Lecture 6:
Data Storage and Indexes

Tuesday, February 13, 2007

Outline

- Storage and indexing: Chapter 8
- B+ trees: Chapter 10
- Hash-based indexes: Chapter 11

Disks and Files

- DBMS stores information on (hard) disks.
- This has major implications for DBMS design!
  - READ: transfer data from disk to main memory
  - WRITE: transfer data from RAM to disk.
- Both are high-cost operations, relative to in-memory operations, so must be planned carefully!

Why Not Store Everything in Main Memory?

- Costs too much. $1000 will buy you either 128MB of RAM or 7.5GB of disk today.
- Main memory is volatile. We want data to be saved between runs. (Obviously!)
- Typical storage hierarchy:
  - Main memory (RAM) for currently used data.
  - Disk for the main database (secondary storage).
  - Tapes for archiving older versions of the data (tertiary storage).

Arranging Pages on Disk

- 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!

Representing Data Elements

- Relational database elements:

```
CREATE TABLE Product (

        pid INT PRIMARY KEY,
        name CHAR(20),
        description VARCHAR(200),
        maker CHAR(10) REFERENCES Company(name)
)
```

- A tuple is represented as a record
- The table is a sequence of records

Issues

- Managing free blocks
- Represent the records inside the blocks
- Represent attributes inside the records

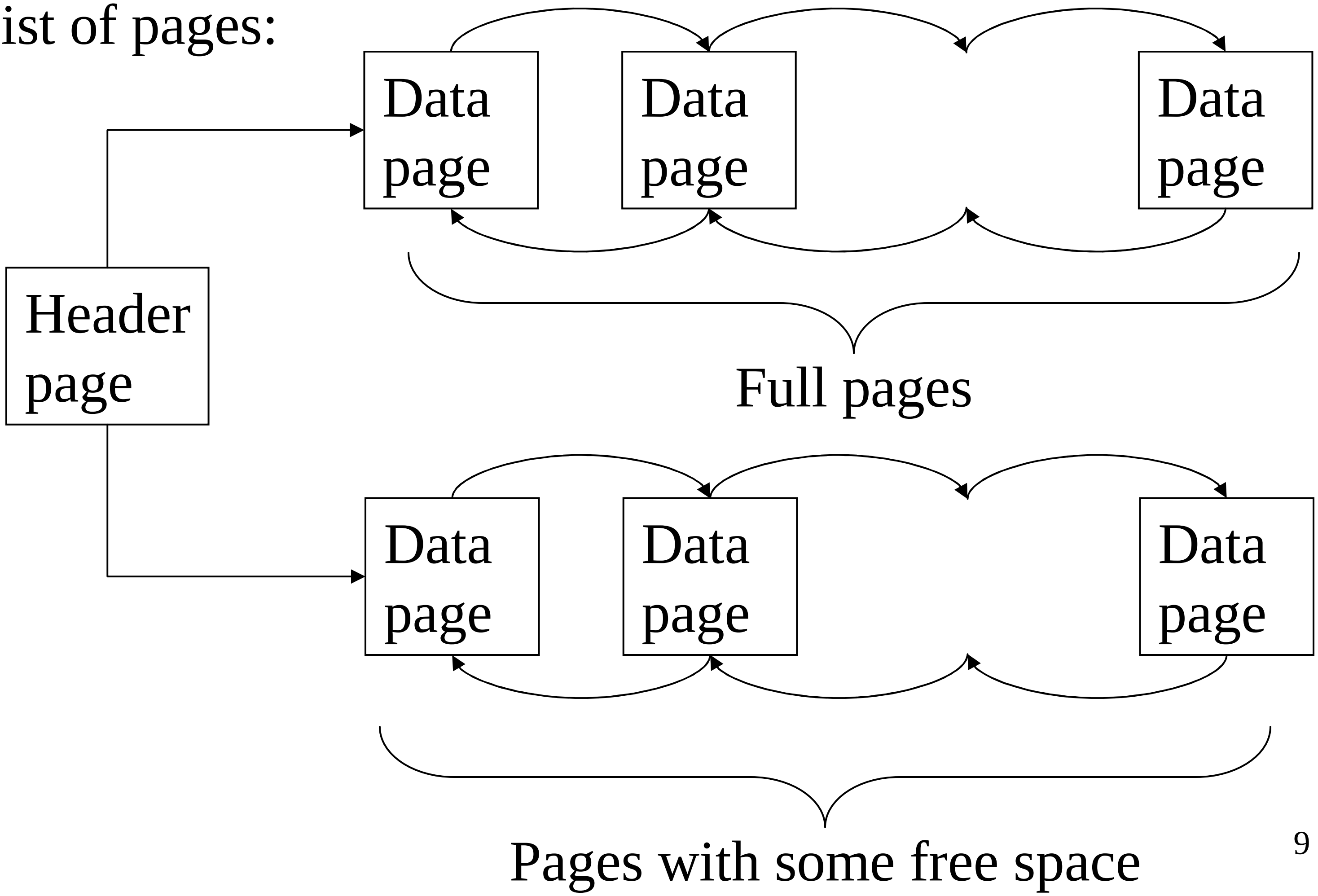
Managing Free Blocks

- By the OS

- By the RDBMS (typical: why ?)
  - Linked list of free blocks
  - Bit map

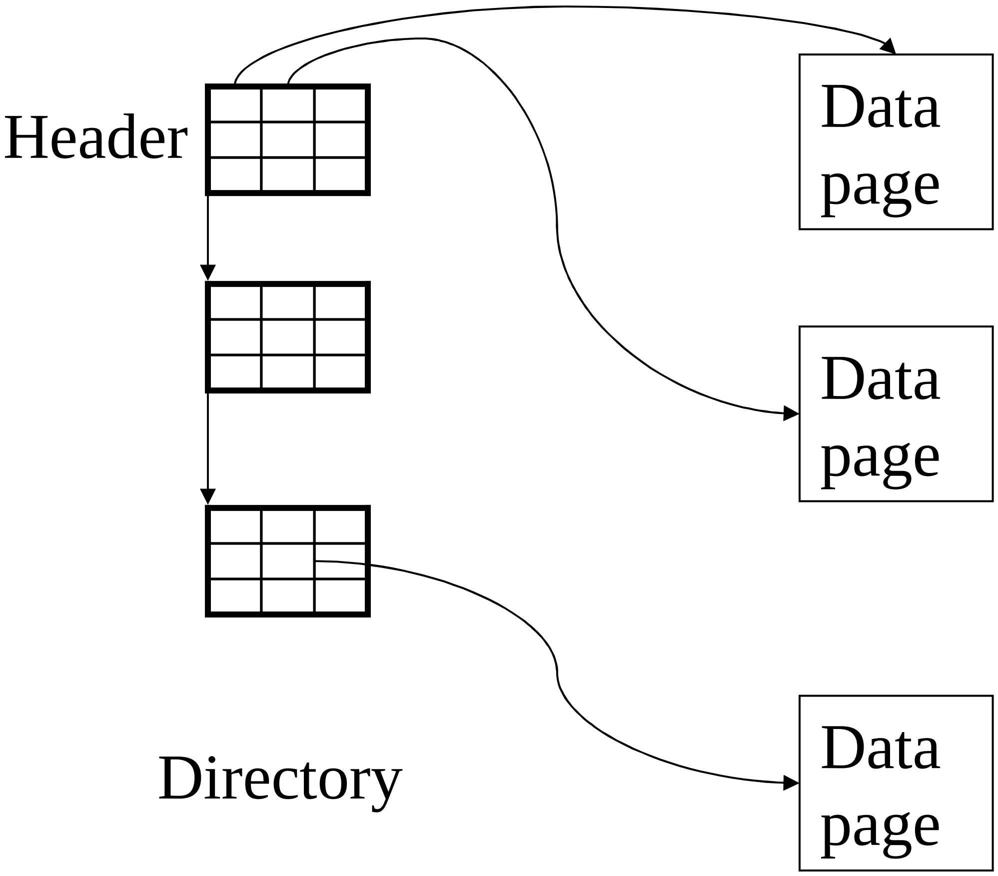
Managing Free Blocks

Linked list of pages:


Data page
Data page
Data page
Header page
Full pages
Data page
Data page
Data page
Pages with some free space

Managing Free Blocks

Better: directory of pages



Page Formats

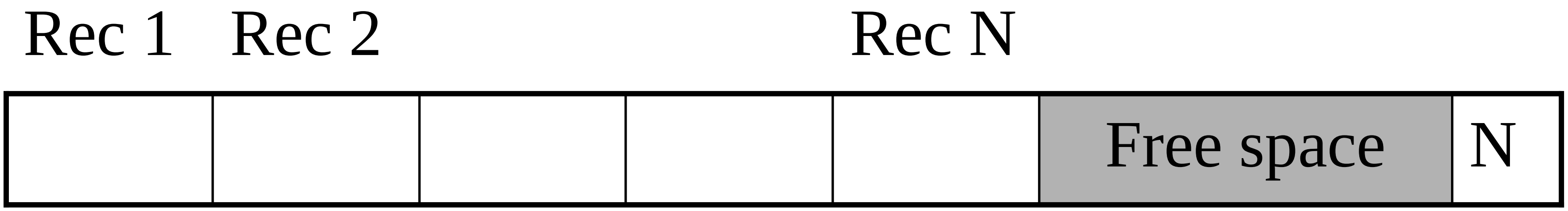
Issues to consider

- 1 page = fixed size (e.g. 8KB)
- Records:
  - Fixed length
  - Variable length
- Record id = RID
  - Typically RID = (PageID, SlotNumber)

Why do we need RID's in a relational DBMS ?

Page Formats

Fixed-length records: packed representation



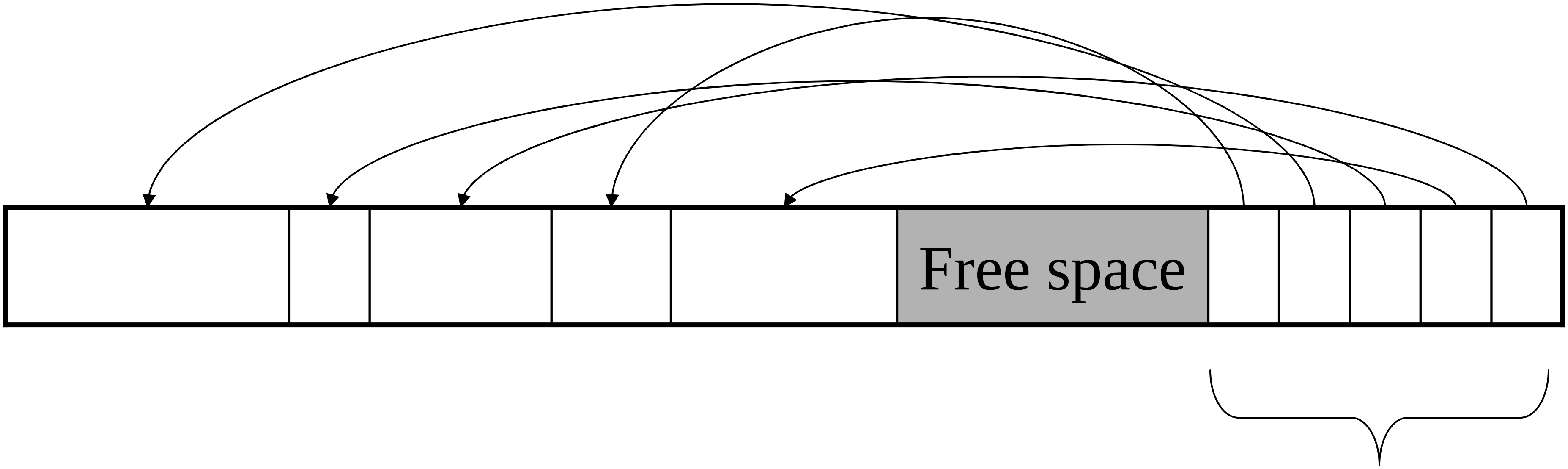
Rec 1 Rec 2 Rec N

Free space

N

Problems ?

Page Formats

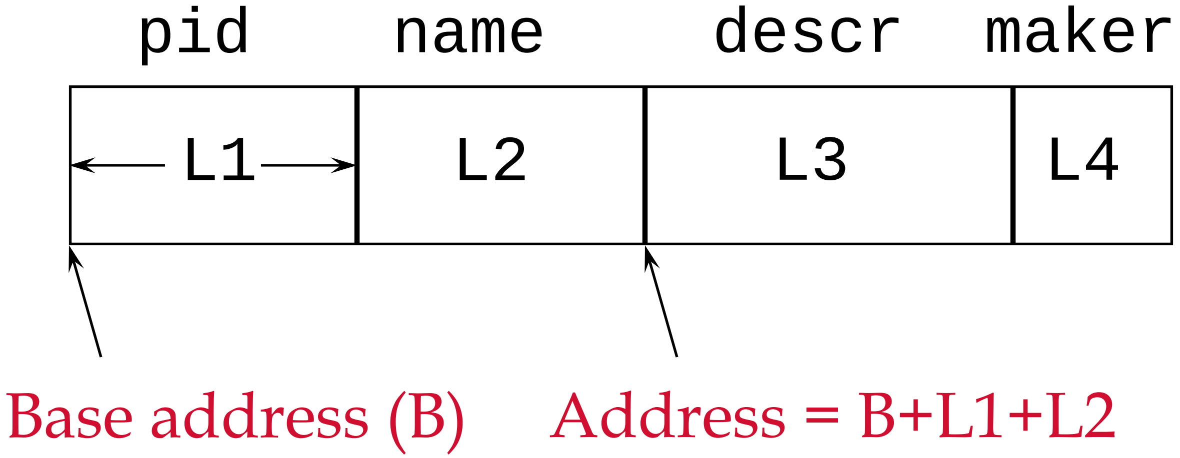


Slot directory

Variable-length records

Record Formats: Fixed Length

Product (pid, name, descr, maker)



pid name descr maker

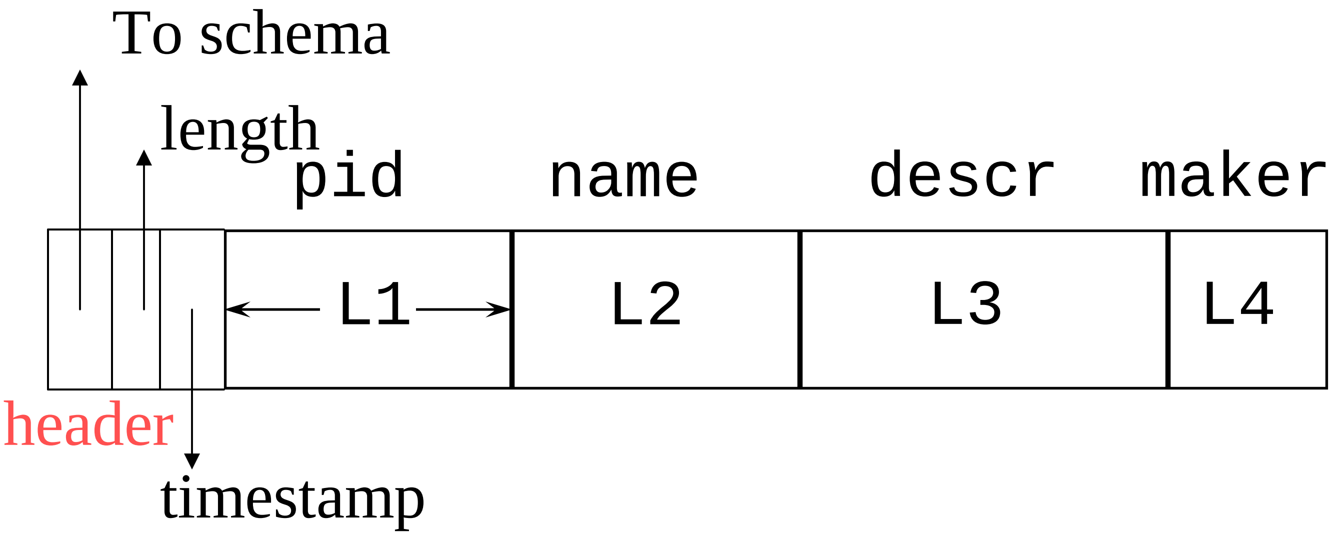
L1 L2 L3 L4

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 requires scan of record.
- **Note the importance of schema information!**

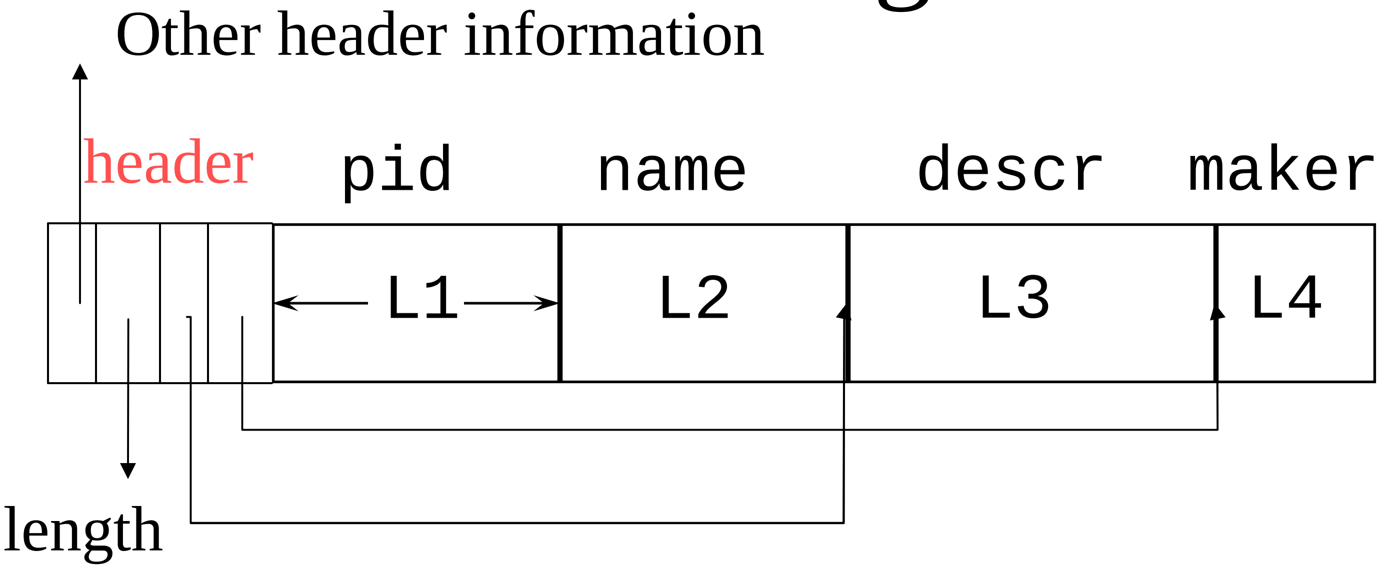
Record Header



Need the header because:

- The schema may change
  for a while new+old may coexist
- Records from different relations may coexist

Variable Length Records



Place the fixed fields first: F1

Then the variable length fields: F2, F3, F4

Null values take 2 bytes only

Sometimes they take 0 bytes (when at the end)

BLOB

- Binary large objects
- Supported by modern database systems
- E.g. images, sounds, etc.
- Storage: attempt to cluster blocks together

CLOB = character large object

- Supports only restricted operations

File Organizations

- **Heap** (random order) files: Suitable when typical access is a file scan retrieving all records.
- **Sorted** Files: Best if records must be retrieved in some order, or only a `range' of records is needed.
- **Indexes**: Data structures to organize records via trees or hashing.
  - Like sorted files, they speed up searches for a subset of records, based on values in certain ("search key") fields
  - Updates are much faster than in sorted files.

Modifications: Insertion

- File is unsorted: add it to the end (easy ☺)
- File is sorted:
  - Is there space in the right block ?
    - Yes: we are lucky, store it there
  - Is there space in a neighboring block ?
    - Look 1-2 blocks to the left/right, shift records
  - If anything else fails, create *overflow block*

Modifications: Deletions

- Free space in block, shift records
- Maybe be able to eliminate an overflow block
- Can never really eliminate the record, because others may *point* to it
    - Place a tombstone instead (a NULL record)

How can we *point* to a record in an RDBMS ?

Modifications: Updates

- If new record is shorter than previous, easy ☺
- If it is longer, need to shift records, create overflow blocks

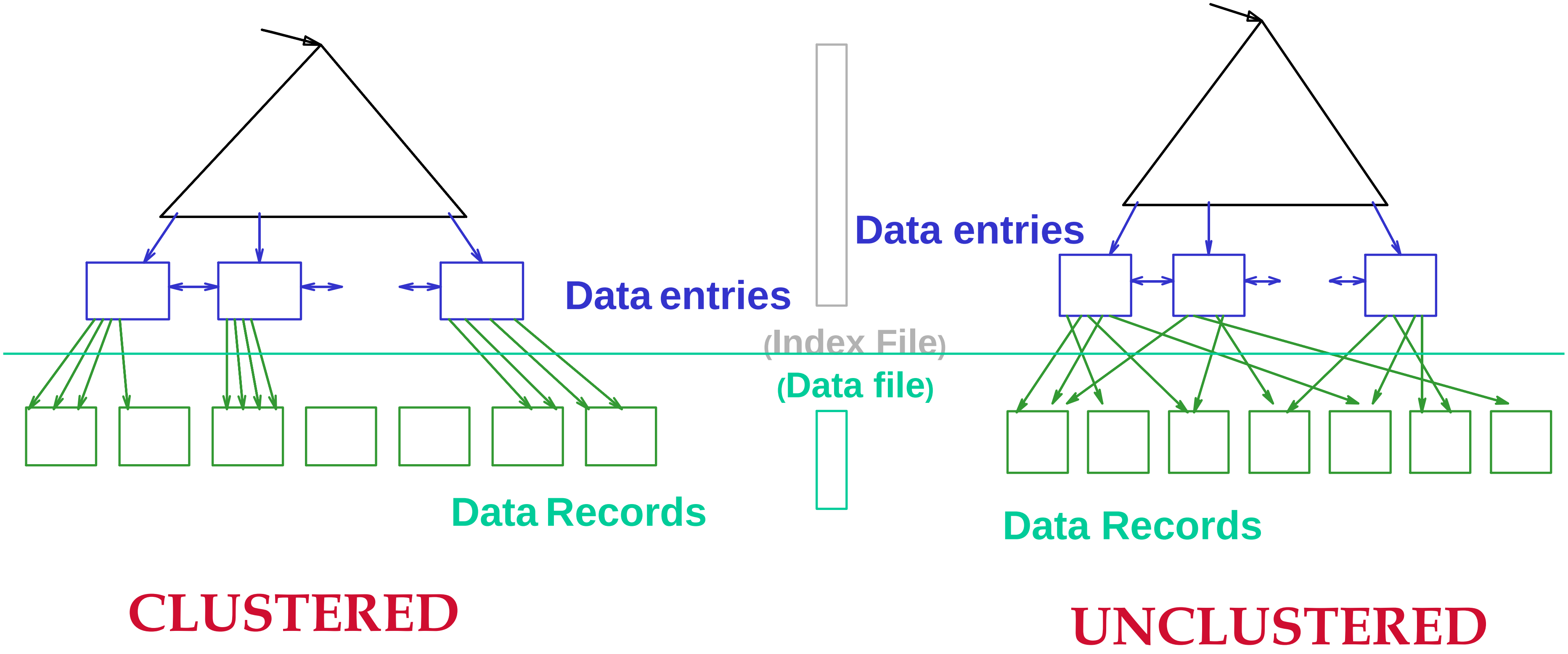
Indexes

- An *index* on a file speeds up selections on the *search key fields* for the index.
  - Any subset of the fields of a relation can be the search key for an index on the relation.
  - *Search key* is not the same as *key* (minimal set of fields that uniquely identify a record in a relation).
- An index contains a collection of *data entries*, and supports efficient retrieval of all data entries **k*** with a given key value **k**.

Index Classification

- Clustered/unclustered
  - Clustered = records close in the index are close in the data
  - Unclustered = records close in the index may be far in the data
- Primary/secondary
  - Sometimes means this:
    - Primary = includes primary key
    - Secondary = otherwise
  - Sometimes means clustered/unclustered
- Dense/sparse
  - Dense = every key in the data appears in the index
  - Sparse = the index contains only some keys
- B+ tree / Hash table / …

Clustered vs. Unclustered Index



Data entries

Data entries

(Index File)

(Data file)

Data Records

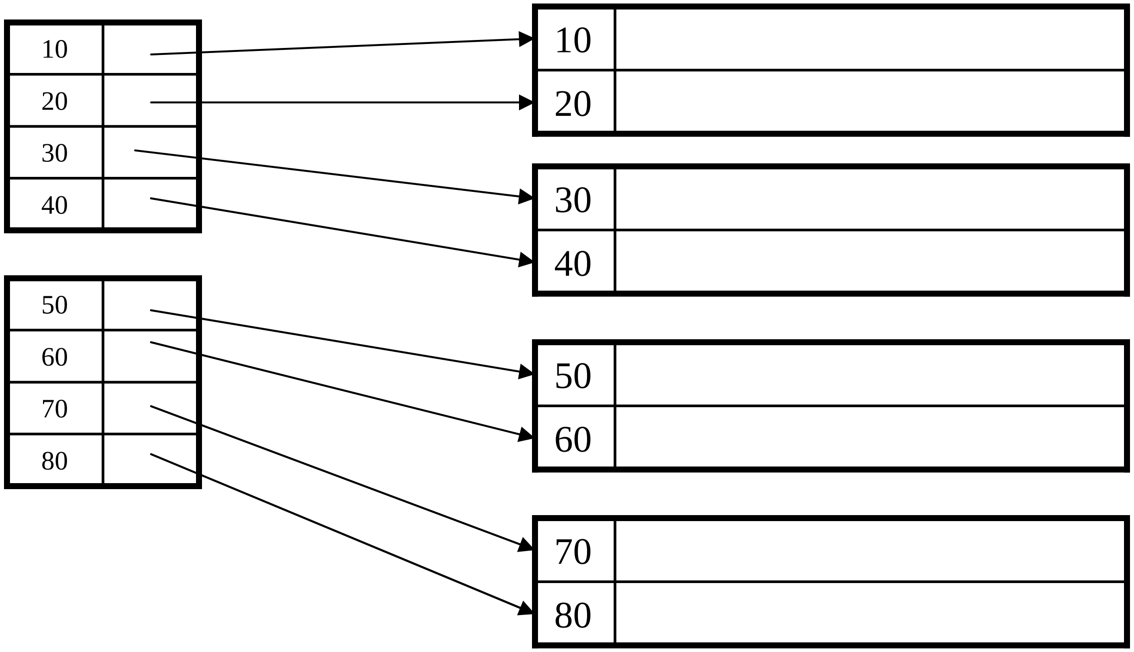
Data Records

CLUSTERED

UNCLUSTERED

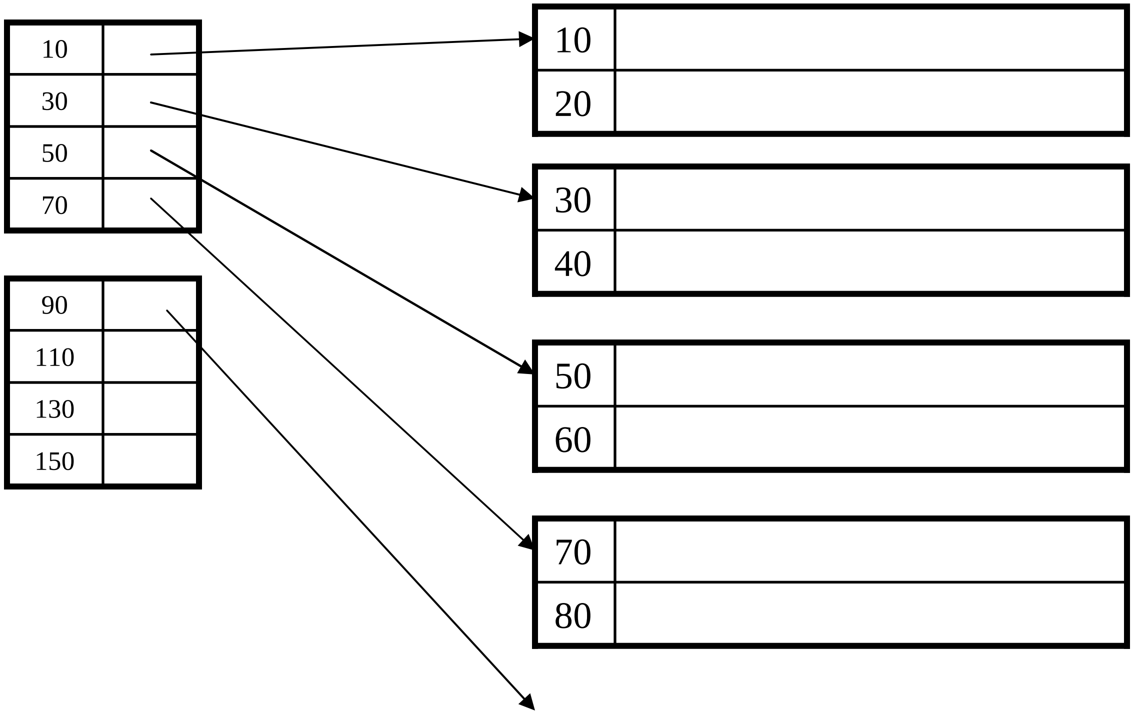
Clustered Index

- File is sorted on the index attribute
- ***Dense*** index: sequence of (key,pointer) pairs



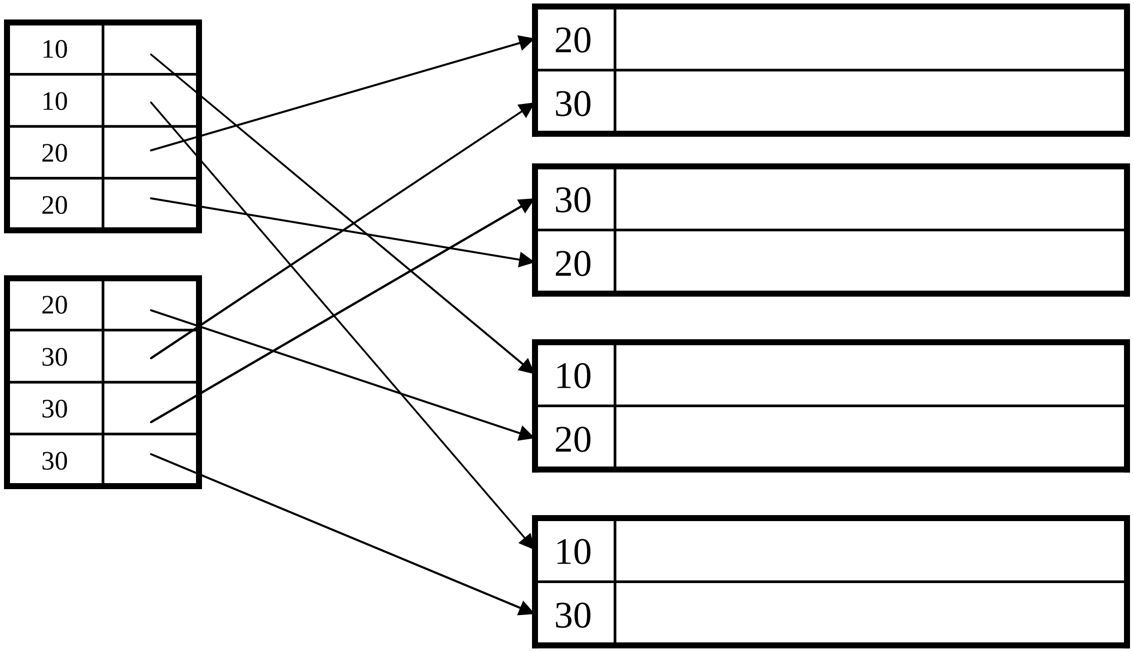
Clustered Index

- *Sparse* index



Unclustered Indexes

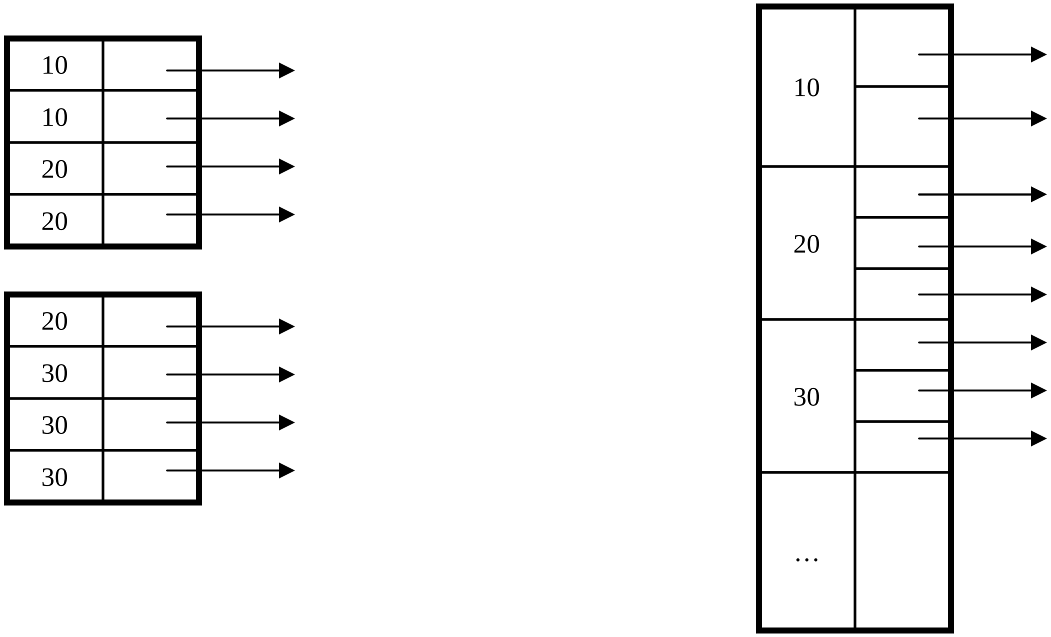
- To index other attributes than primary key
- Always dense (why ?)



Alternatives for Data Entry k* in Index

- Three alternatives for **k***:
  - – Data record with key value **k**
  - – <**k**, **rid** of data record with key = **k**>
  - – <**k**, list of **rids** of data records with key = **k**>
- Last two choices are orthogonal to the indexing technique used to locate data entries with a given key value k.

Alternatives 2 and 3



Using an Index

- The **scan** operation:
  - Read index entries in order

- Clustered index:
  - Index scan = Table scan

- Unclustered index:
  - Scan much more expensive

Using an Index

- Exact key values:
  - Scan index, lookup relation
  - B+ trees or hash tables

```
Select name
From people
Where salary = 25
```

- Range queries:
  - B+ trees

```
Select name
From people
Where 20 <= age and  age <= 30
```

- Use index exclusively

```
Select distinct age
From people
```

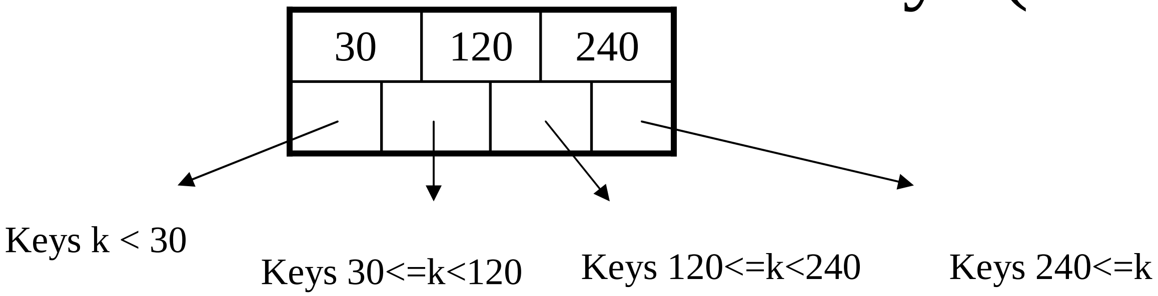
DEMO (see notes)

B+ Trees

- Search trees
- Idea in B Trees:
  - make 1 node = 1 block
- Idea in B+ Trees:
  - Make leaves into a linked list (range queries are easier)

B+ Trees Basics

- Parameter d = the *degree*
- Each node has >= d and <= 2d keys (except root)



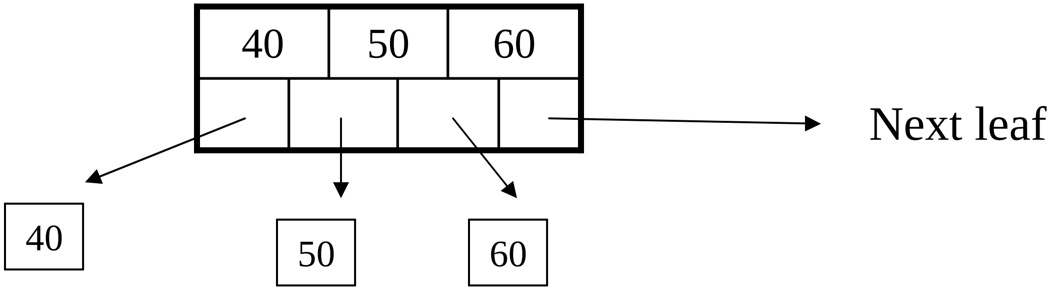
Keys k < 30

Keys 30<=k<120

Keys 120<=k<240

Keys 240<=k

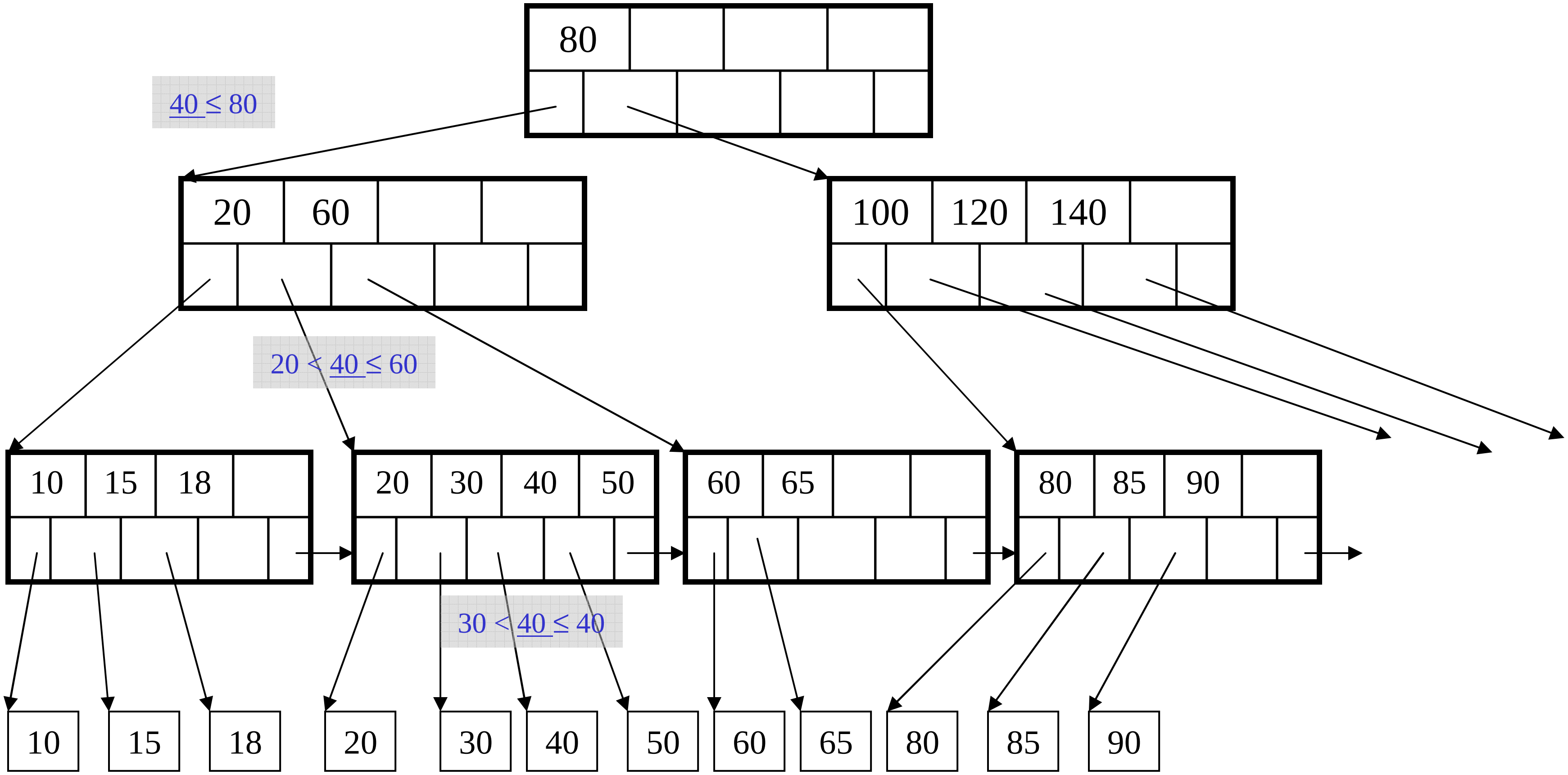
- Each leaf has >=d and <= 2d keys:



Next leaf

B+ Tree Example

d = 2



Find the key $\underline{40}$

$\underline{40} \leq 80$

$20 < \underline{40} \leq 60$

$30 < \underline{40} \leq 40$

B+ Tree Design

- How large d ?
- Example:
  - Key size = 4 bytes
  - Pointer size = 8 bytes
  - Block size = 4096 byes
- 2d x 4 + (2d+1) x 8 <= 4096
- d = 170

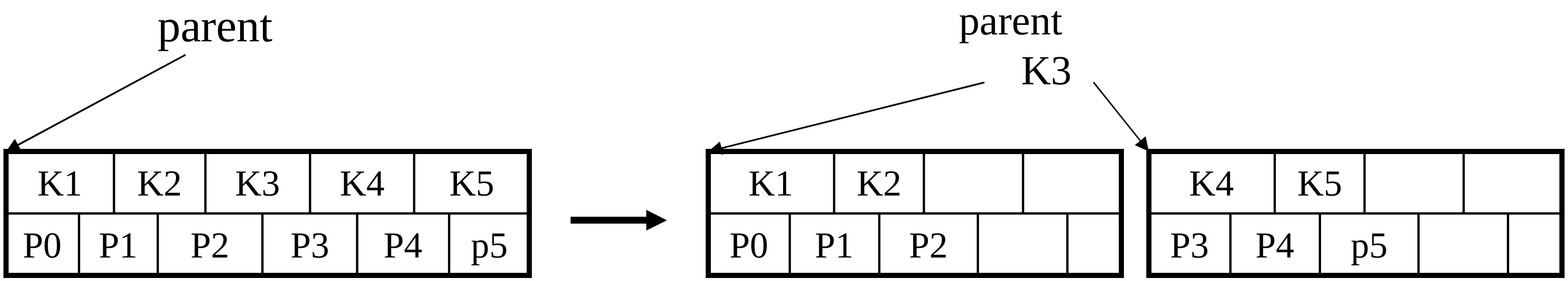
B+ Trees in Practice

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

Insertion in a B+ Tree

Insert (K, P)

- Find leaf where K belongs, insert
- If no overflow (2d keys or less), halt
- If overflow (2d+1 keys), split node, insert in parent:



parent

K1	K2	K3	K4	K5	
P0	P1	P2	P3	P4	p5

→

parent
K3

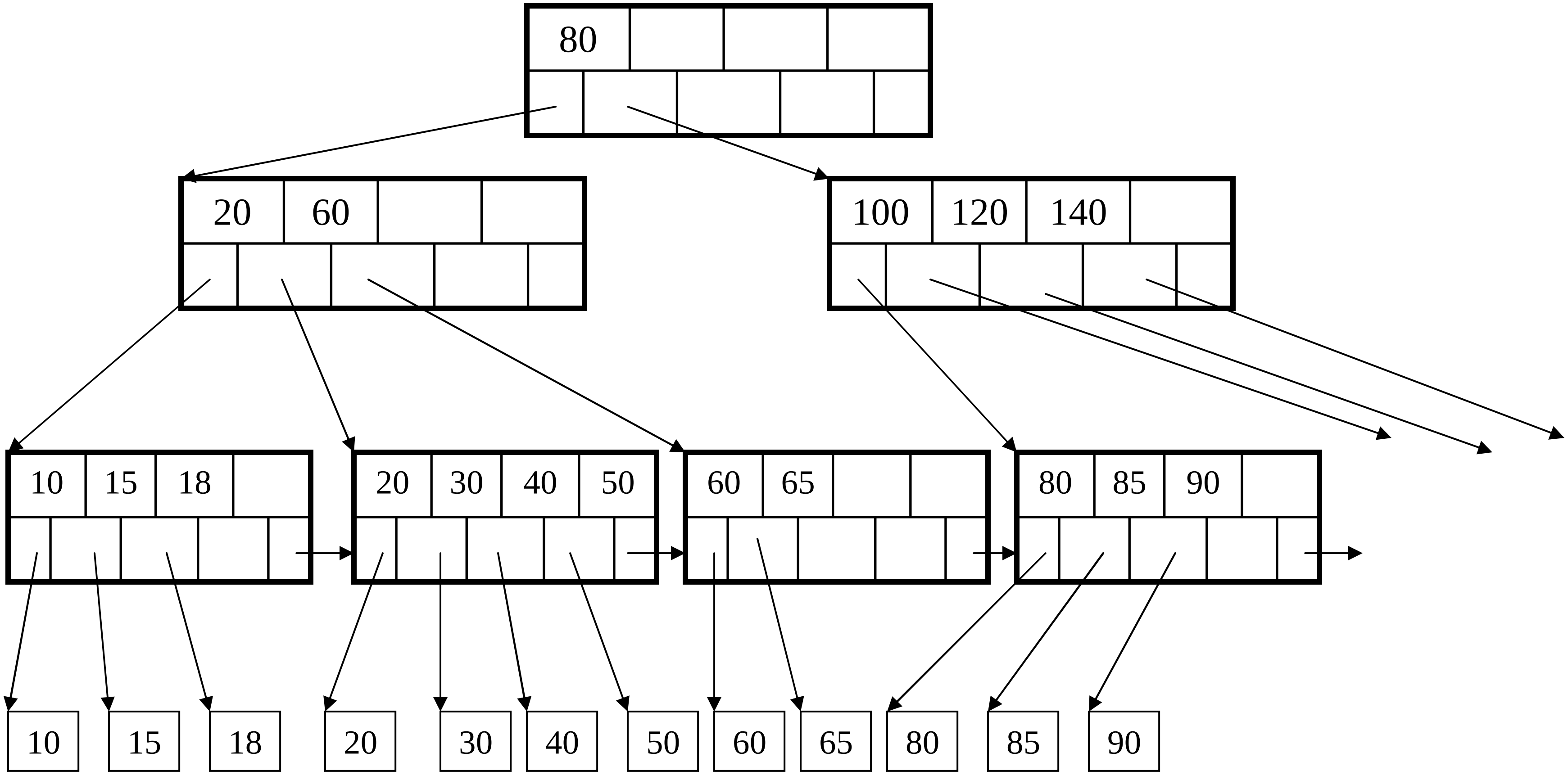
K1	K2			
P0	P1	P2		

K4	K5			
P3	P4	p5		

- If leaf, keep K3 too in right node
- When root splits, new root has 1 key only

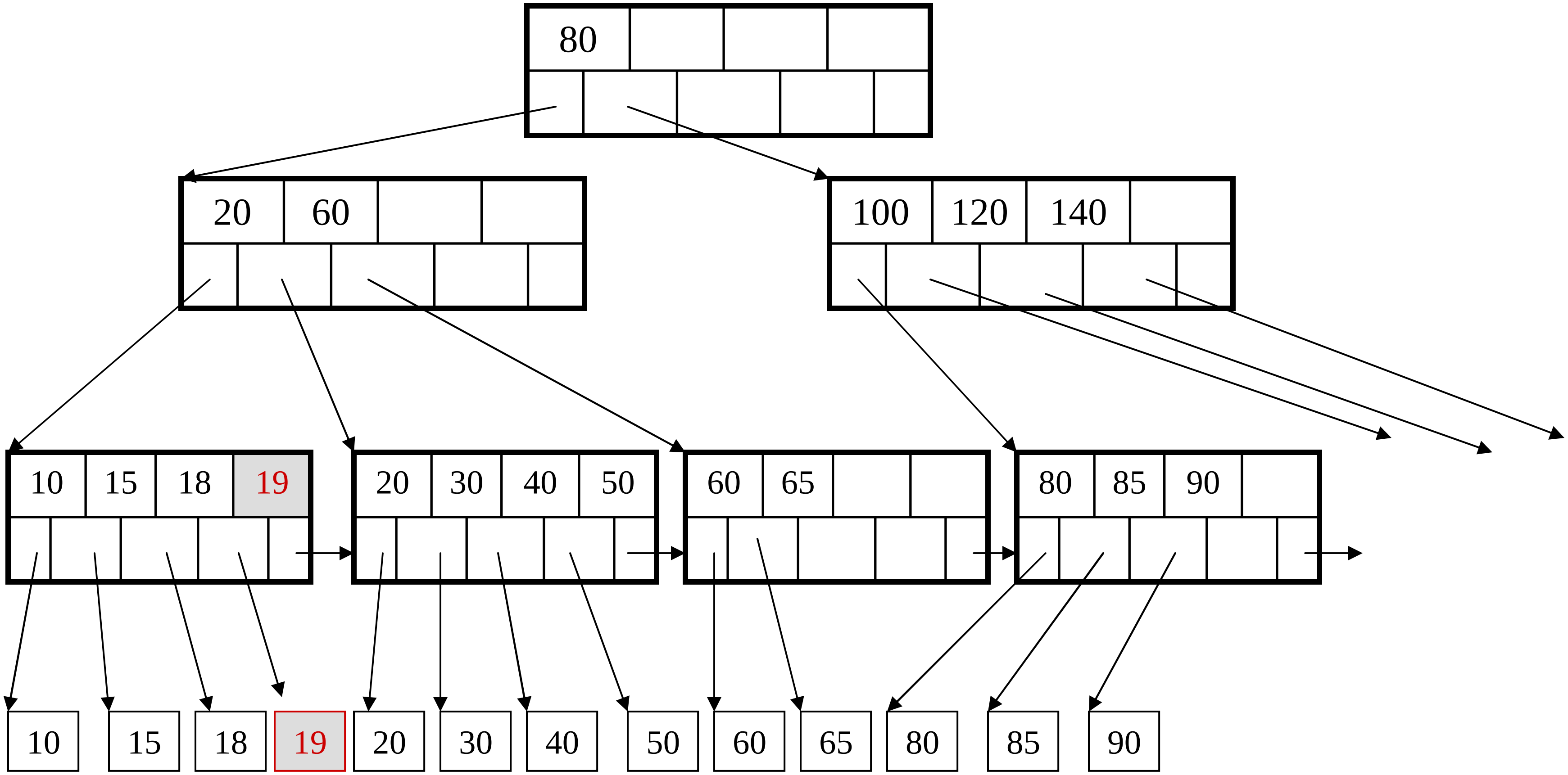
Insertion in a B+ Tree

Insert K=19



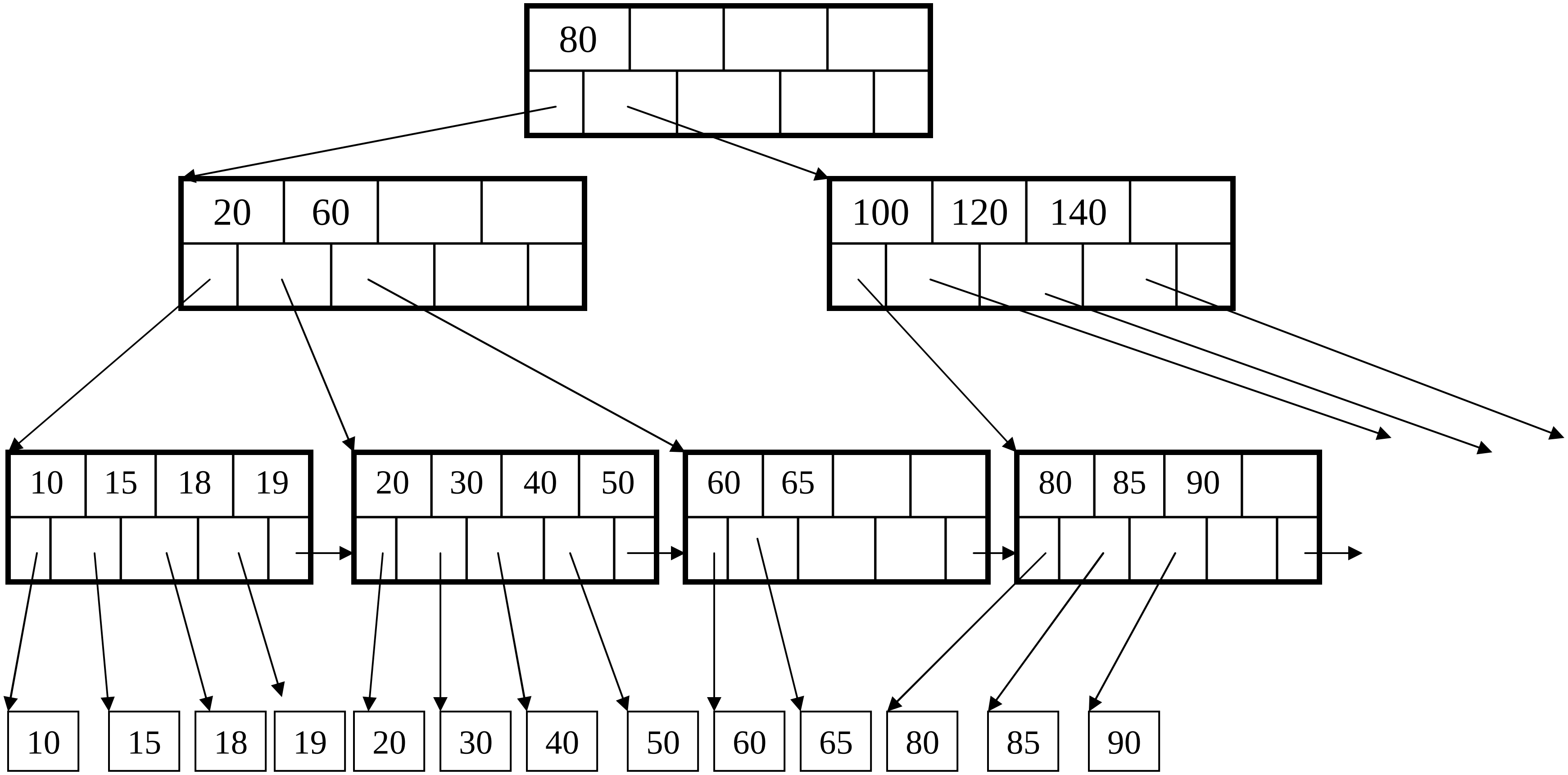
Insertion in a B+ Tree

After insertion



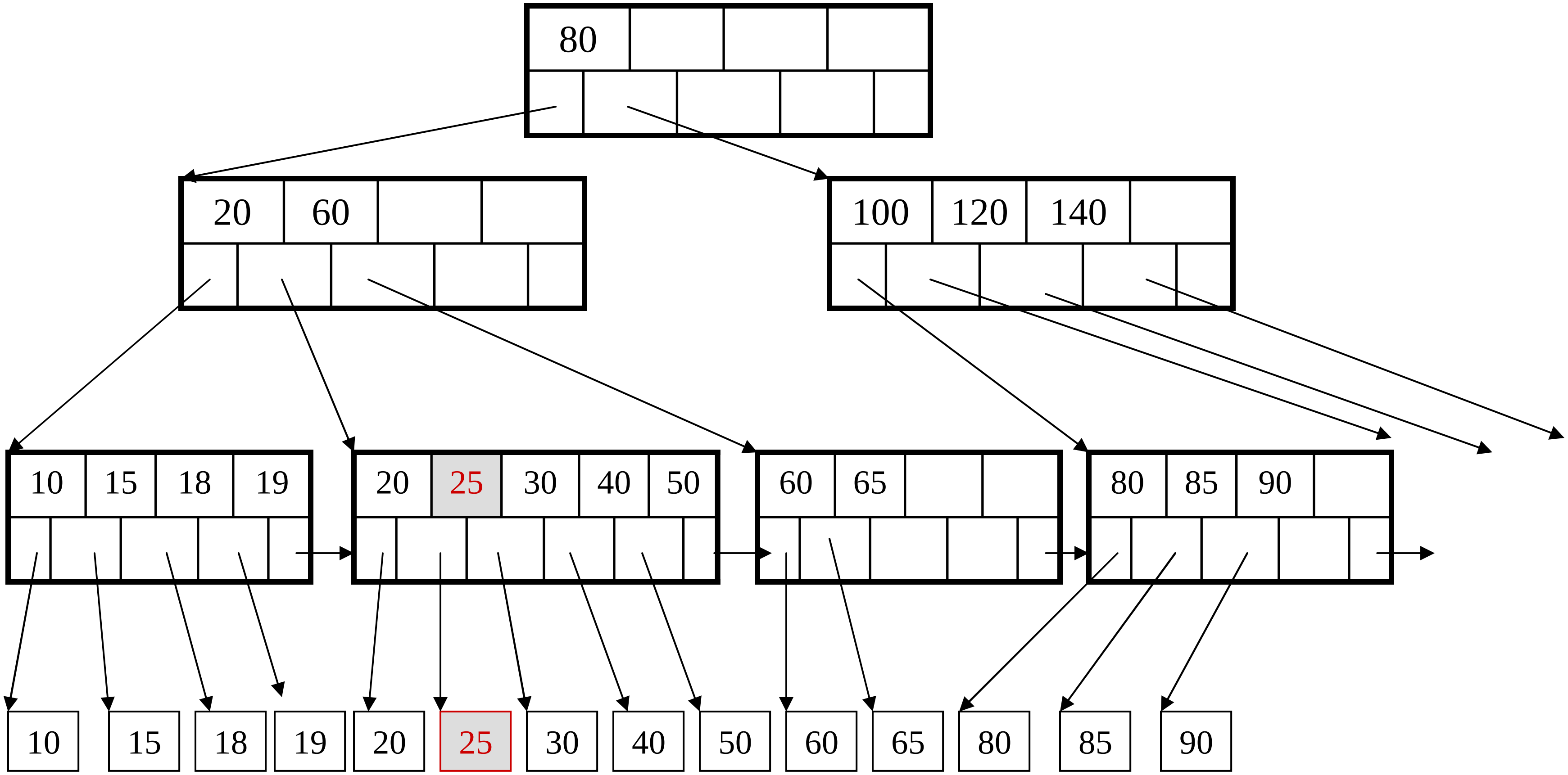
Insertion in a B+ Tree

Now insert 25



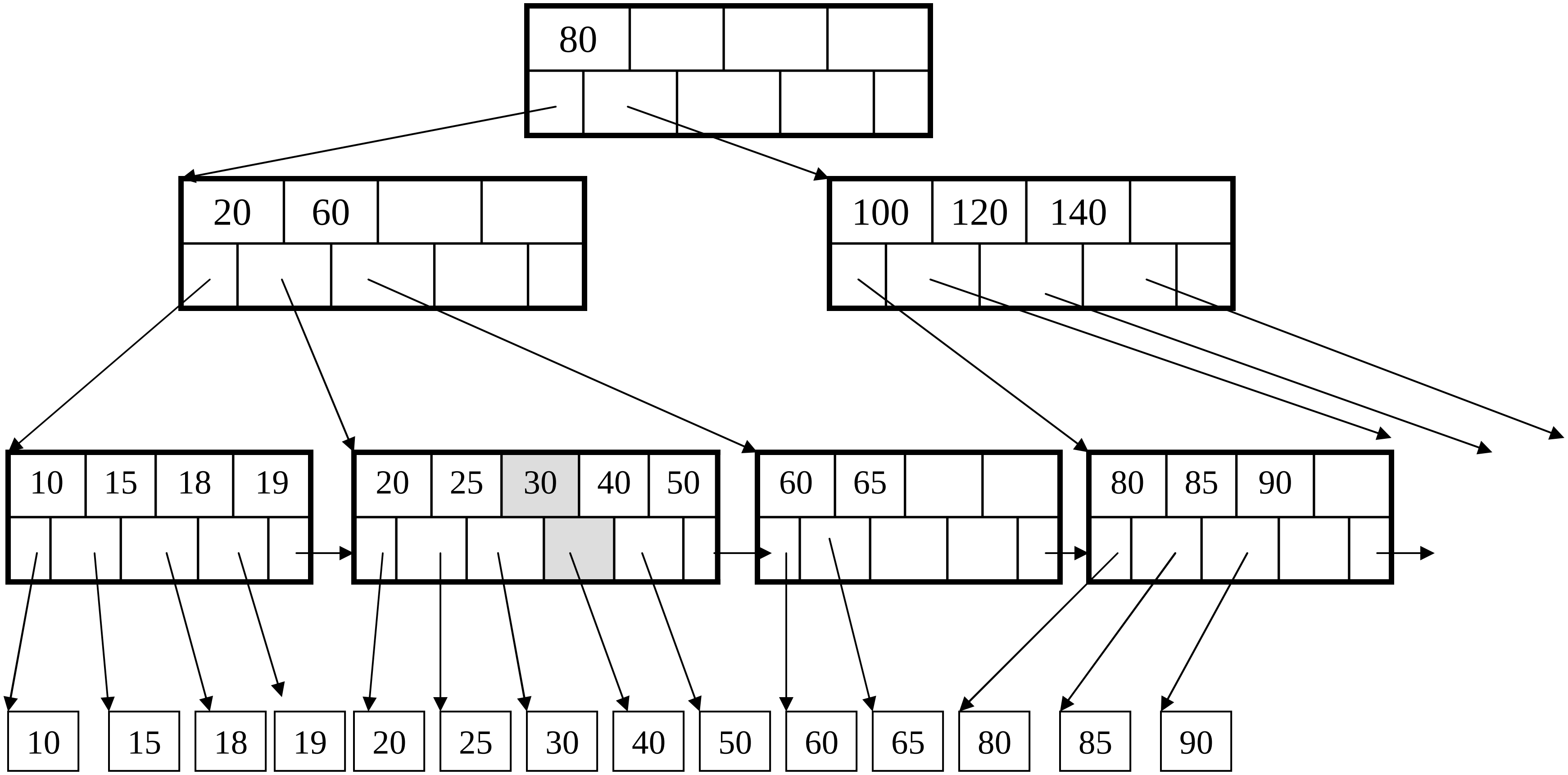
Insertion in a B+ Tree

After insertion



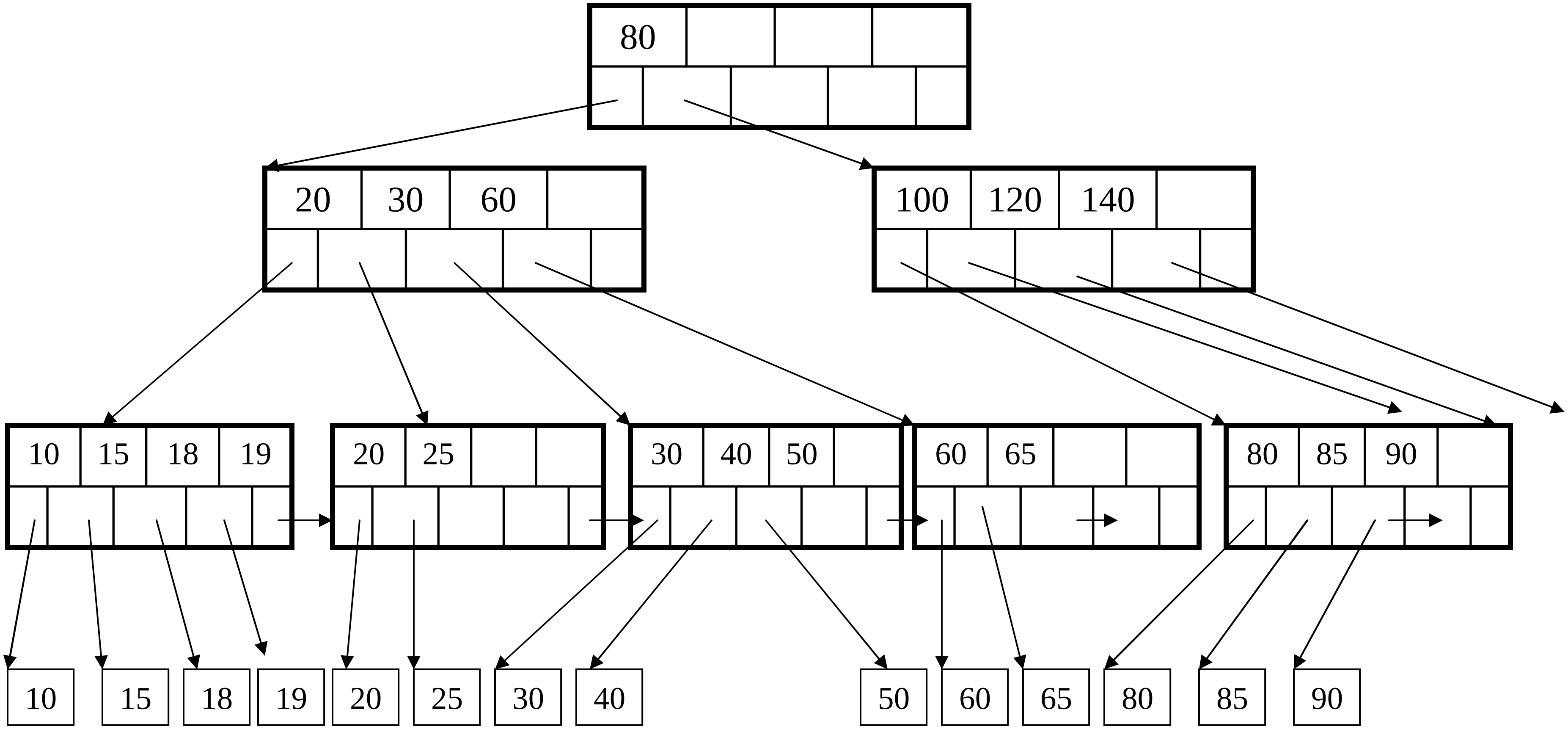
Insertion in a B+ Tree

But now have to split !



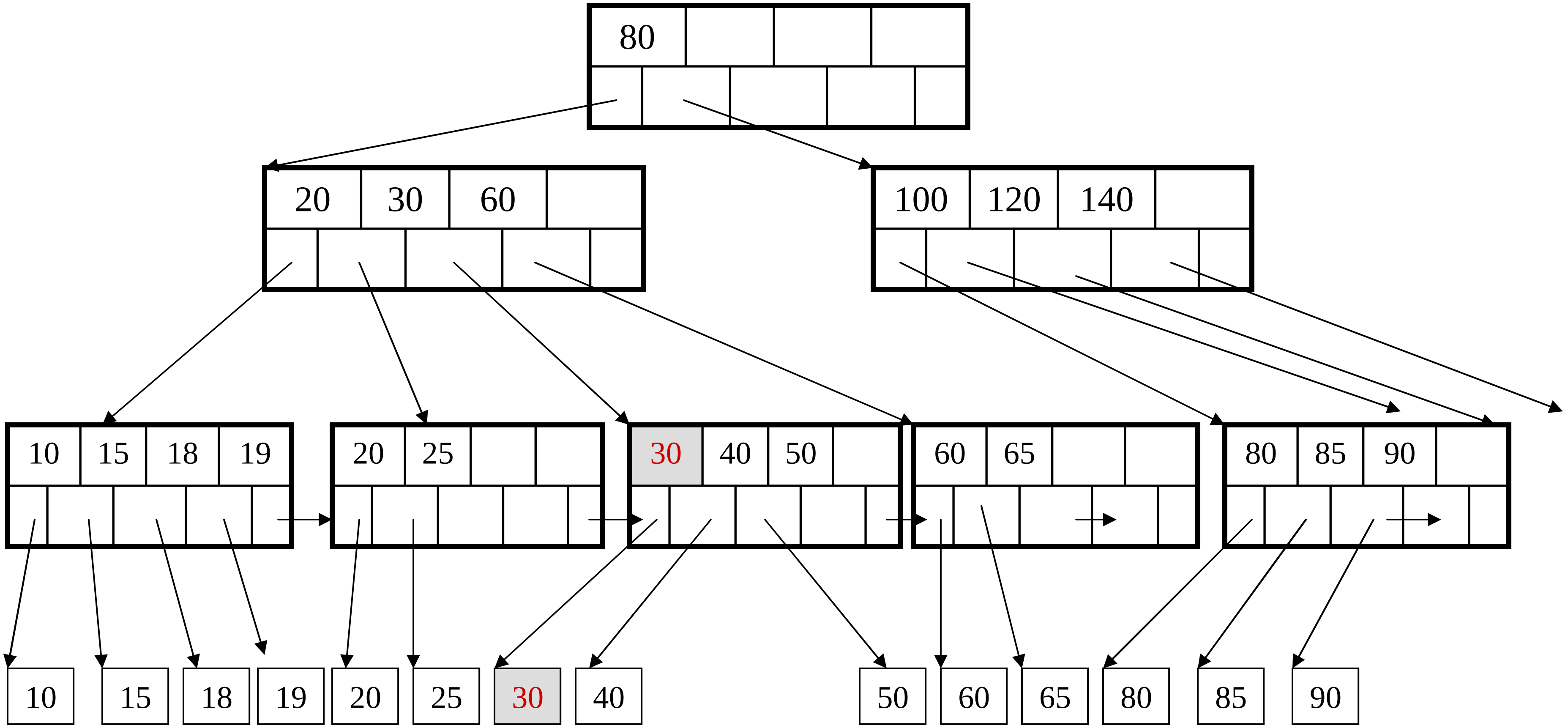
Insertion in a B+ Tree

After the split



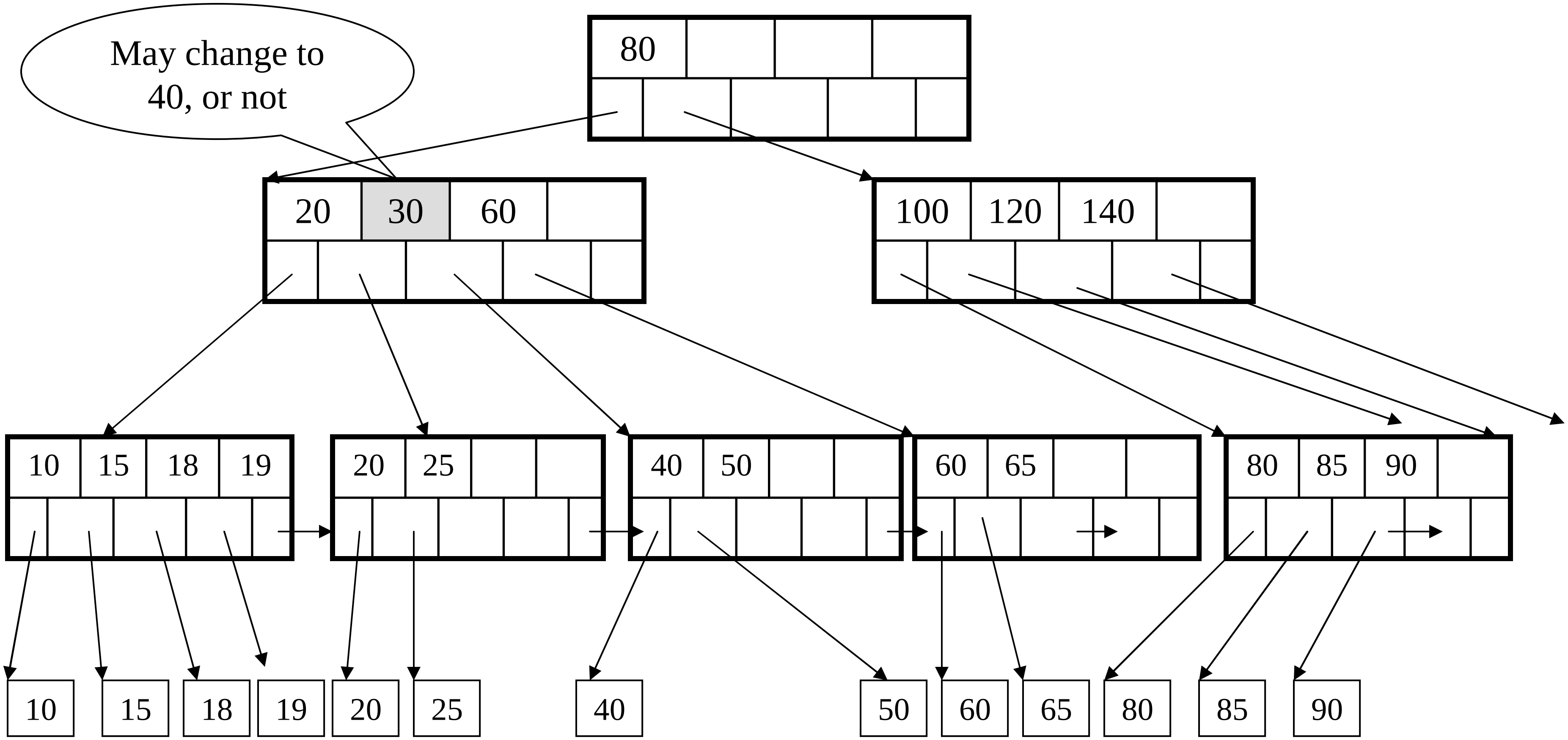
Deletion from a B+ Tree

Delete 30



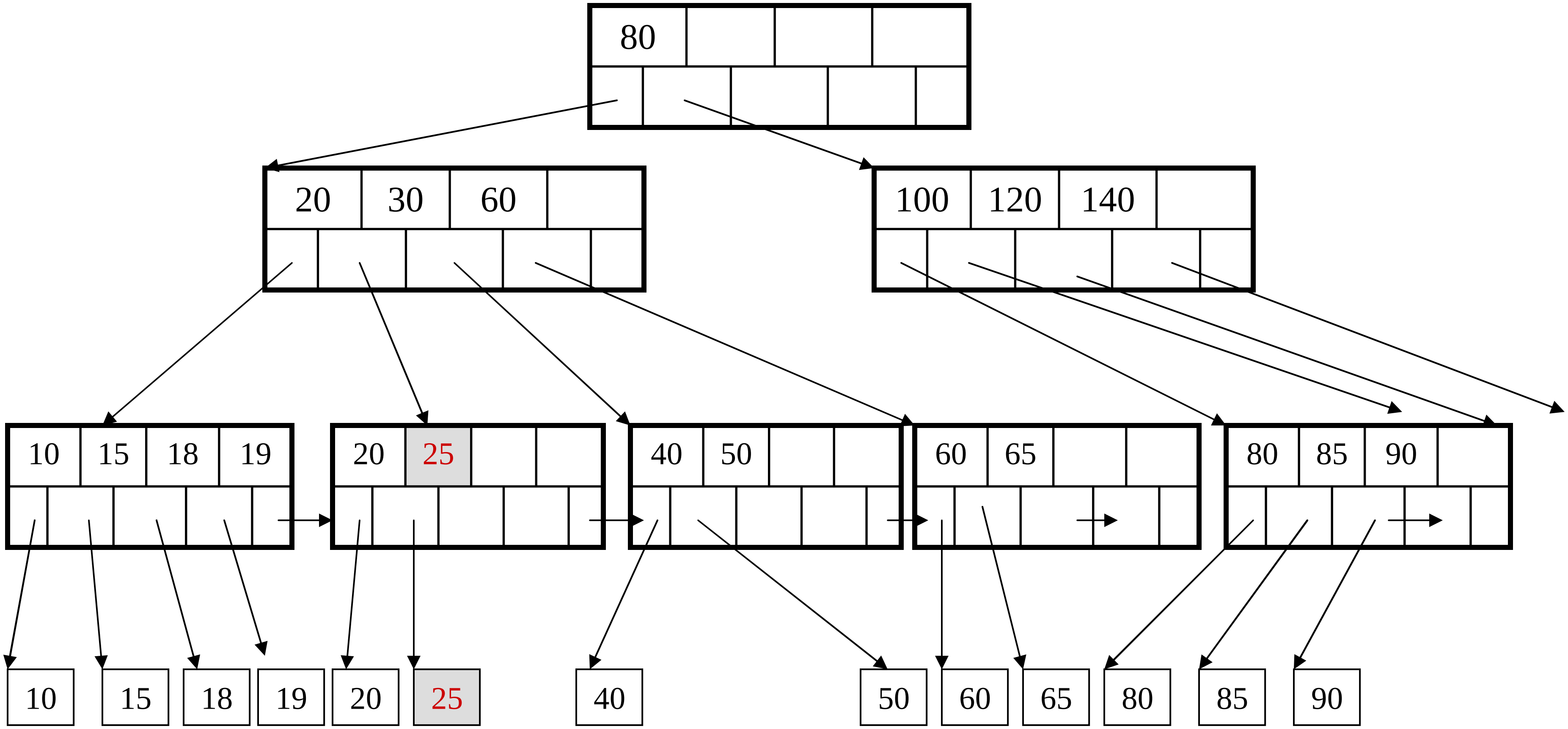
Deletion from a B+ Tree

After deleting 30



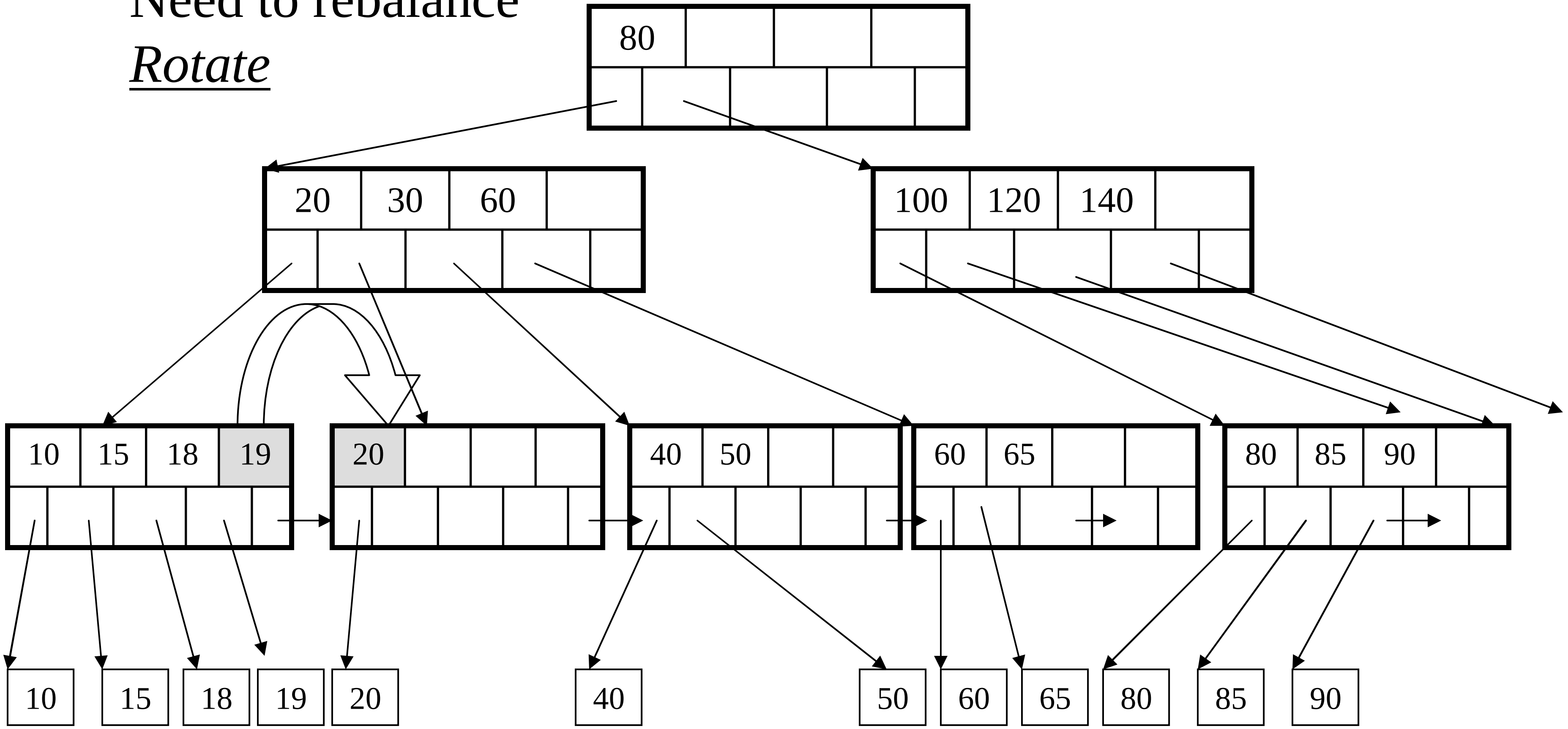
Deletion from a B+ Tree

Now delete 25



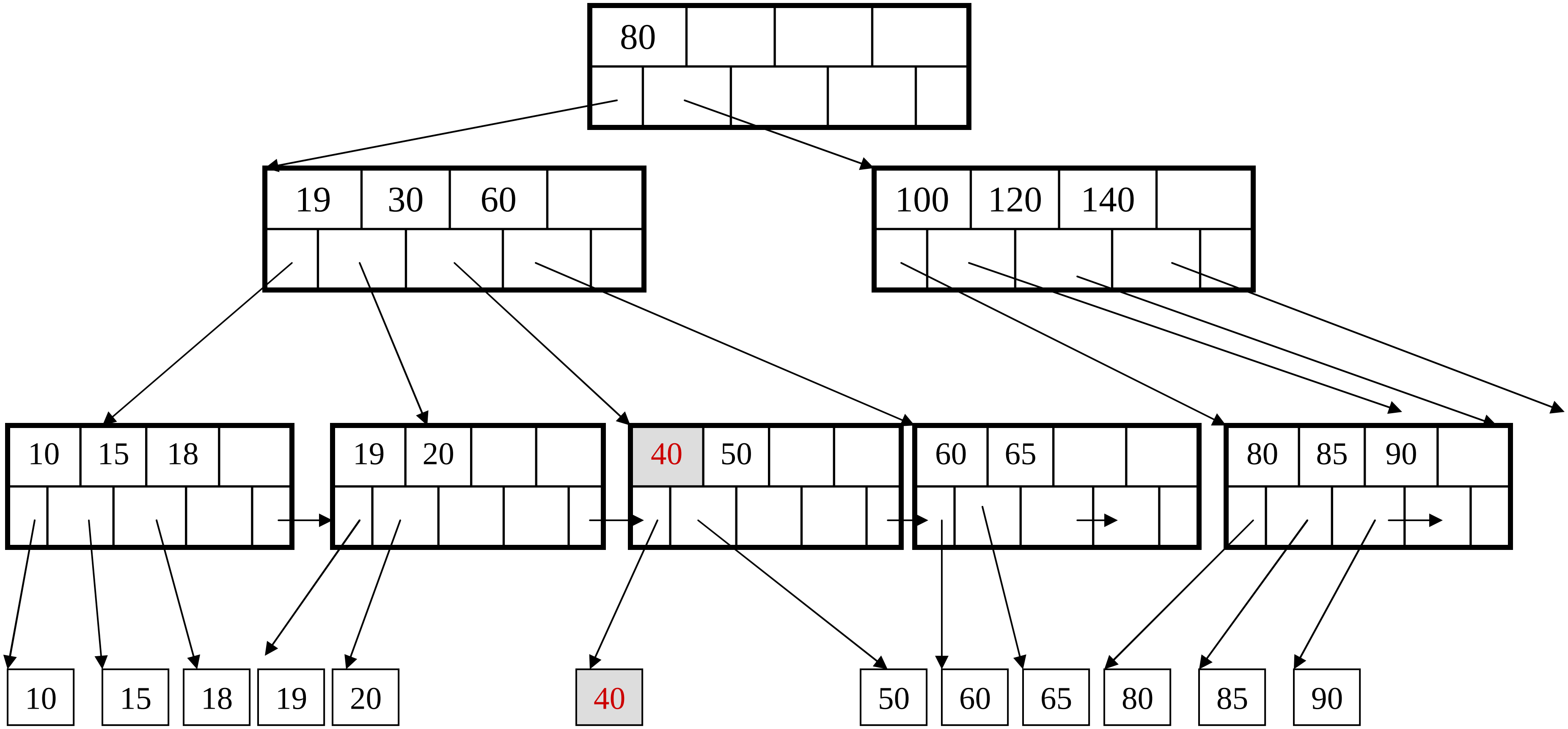
Deletion from a B+ Tree

After deleting 25
Need to rebalance
Rotate



Deletion from a B+ Tree

Now delete 40

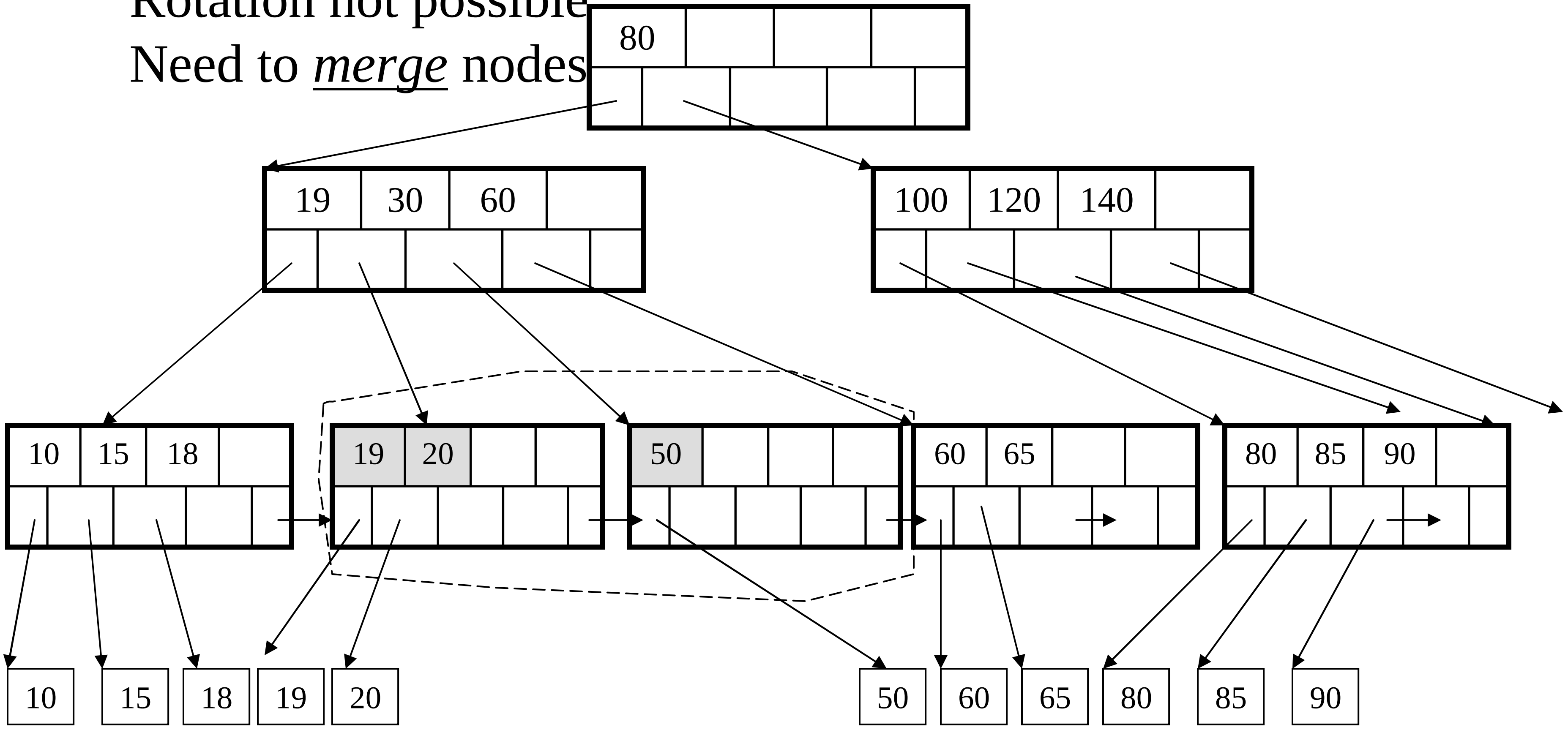


Deletion from a B+ Tree

After deleting 40

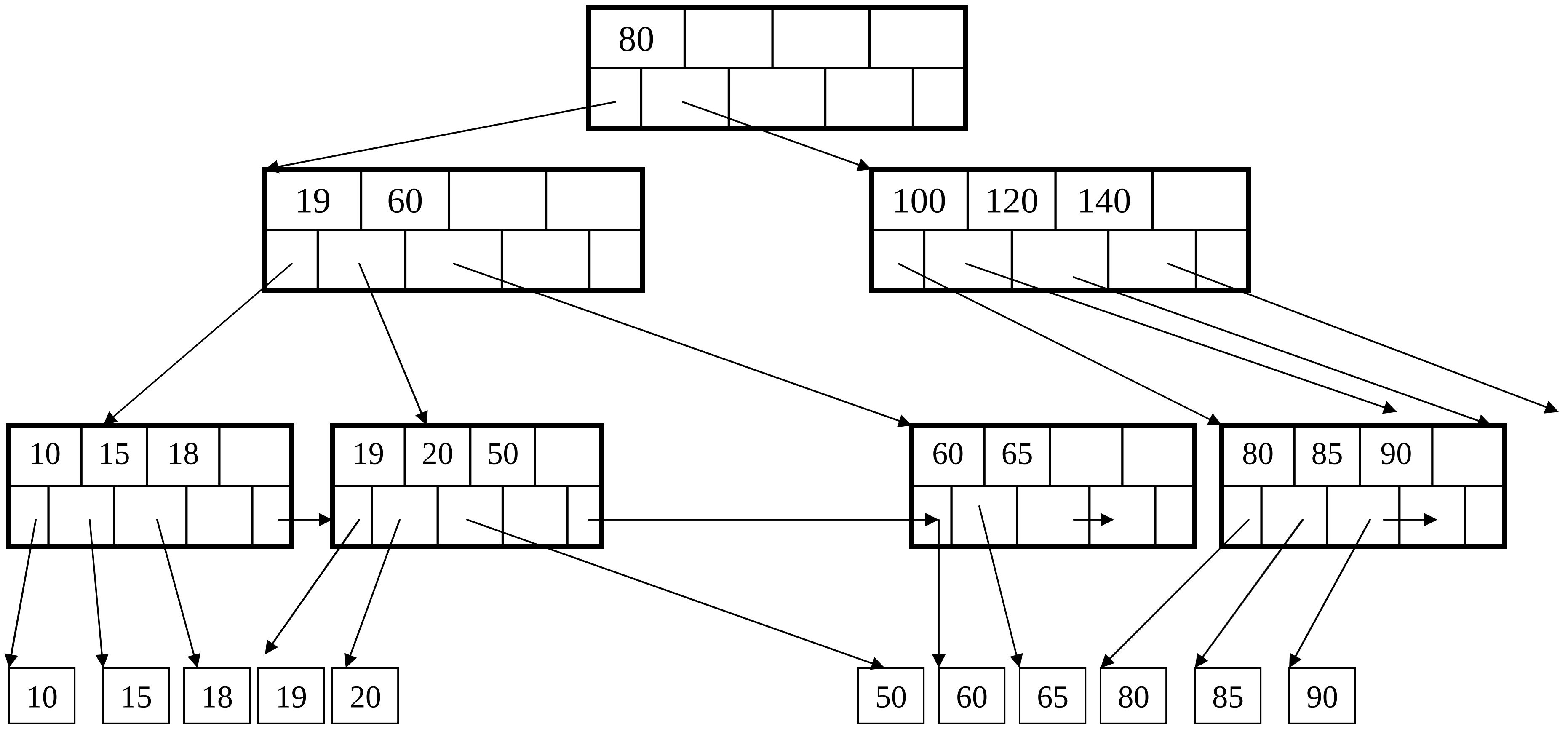
Rotation not possible

Need to *merge* nodes



Deletion from a B+ Tree

Final tree



Summary on B+ Trees

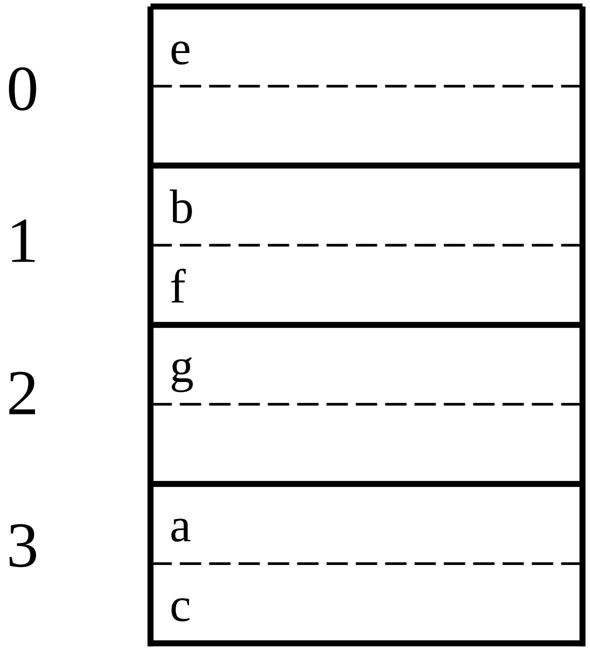
- Default index structure on most DBMS
- Very effective at answering ‘point’ queries:
  productName = ‘gizmo’
- Effective for range queries:
  50 < price AND price < 100
- Less effective for multirange:
  50 < price < 100 AND 2 < quant < 20

Hash Tables

- Secondary storage hash tables are much like main memory ones
- Recall basics:
  - There are n *buckets*
  - A hash function f(k) maps a key k to {0, 1, …, n-1}
  - Store in bucket f(k) a pointer to record with key k
- Secondary storage: bucket = block, use overflow blocks when needed

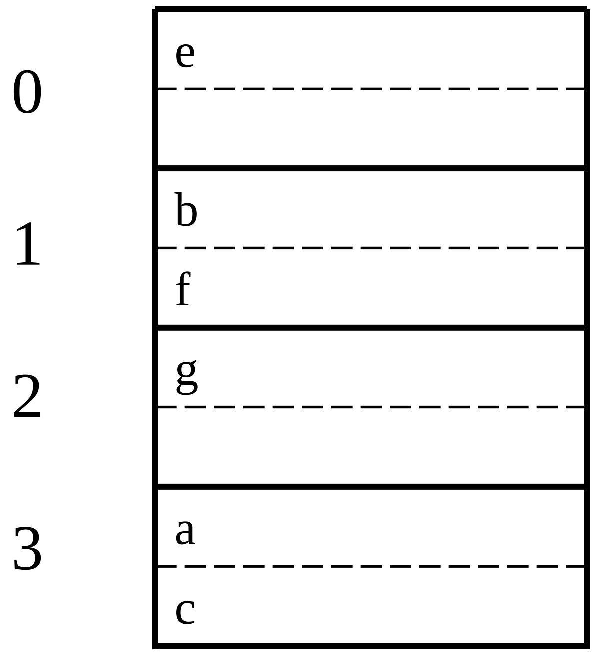
Hash Table Example

- Assume 1 bucket (block) stores 2 keys + pointers
- h(e)=0
- h(b)=h(f)=1
- h(g)=2
- h(a)=h(c)=3



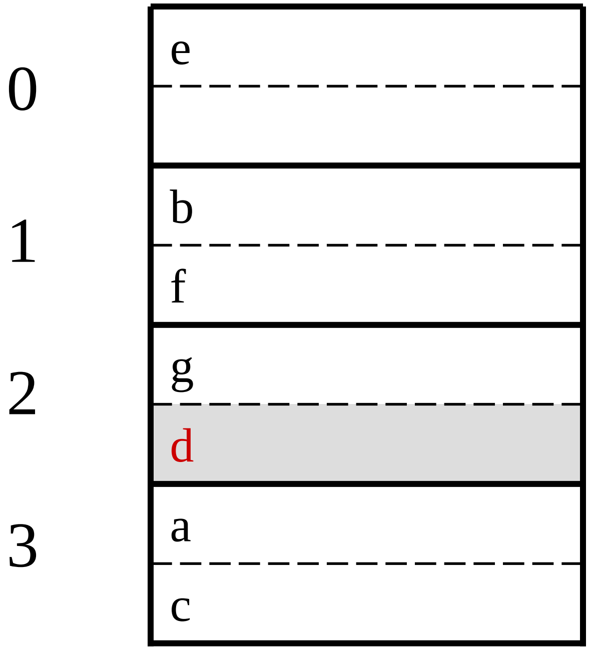
Searching in a Hash Table

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



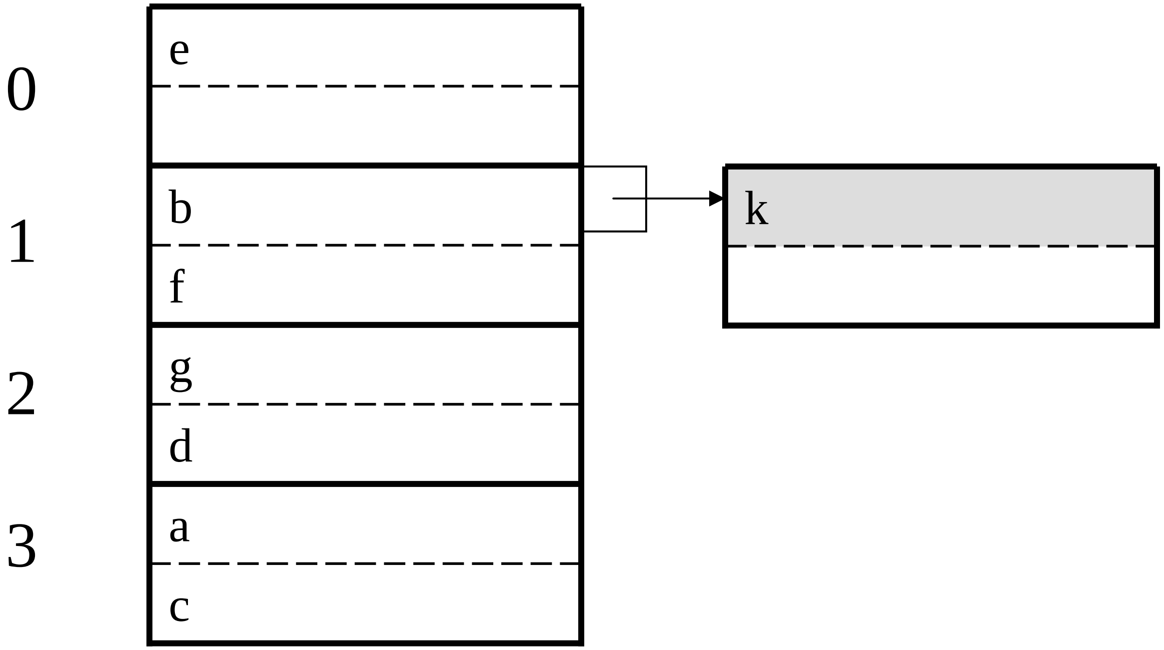
Insertion in Hash Table

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



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 considerably when number of keys exceeds the number of buckets (I.e. many overflow blocks).

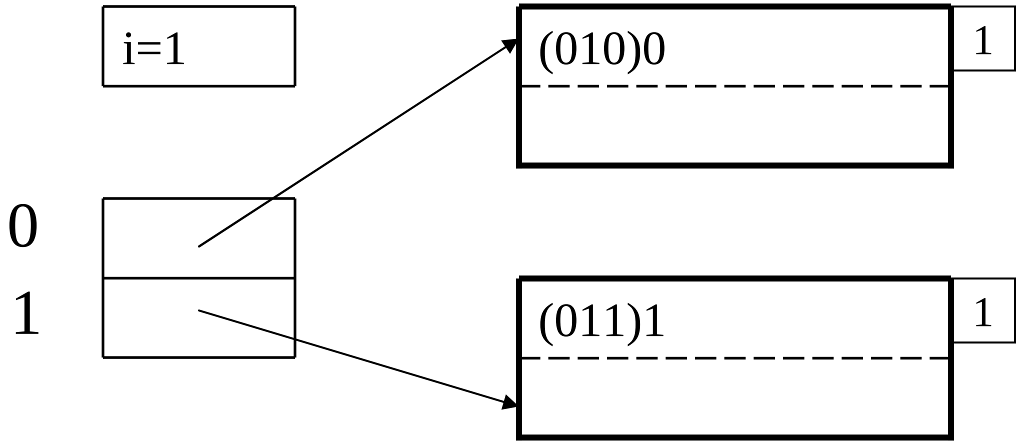
Extensible Hash Table

- Allows has table to grow, to avoid performance degradation
- Assume a hash function h that returns numbers in $\{0, \ldots, 2^k - 1\}$
- Start with $n = 2^i \ll 2^k$, only look at i least significant bits

Extensible Hash Table

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

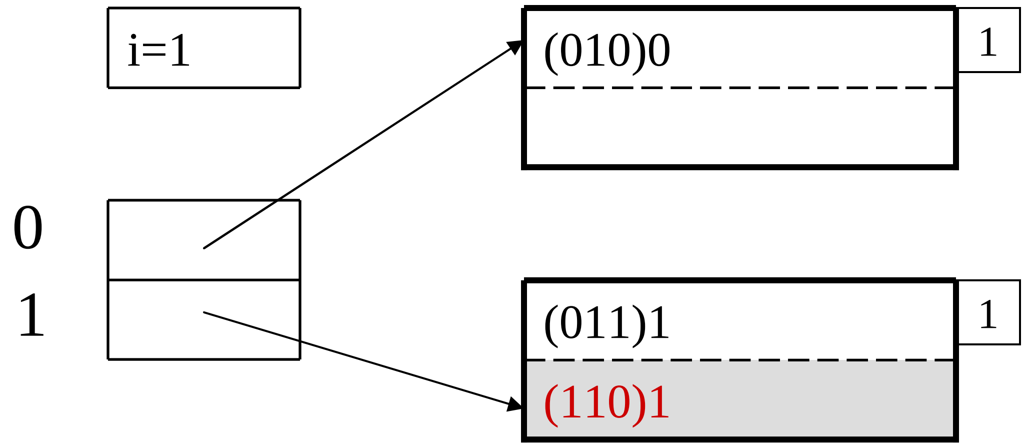
- Keys:
  - 4 (=0100)
  - 7 (=0111)



- Note: we only look at the last bit (0 or 1)

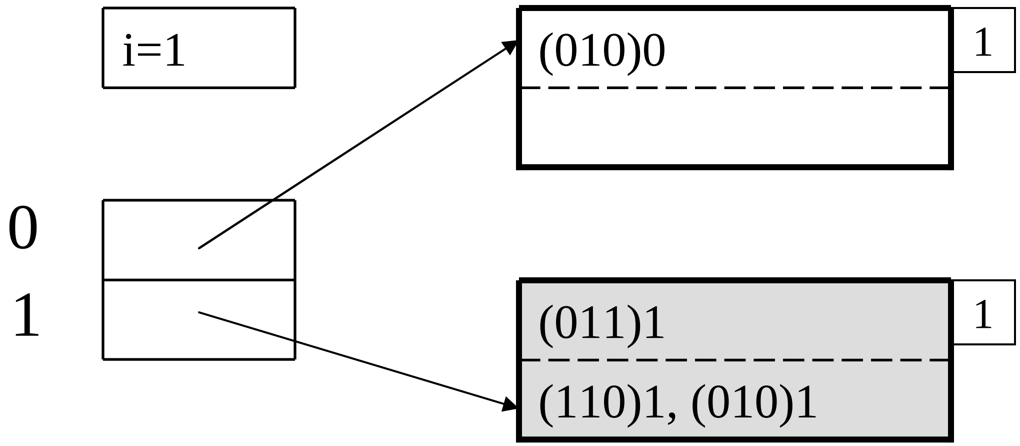
Insertion in Extensible Hash Table

- Insert 13 (=1101)



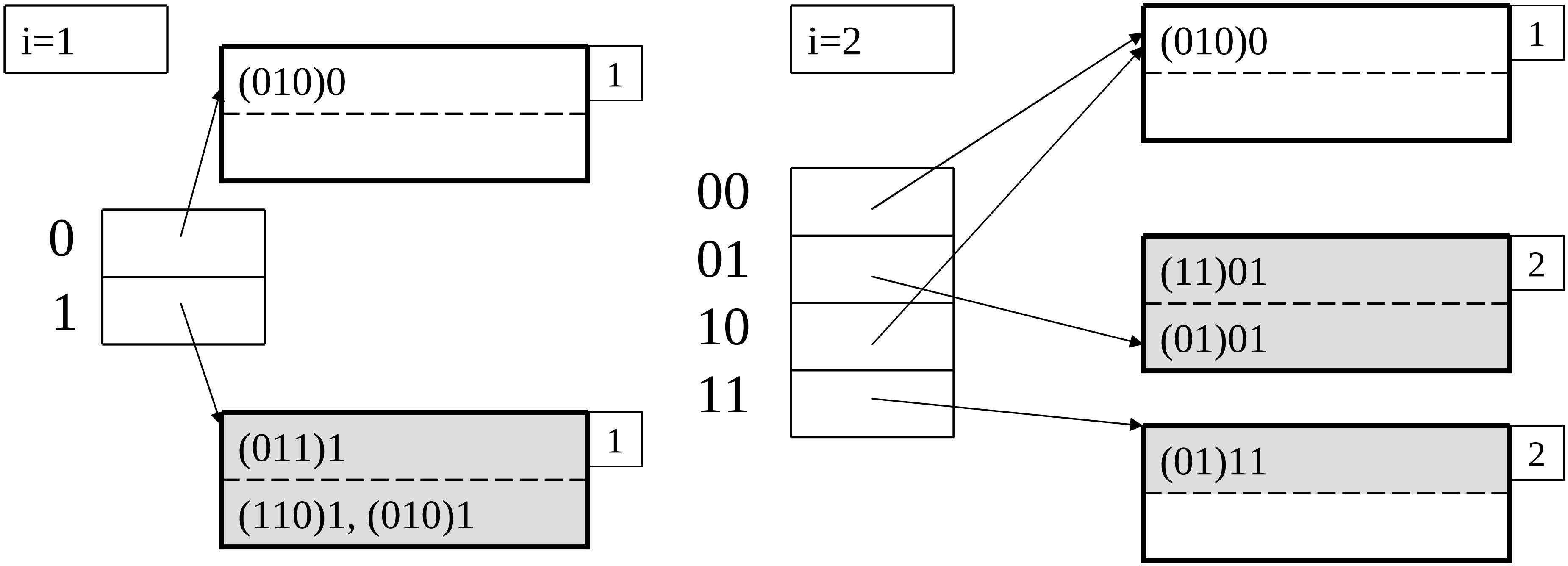
Insertion in Extensible Hash Table

- Now insert 0101



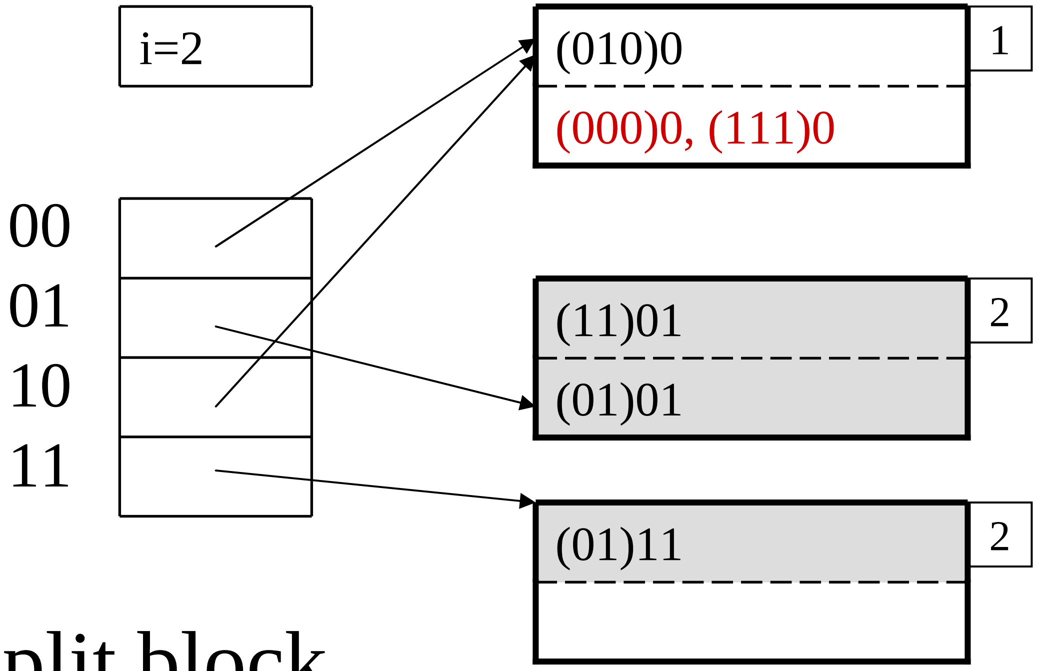
- Need to extend table, split blocks
- i becomes 2

Insertion in Extensible Hash Table



Insertion in Extensible Hash Table

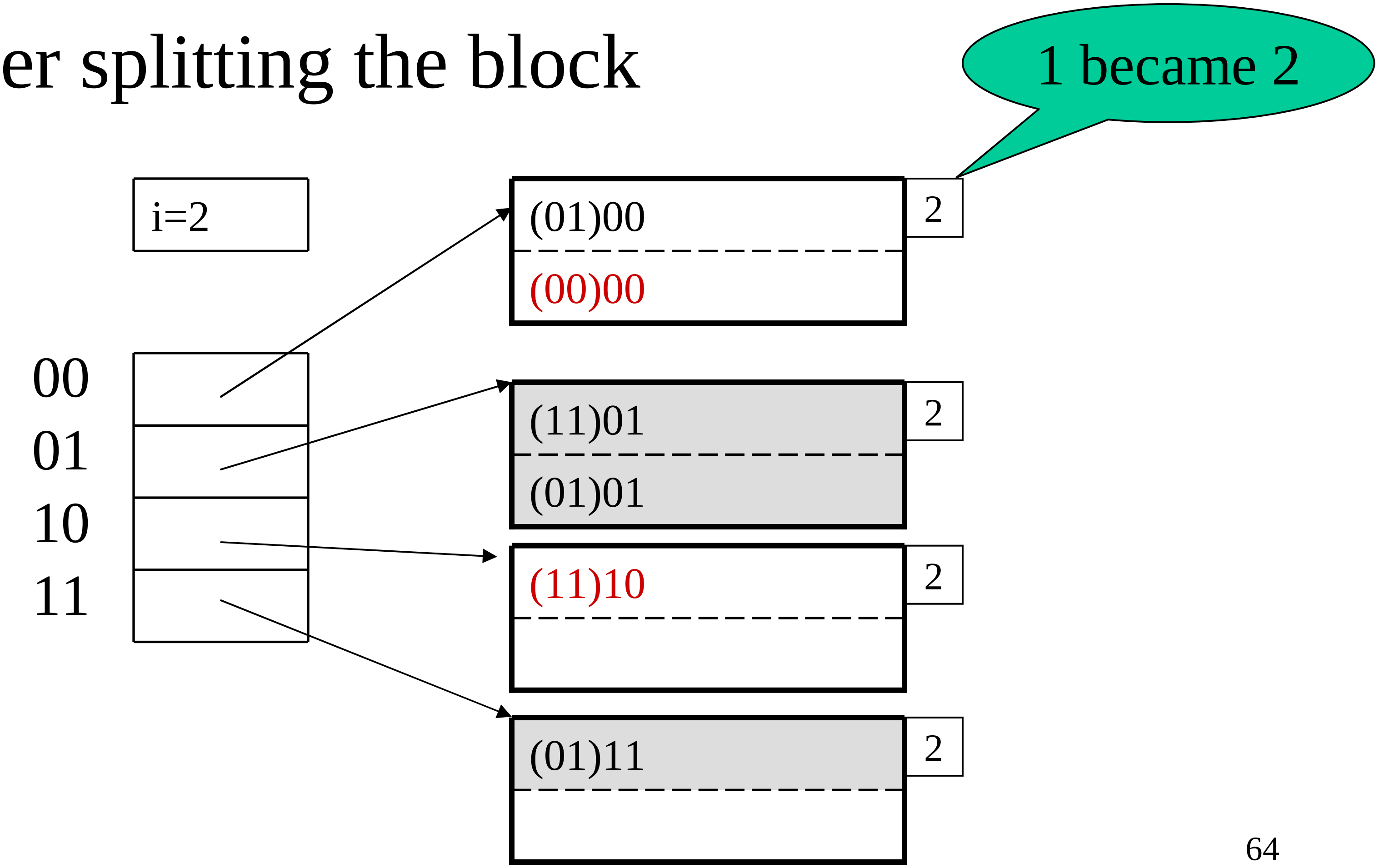
- Now insert 0000, 1110



- Need to split block

Insertion in Extensible Hash Table

- After splitting the block



Extensible Hash Table

- How many buckets (blocks) do we need to touch after an insertion ?

- How many entries in the hash table do we need to touch after an insertion ?

Performance Extensible Hash Table

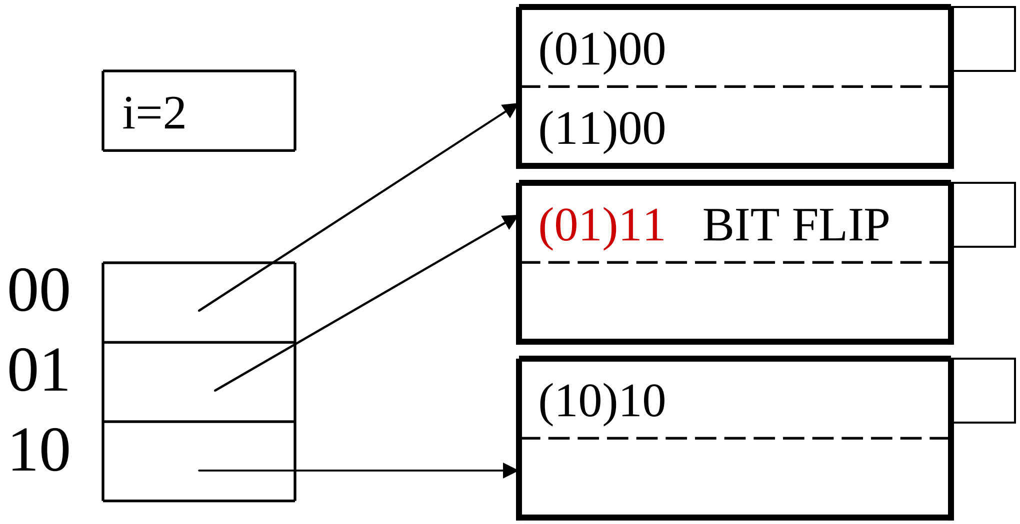
- No overflow blocks: access always one read
- BUT:
  - Extensions can be costly and disruptive
  - After an extension table may no longer fit in memory

Linear Hash Table

- Idea: extend only one entry at a time
- Problem: n= no longer a power of 2
- Let i be such that $2^i <= n < 2^{i+1}$
- 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)

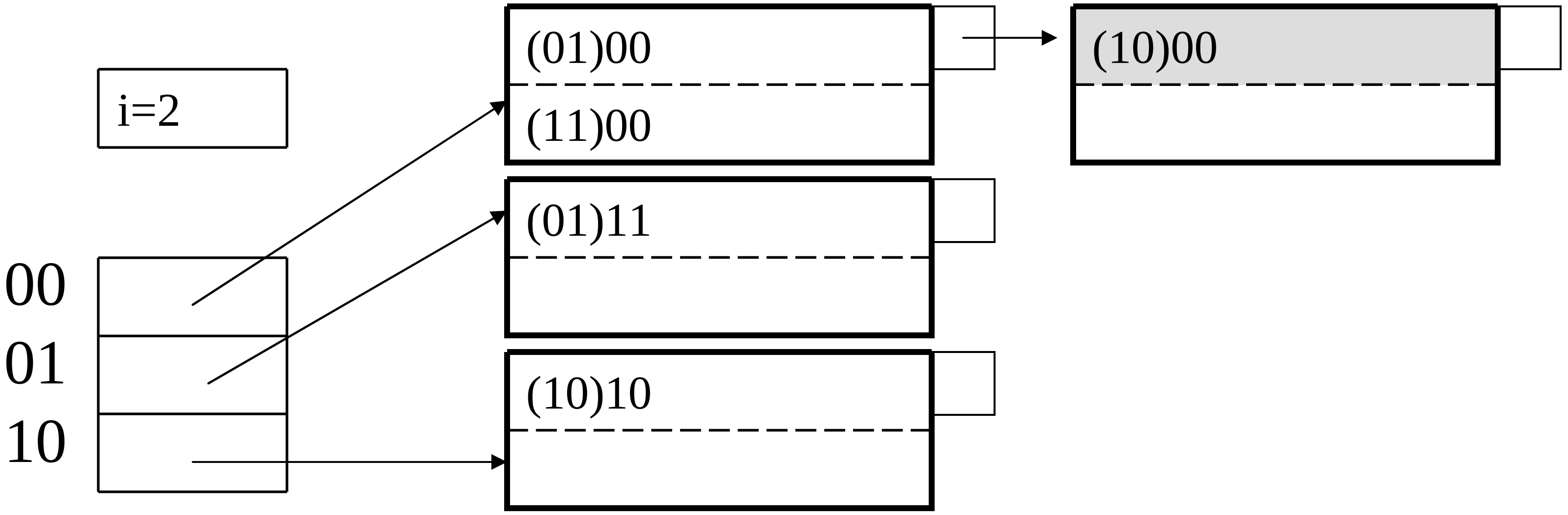
Linear Hash Table Example

- n=3



Linear Hash Table Example

- Insert 1000: overflow blocks…

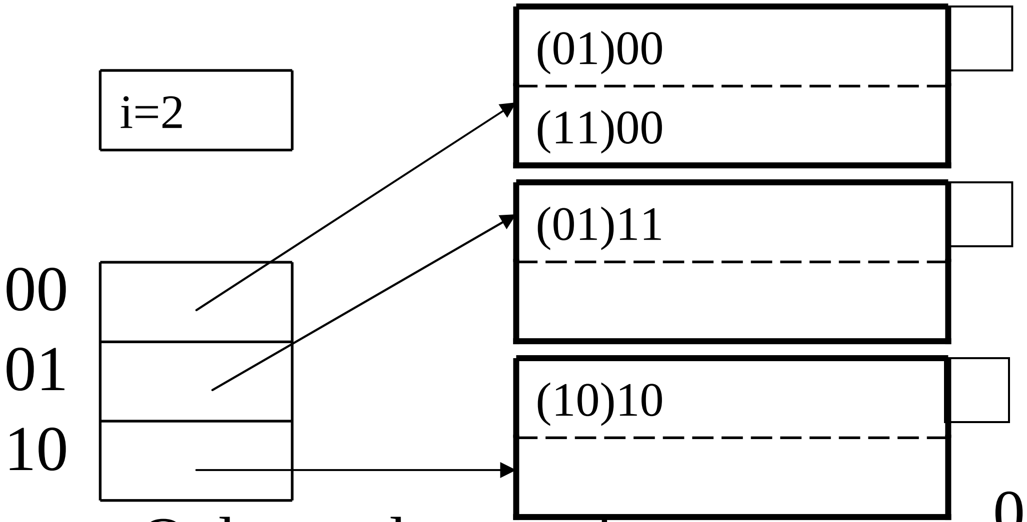


Linear Hash Tables

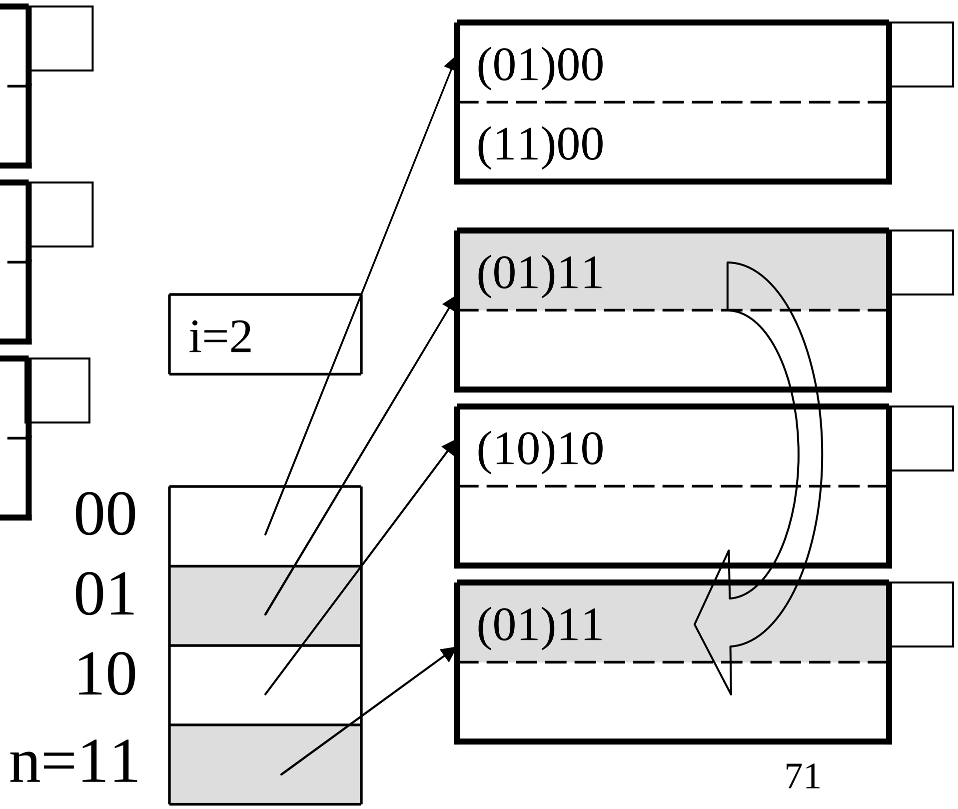
- Extension: independent on overflow blocks
- Extend n:=n+1 when average number of records per block exceeds (say) 80%

Linear Hash Table Extension

- From n=3 to n=4



- Only need to touch one block (which one ?)



Linear Hash Table Extension

- From n=3 to n=4 finished

- Extension from n=4 to n=5 (new bit)
- Need to touch every single block (why ?)