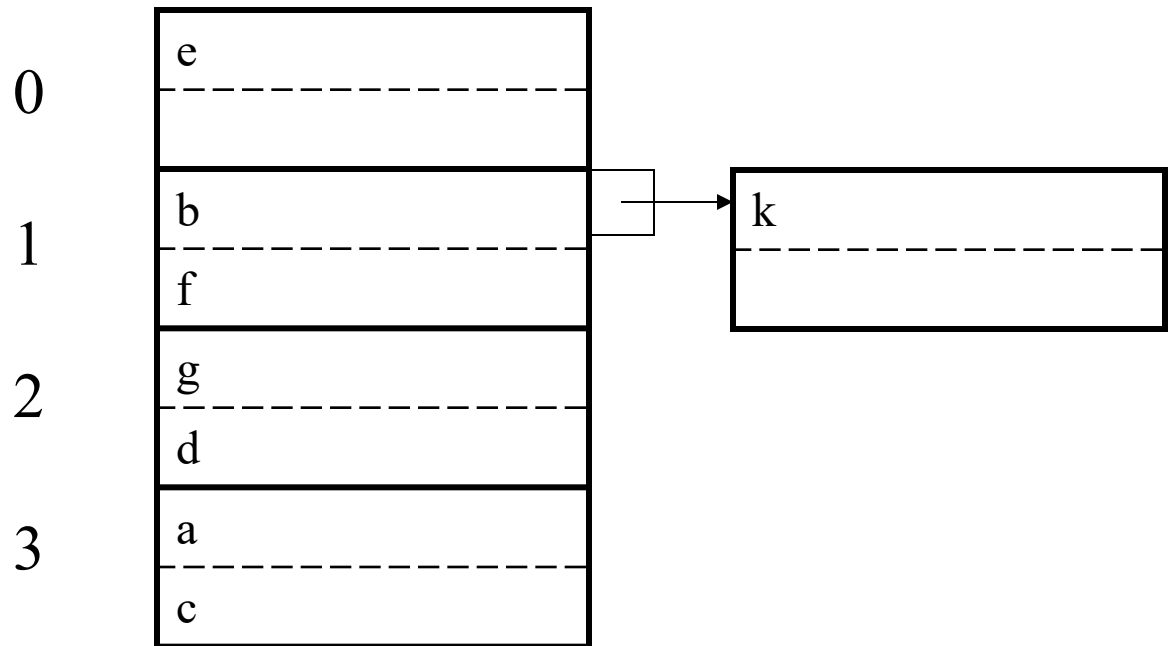|	
0	e
1	b / f
2	g
3	a / c

Insertion in Hash Table

- Place in right bucket, if space
- E.g. h(d)=2

0	e
1	b / f
2	g / d
3	a / c

Insertion in Hash Table

- Create overflow block, if no space
- E.g. h(k)=1

- More over-
 flow blocks
  may be needed

Hash Table Performance

- Excellent, if no overflow blocks

- Degrades when #keys > #buckets; many overflow blocks.

- Closer to full scan.

Extensible Hash Table

- Allows hash table to grow, to avoid performance degradation

- Assume a hash function h that returns numbers in $\{0, \ldots, 2^k - 1\}$

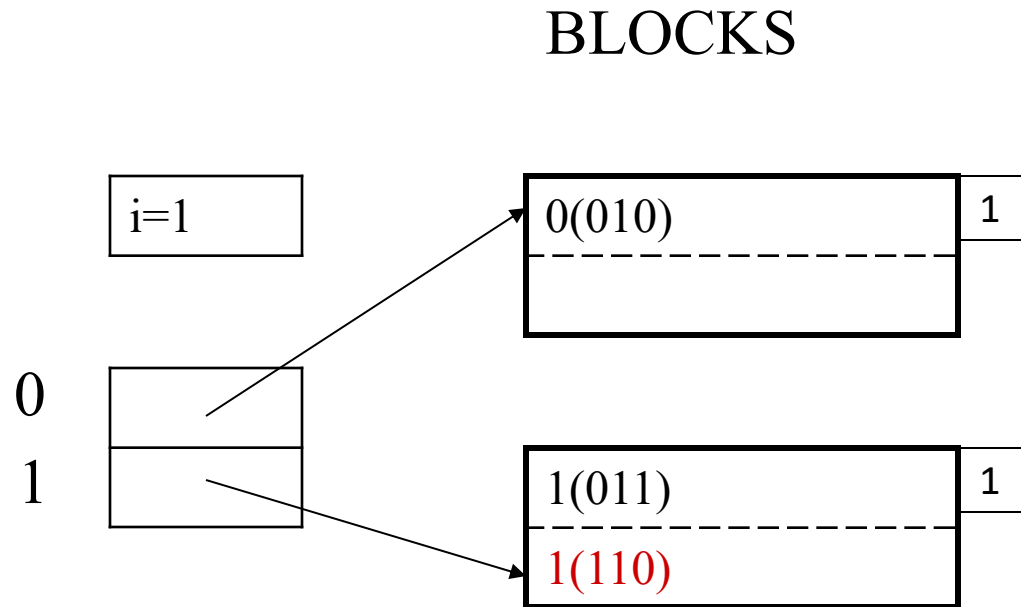- Start with n = $2^i << 2^k$, only look at first i most significant bits

Extensible Hash Table

- E.g. i=1, n=2, k=4

BLOCKS

| i=1 |

0(010) | 1 |

0

1

1(011) | 1 |
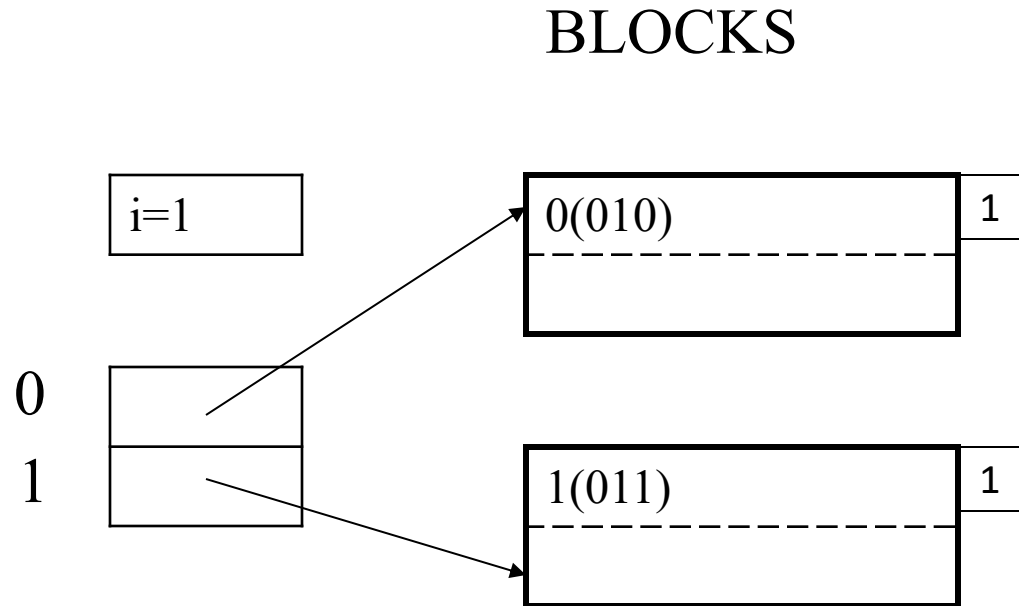
- We only look at the first bit (0 or 1).
- Note: similar algorithm for the last bit.

Insertion in Extensible Hash Table

- Insert 1110

BLOCKS

i=1

```
0 [ ]
1 [ ]
```

0(010)	1

1(011)	1
1(110)	

Insertion in Extensible Hash Table

- Insert 1010

BLOCKS

| i=1 |

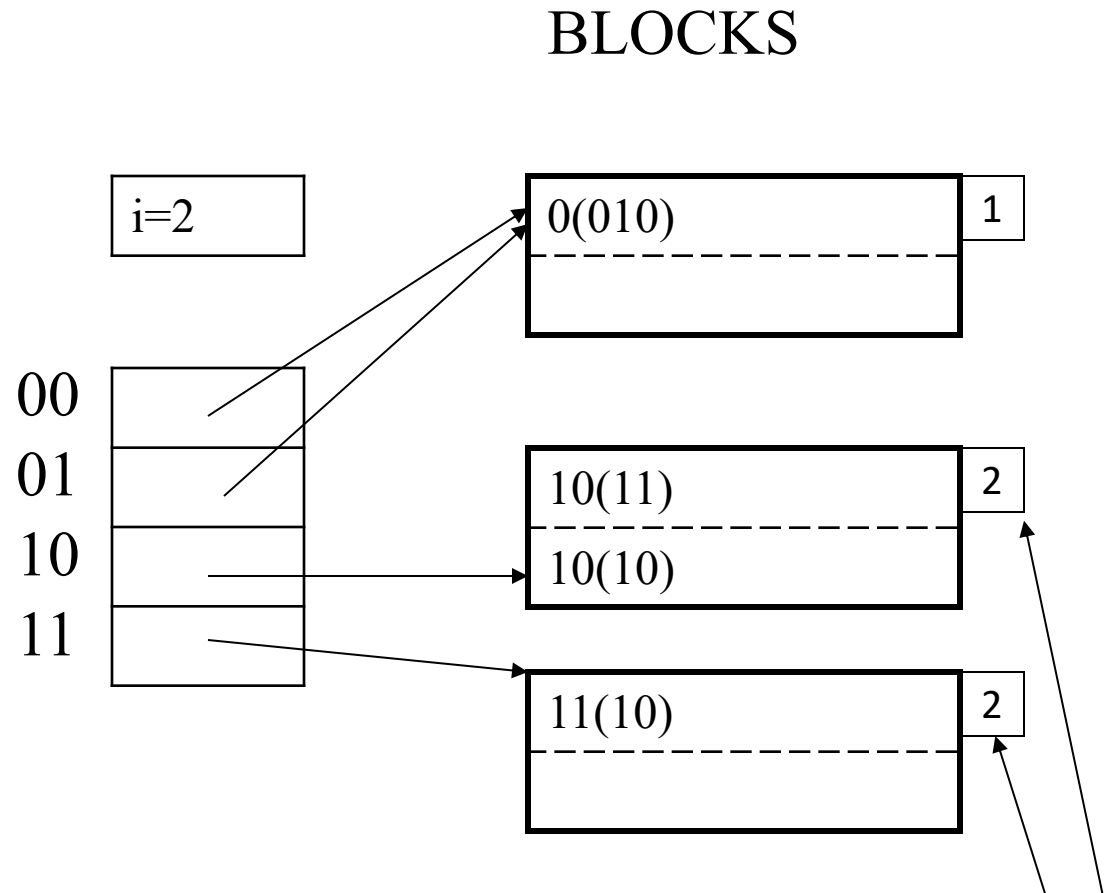| 0(010)	1

0

1

1(011)	1
1(110), 1(010)	
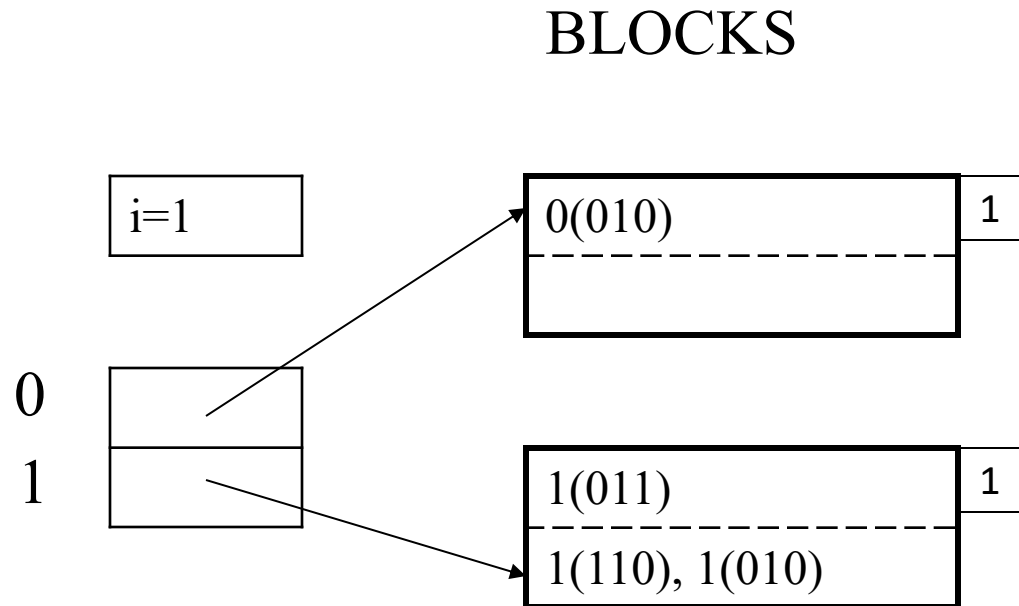
- Split block and extend bucket array
- Need more bits to address: i becomes 2

Insertion in Extensible Hash Table

- Insert 1010

BLOCKS

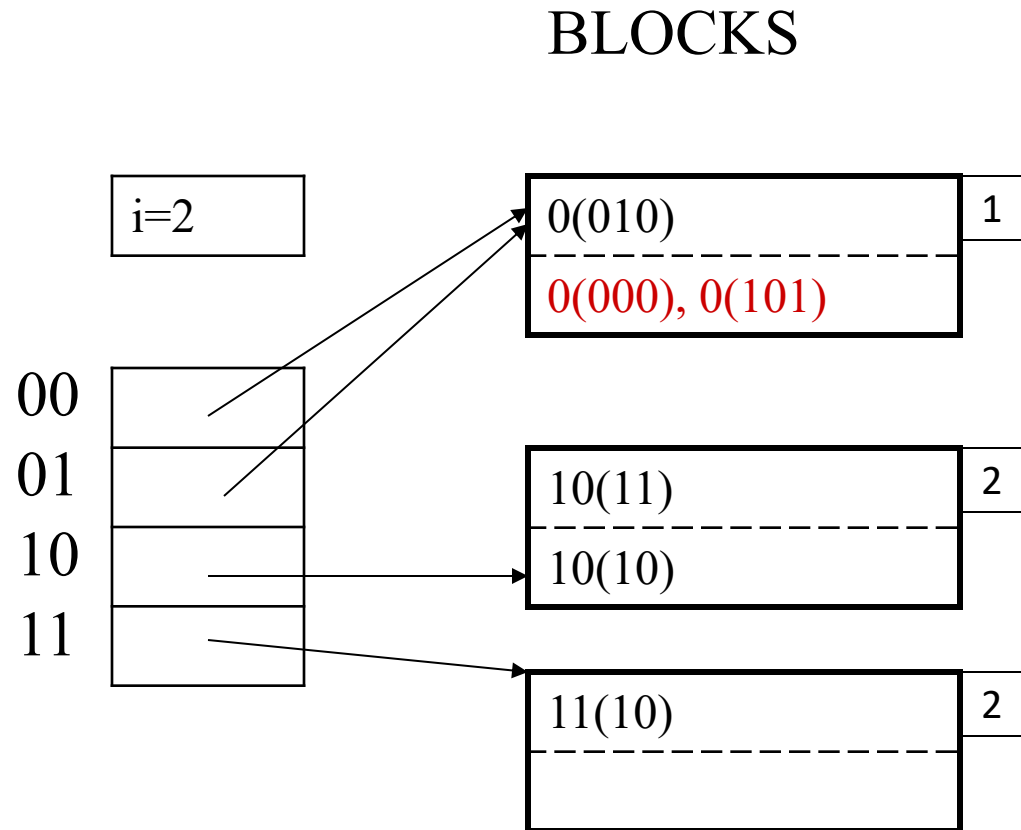| i=2 |
| --- |

0(010)	1

```
00
01
10
11
```

10(11)	2
10(10)	

11(10)	2

Update the number of bits.

Insertion in Extensible Hash Table

- Insert 0000 and 0101

BLOCKS

i=2	

0(010)	1
0(000), 0(101)	

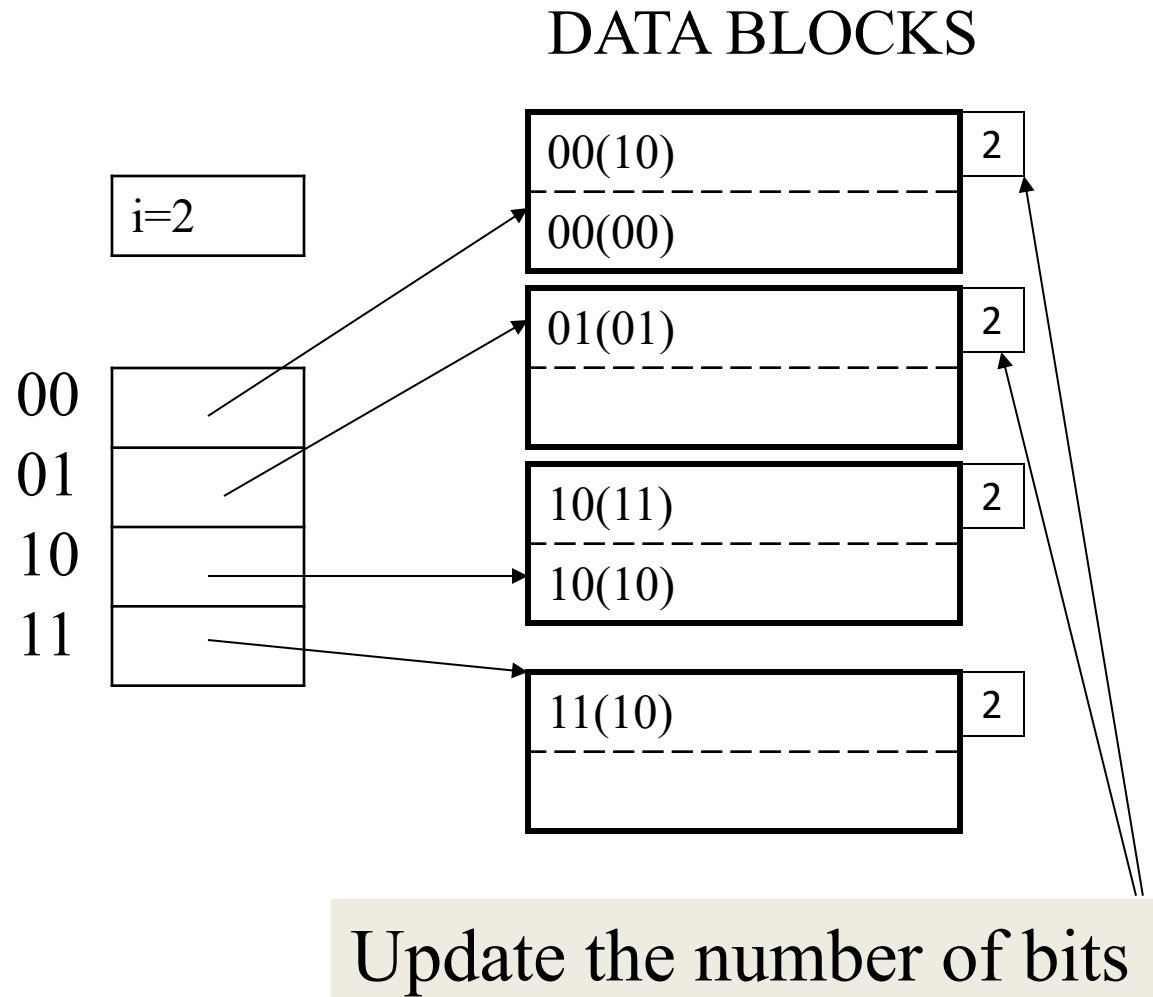| 00	
01	
10	
11	

10(11)	2
10(10)	

11(10)	2

- Split the block, re-use the bucket array.

Insertion in Extensible Hash Table

- After splitting the block

DATA BLOCKS

i=2

```
00
01
10
11
```

00(10)	2
00(00)	

01(01)	2

10(11)	2
10(10)	

11(10)	2

Update the number of bits

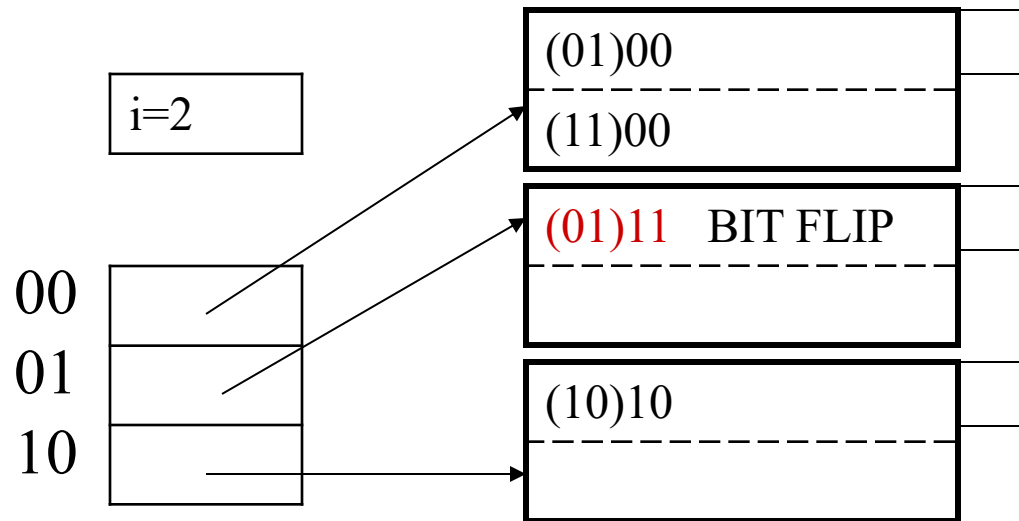# Extensible Hash Table: Performance

- No overflow blocks: always one read

- Not as good as it sounds:
  - Extensions are costly and disruptive
  - After an extension, the table may no longer fit in main memory
  - Example: three records whose keys share the first 30 bits. A block split would require setting i = 30, i.e., accommodating for 2^30 buckets!
  - Many useless buckets

Linear Hash Table

- Add only one bucket at a time
- But n is no longer a power of 2
- Let i be #bits required to address n buckets.
  - $2^{i-1} < n <= 2^i$
- After computing h(k), use last i bits:
  - If last i bits represent a number >= n, change msb from 1 to 0 (get a number < n)
- Allow overflow blocks

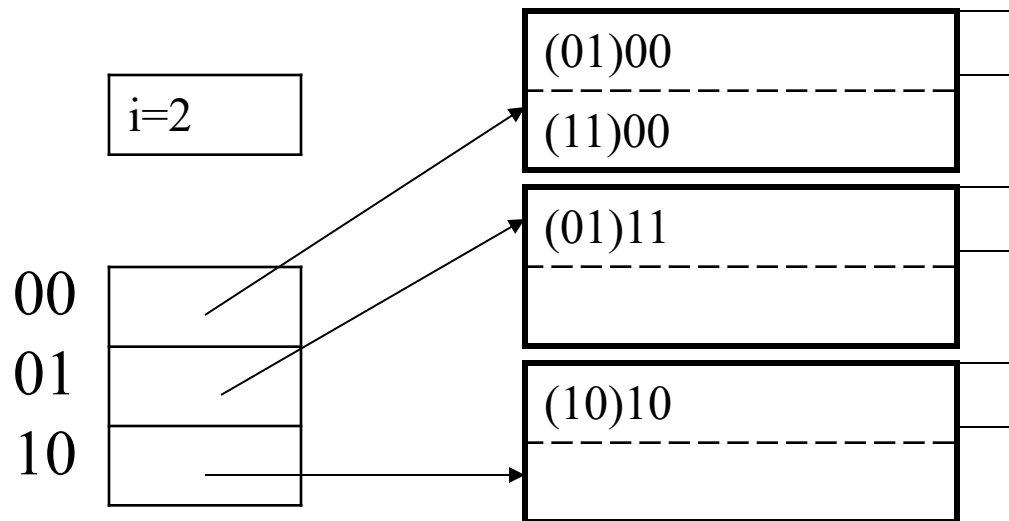# Linear Hash Table Example

- N=3

i=2

(01)00
- - - - - - - - - - - - - -
(11)00

(01)11   BIT FLIP

00
01
10

(10)10

Linear Hash Table Example

- Insert 1000:

	i=2	

(01)00
- - - - - - - -
(11)00

(01)11
- - - - - - - -

(10)10
- - - - - - - -

00
01
10

Linear Hash Table Example

- Insert 1000: overflow blocks…

i=2

```
        (01)00              (10)00
        --------            --------
        (11)00

        (01)11
        --------

00

01      (10)10
        --------
10
```
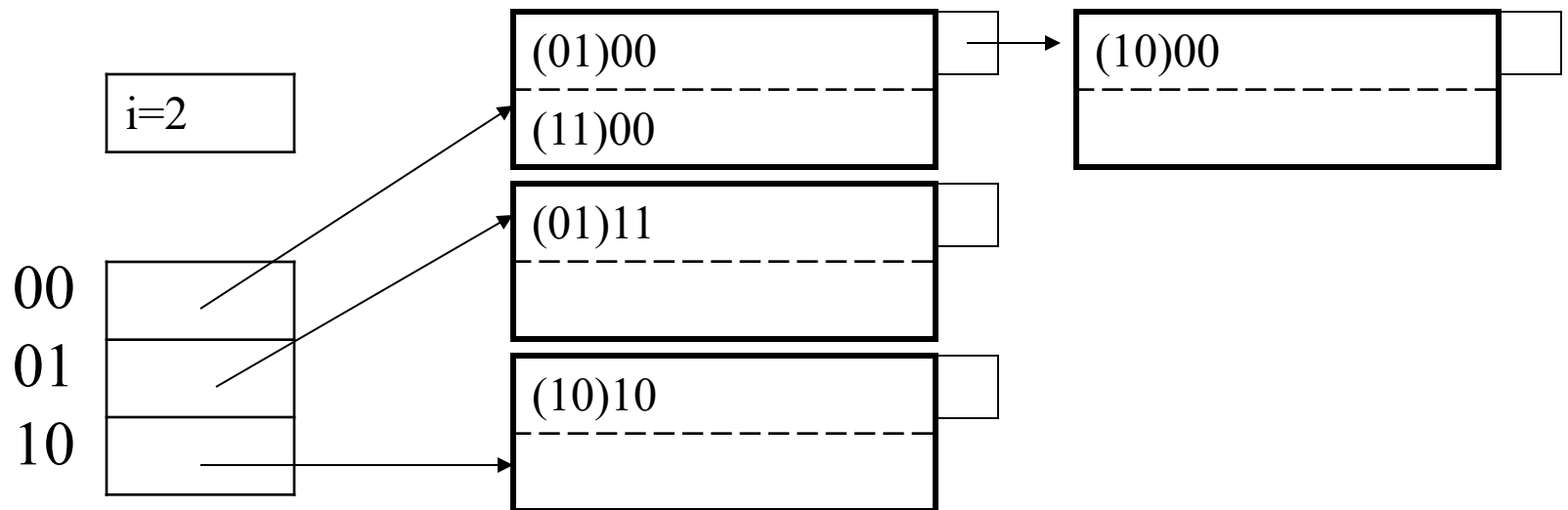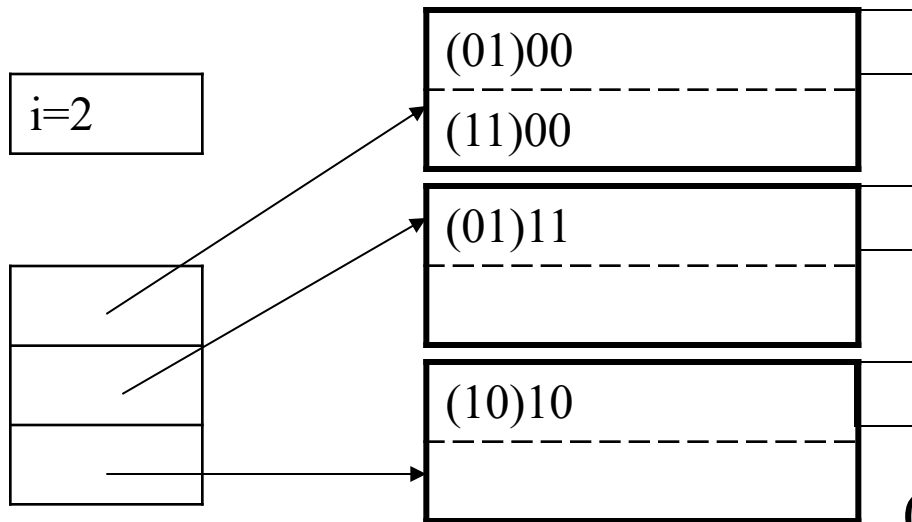
Linear Hash Tables
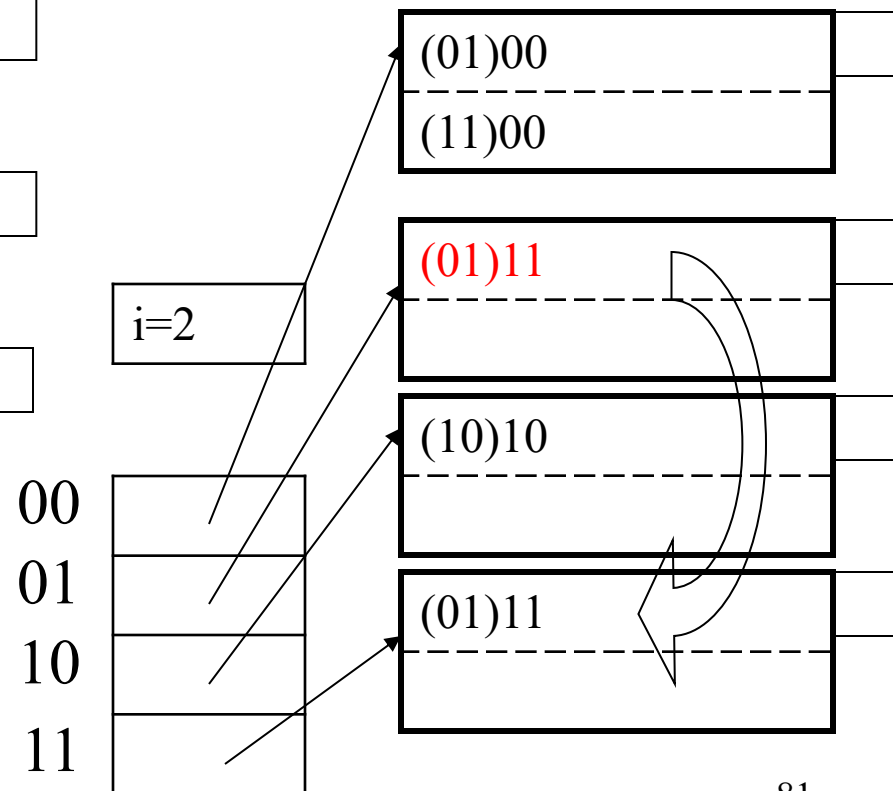
- Extend n:=n+1 when average number of records per block exceeds some number, e.g 85%
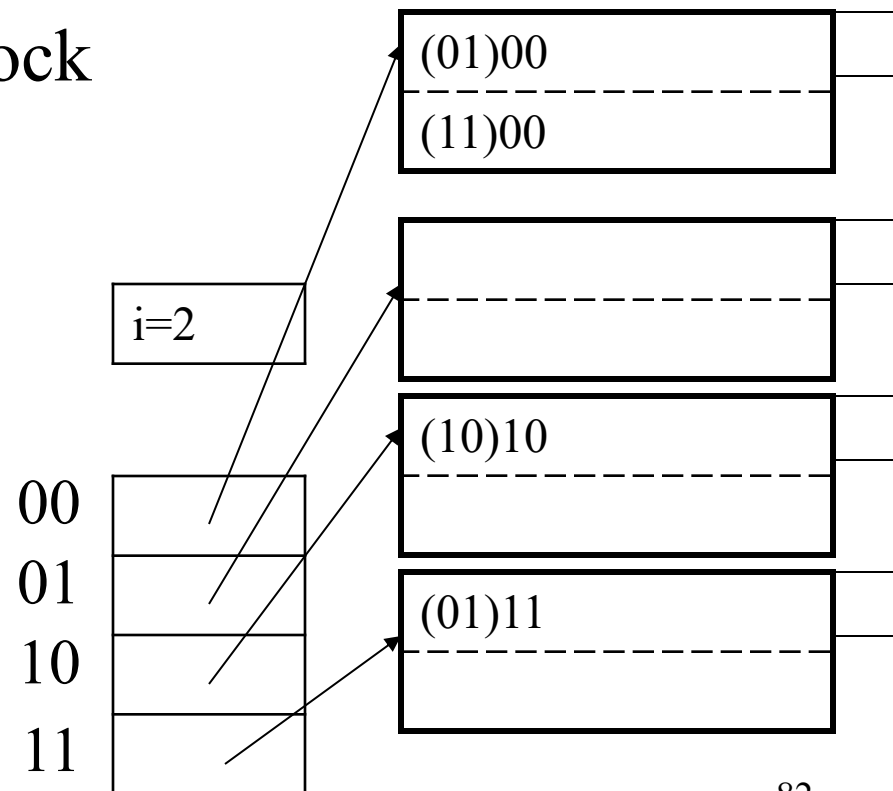
Linear Hash Table Extension

- From n=3 to n=4

i=2

(01)00
(11)00

(01)11

(10)10

00
01
10

How many block access?

i=2

(01)00
(11)00

(01)11

(10)10

(01)11

00
01
10
11

81

Linear Hash Table Extension

- Extension from n=4 to n=5 (new bit)
- How many blocks to access?
  - Need to access every block

i=2

```
00
01
10
11
```

(01)00
(11)00

(10)10

(01)11

Index Creation

CREATE TABLE Person(Name varchar(50), Pos int, Age int);
CREATE INDEX Person_ID ON Person(ID);

Default is normally B-tree.

CLUSTER Person USING ON Person_ID;

Cluster Person_ID index

CREATE INDEX Pos_Age ON Person(Pos, Age);

Multi-attribute index

Index Selection

- Queries that benefit from an index over A
  - Point queries on A.
  - Range queries on A
  - Joins on A
- Some join algorithms are faster using hash indexes.

Index Selection

- Let's index every attribute on every table to speed up all queries!

- Indexes slow down data manipulation
  - INSERT, DELETE, UPDATE.
  - Not all updates

    UPDATE Person

    SET Age = 34

    WHERE Pos = 2;

Index Selection

- Given a ***query workload*** and a ***schema***, find the set of indexes that optimize the execution.

- The query workload**:**

  - Queries and their frequencies.
  - Queries are both data retrieval (SELECT) and data manipulation (INSERT, UPDATE, DELETE).

Index Selection

- Part of **physical database design**
  - File structure, indexing, tuning queries,…
- Physical database design may affect logical design!
  - Change the schema to run the queries faster
  - We talk more on database tuning

Index Selection

- Generally, a hard problem.
- RDBMS vendors provide wizards:
  - Started with AutoAdmin project for SQL Server
  - SQL Server/ Oracle Index Tuning Wizard
  - DB2 Index Advisor
- They try many configurations and pick the one that minimizes the time and overheads.