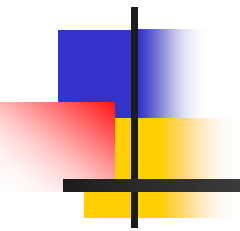
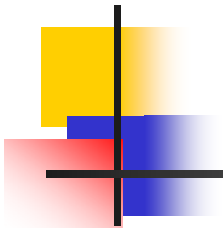


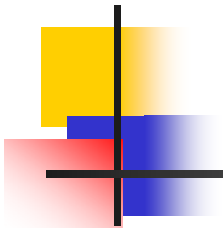
Index Structures

Chapter 13 of GUW



Objectives

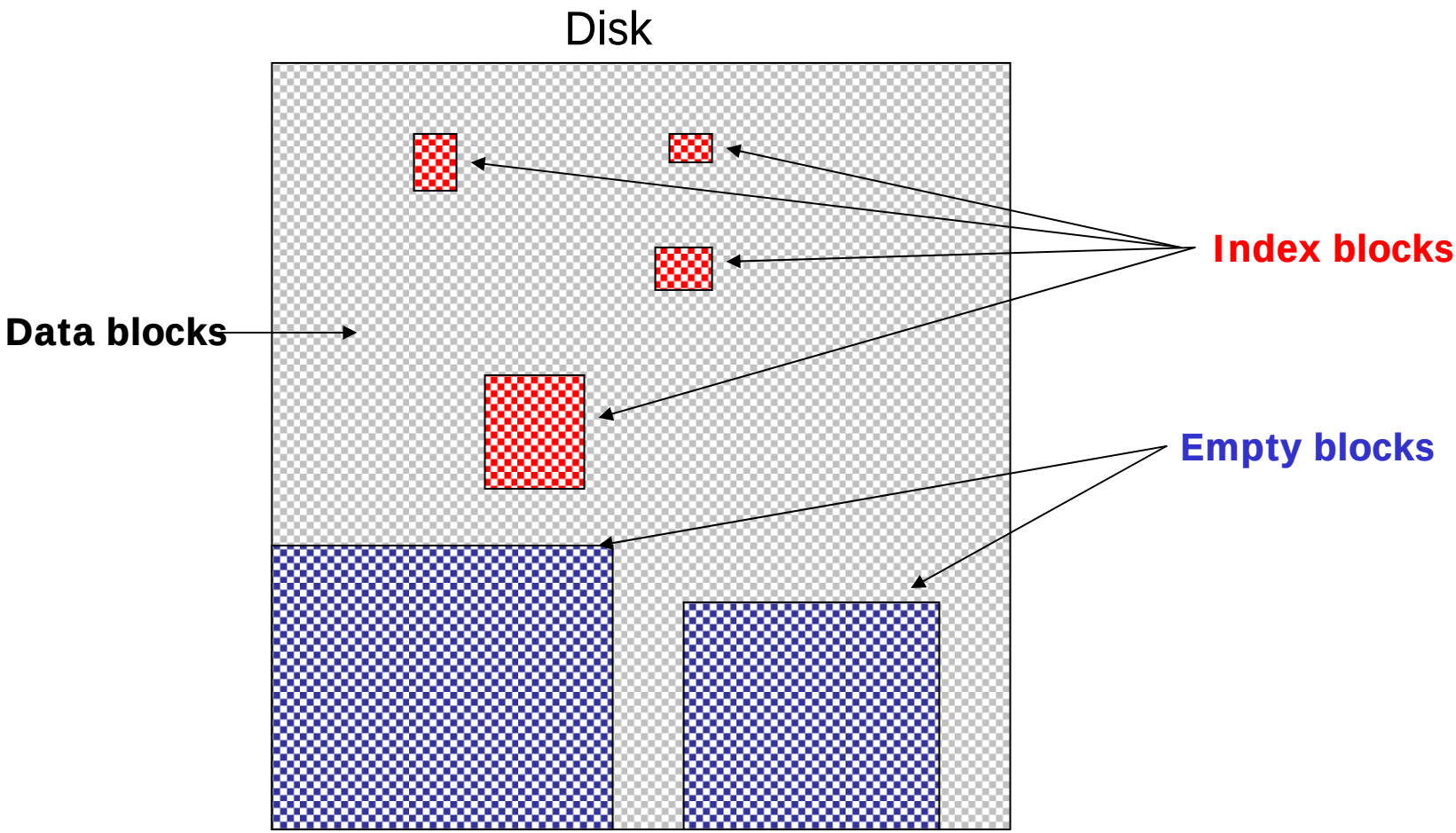
- Different ways of organizing blocks
- What is the best way to organize record in blocks to minimize:
  - Query cost
    - Exact match
    - Partial match
    - Range
    - Join
  - Insertion cost
  - Deletion cost
  - Update cost
  - Storage cost



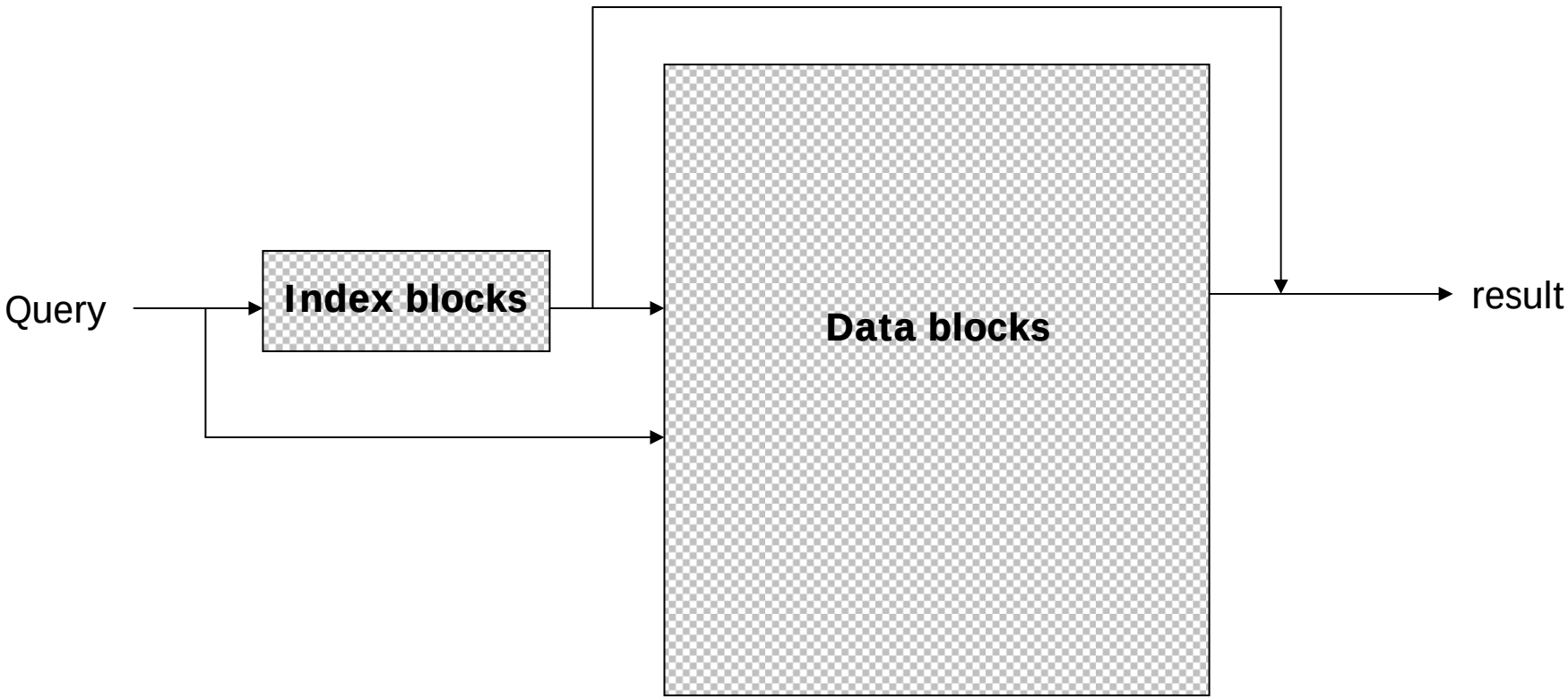
- Lecture outline

- Basic Concepts
- Index on Sequential files
- Secondary Indexes
- B-Trees
- Hash Tables

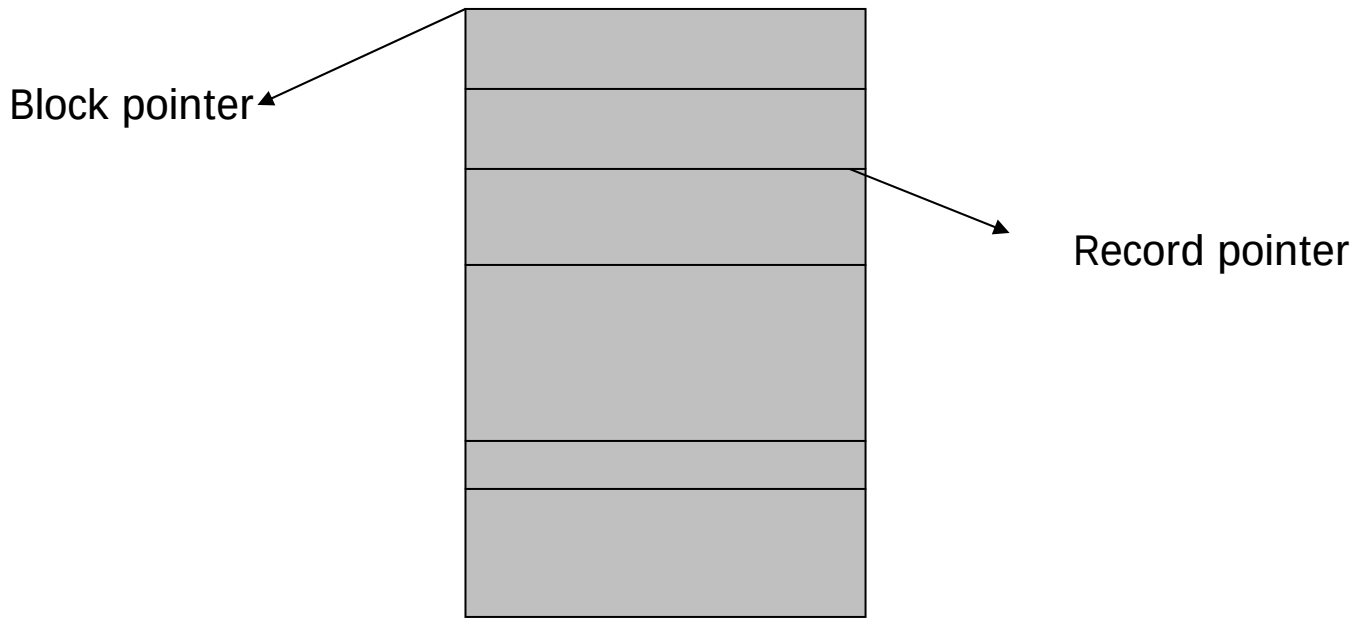
- Basic Concepts ...



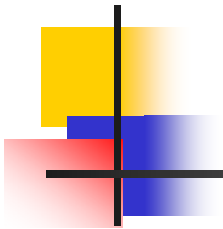
... - Basic Concepts ...



... - Basic Concepts



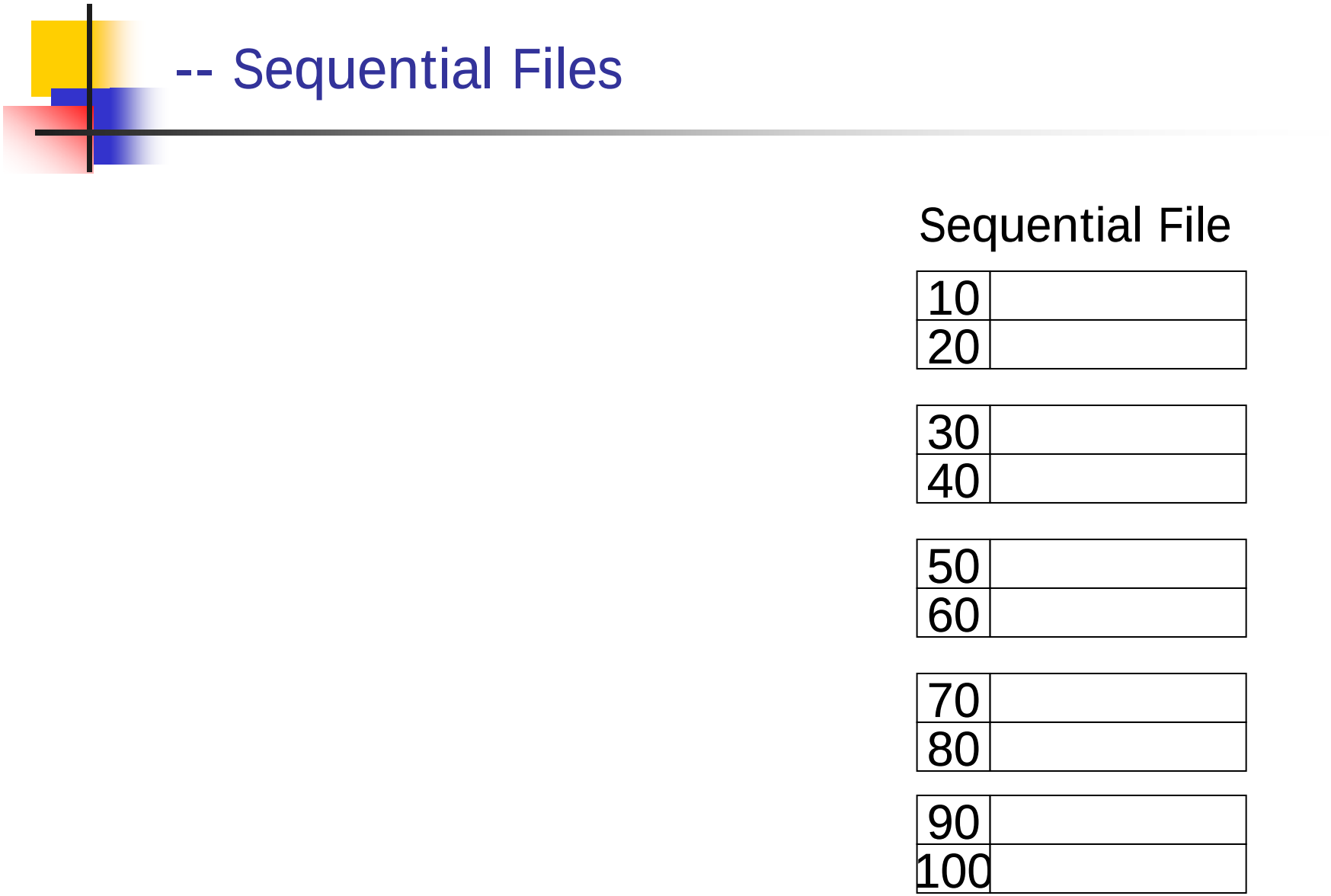
- Record pointers take more space than block pointers. Why?



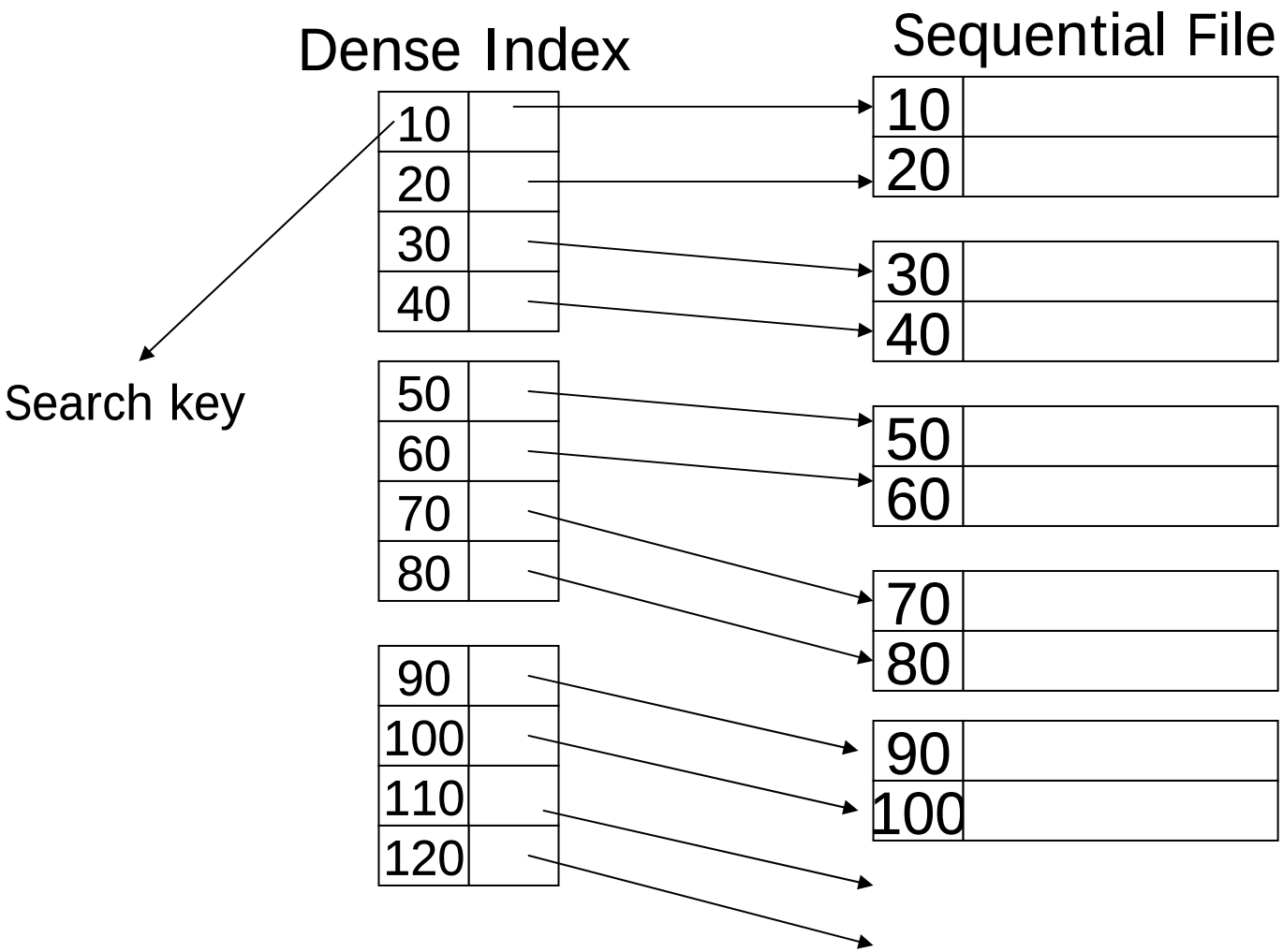
- Indexes Sequential Files

- Sequential files
- Dense Index
- Sparse Index
- Multiple level of Index
- Index with Duplicate Search Keys
- Managing Indexes During Data Modification

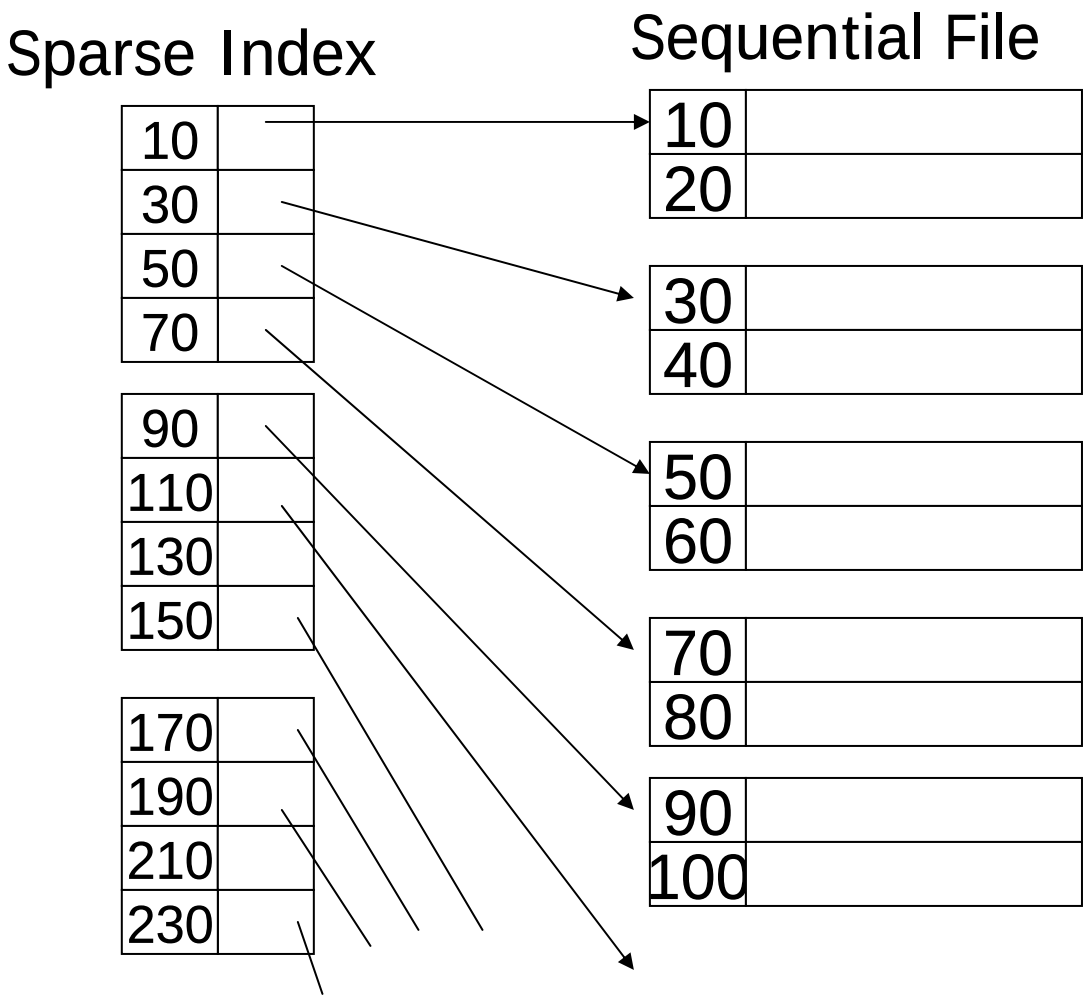
-- Sequential Files

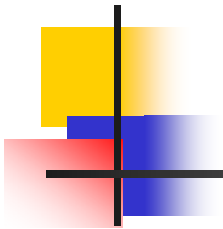


-- Dense Index


Dense Index
Sequential File
Search key
10
20
30
40
50
60
70
80
90
100
110
120
10
20
30
40
50
60
70
80
90
100

-- Sparse Index

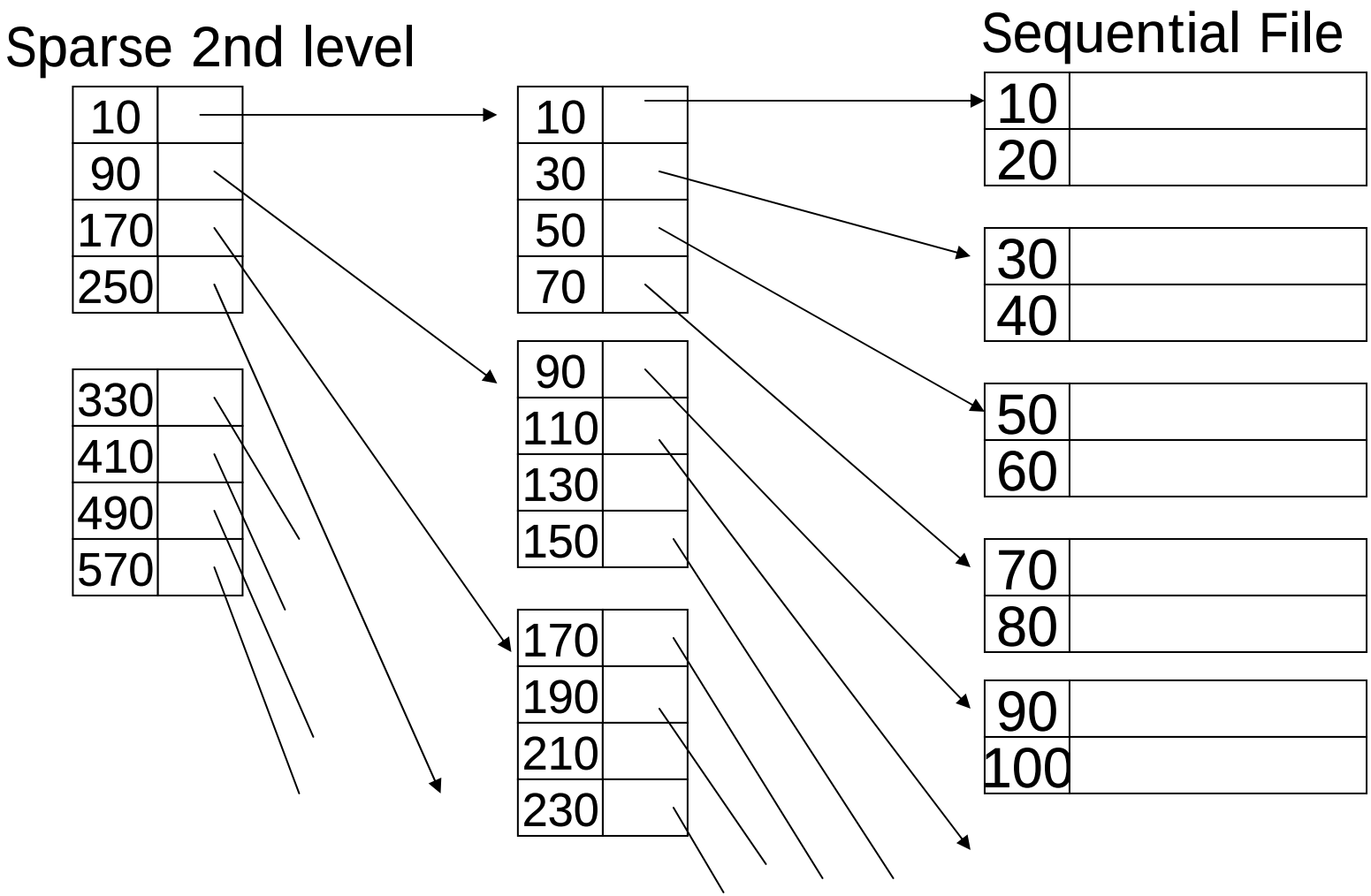




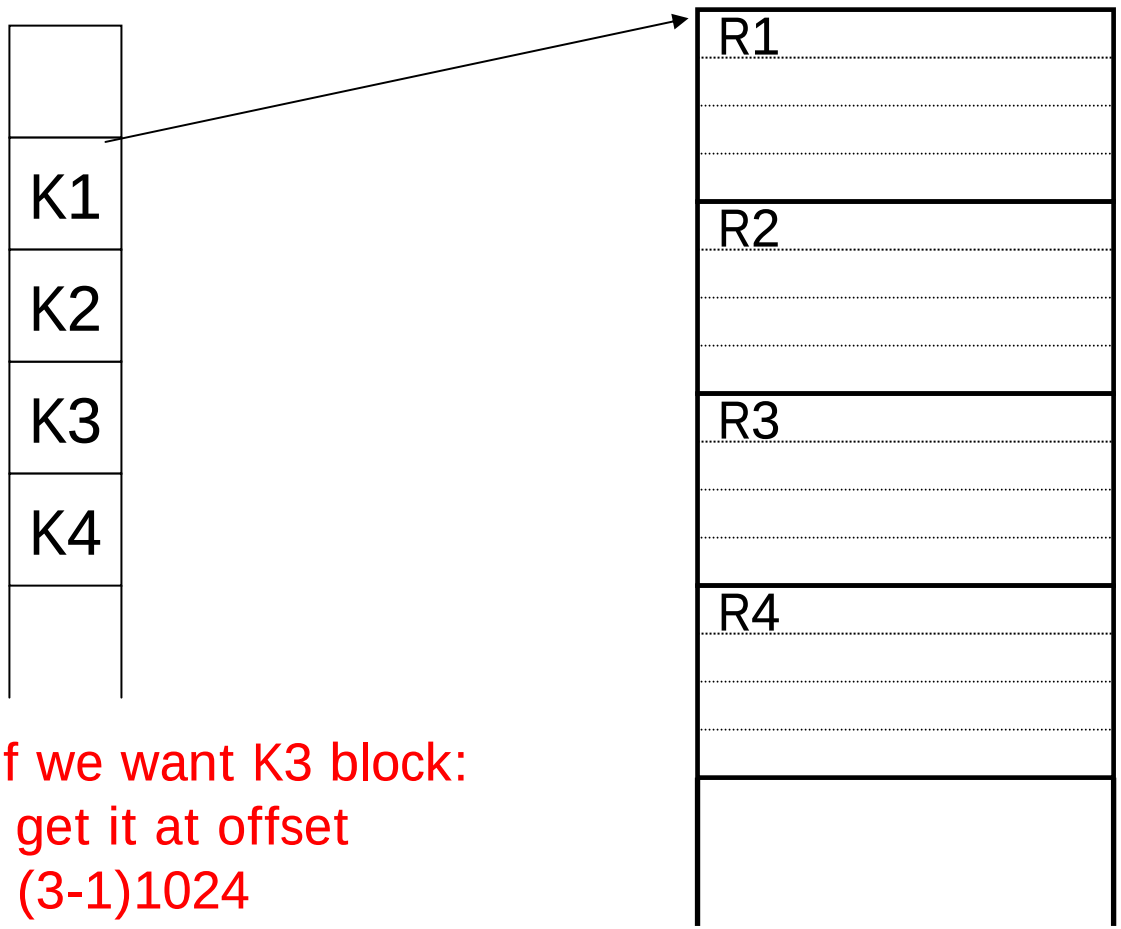
-- Dense Vs Sparse Index: Example

- Relation
  - Relation R with 1,000,000 tuples
  - A block of size 4096 bytes (4k)
  - 10 R tuples per block
  - Data space required => 1000000/10 * 4k = 400MB.
- Dense index
  - record size: 30 Bytes for search key + 8 bytes for record pointer
  - Can fit 100 index records per block
  - Dense index space = 1000000/100 * 4k = 40MB.
  - Binary search cost $\log_2(10000) = 14$ disk accesses at most
  - Keeping blocks (1/2, ¼, ¾, 1/8, ...) in memory can lower disk access.
- Sparse index
  - 1000 index blocks = 4MB
  - Binary search cost $\log_2(1000) = 10$ disk accesses at most

-- Multiple level of Index



-- Contiguous sequential file

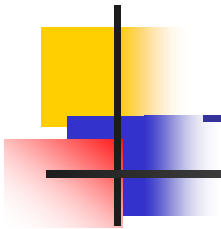


say:
1024 B
per block

- if we want K3 block:
  get it at offset
  (3-1)1024
  = 2048 bytes

-- Sparse vs. Dense Tradeoff

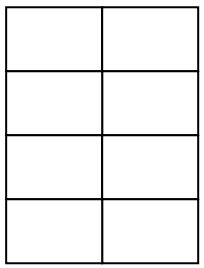
- Sparse
  - Less index space per record can keep more of index in memory
- Dense
  - Can tell if any record exists without accessing file
  - Must be used for secondary index

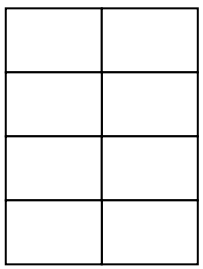


--Terms

- Index sequential file
- Search key (≠ primary key)
- Primary index (on Sequencing field)
- Secondary index
- Dense index (all Search Key values in)
- Sparse index
- Multi-level index

-- Duplicate keys ...





10	
10	

10	
20	

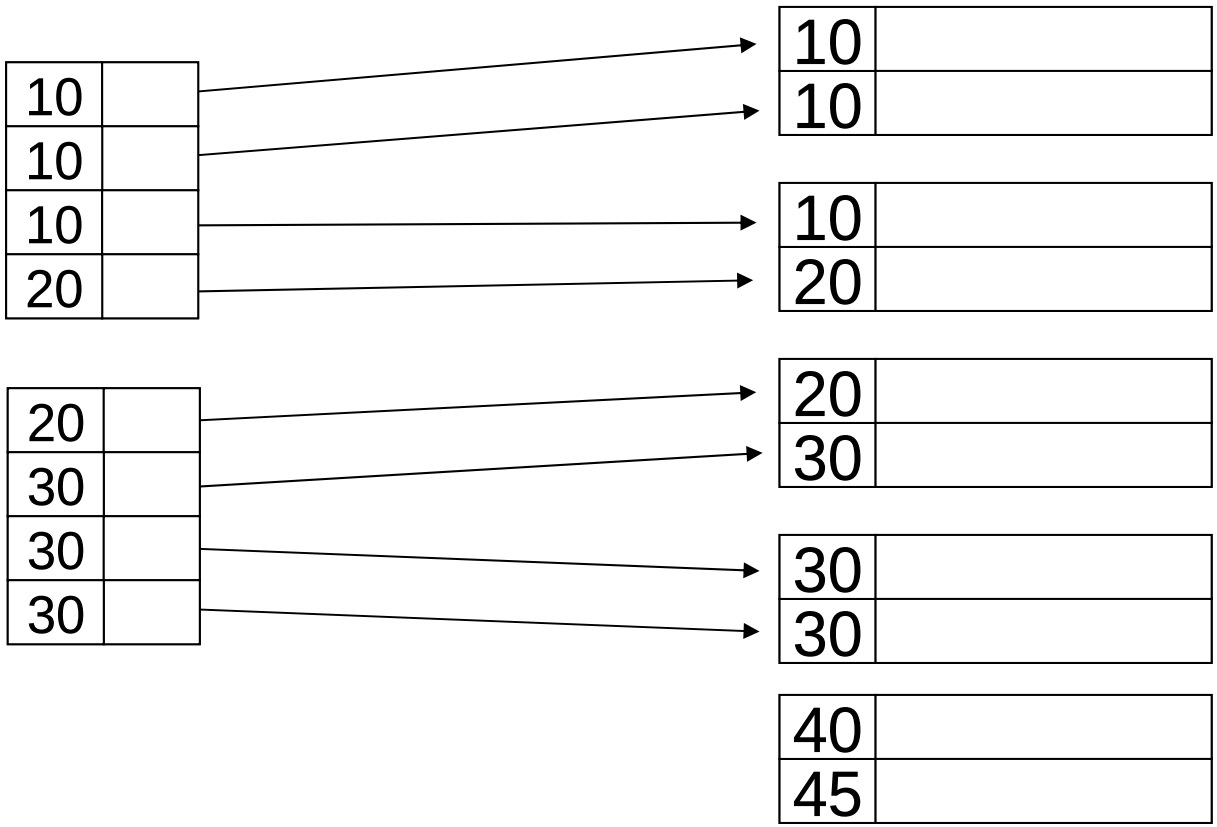
20	
30	

30	
30	

40	
45	

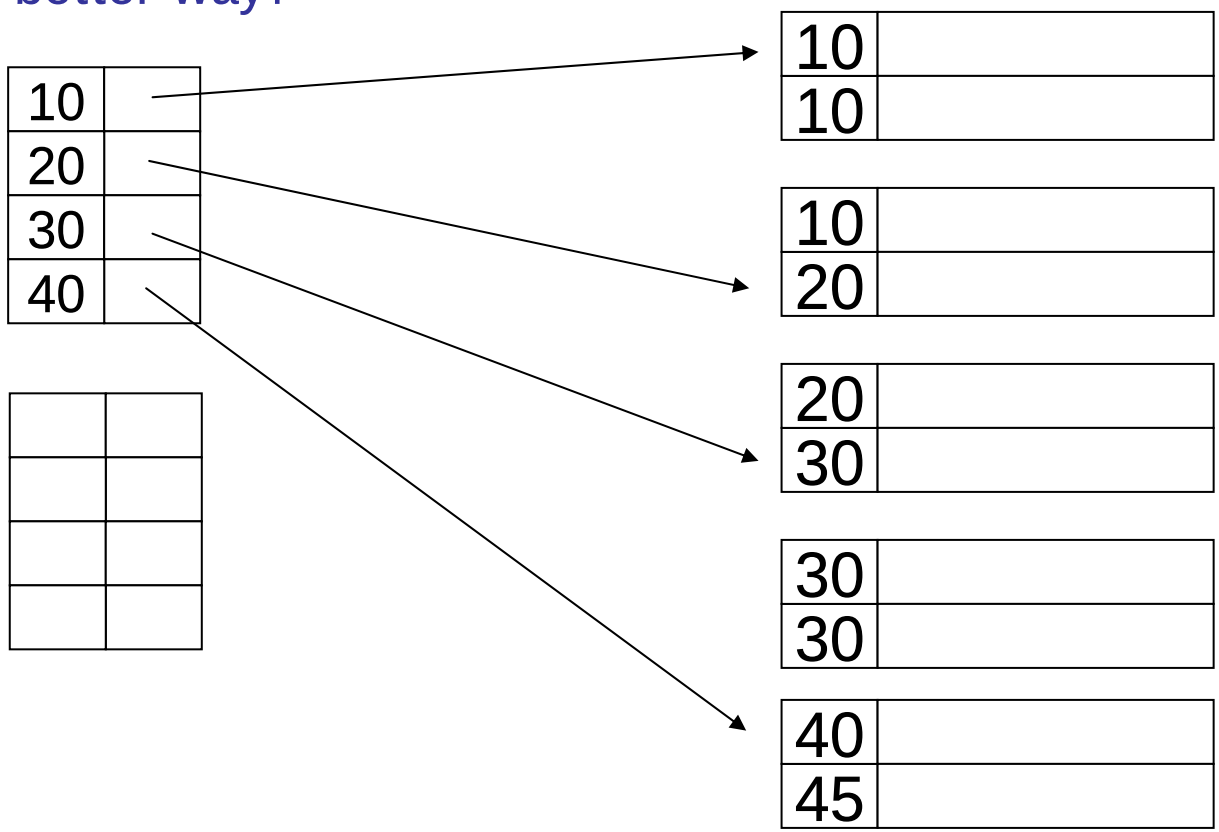
-- Duplicate keys ...

Dense index, one way to implement?



... -- Duplicate keys ...

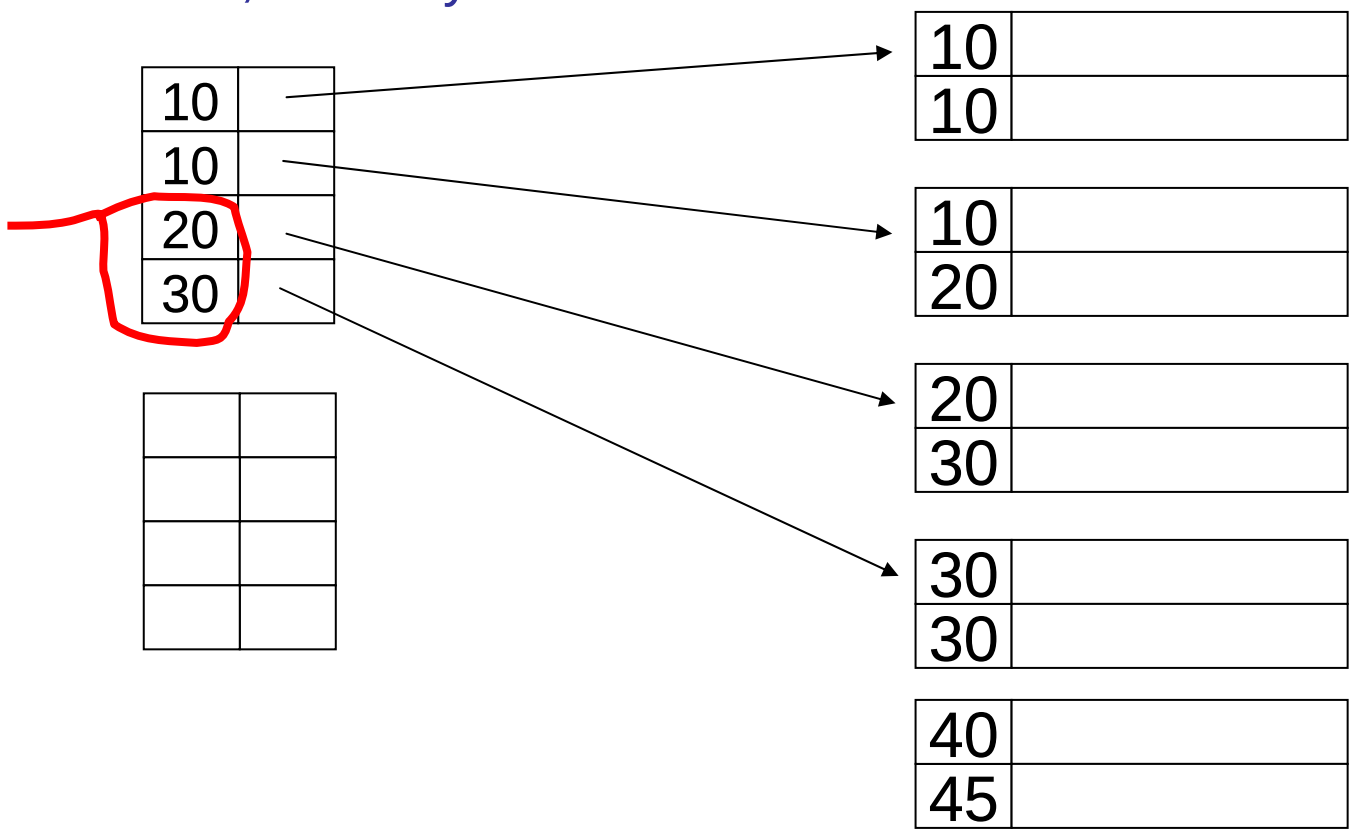
Dense index, better way?



... -- Duplicate keys ....

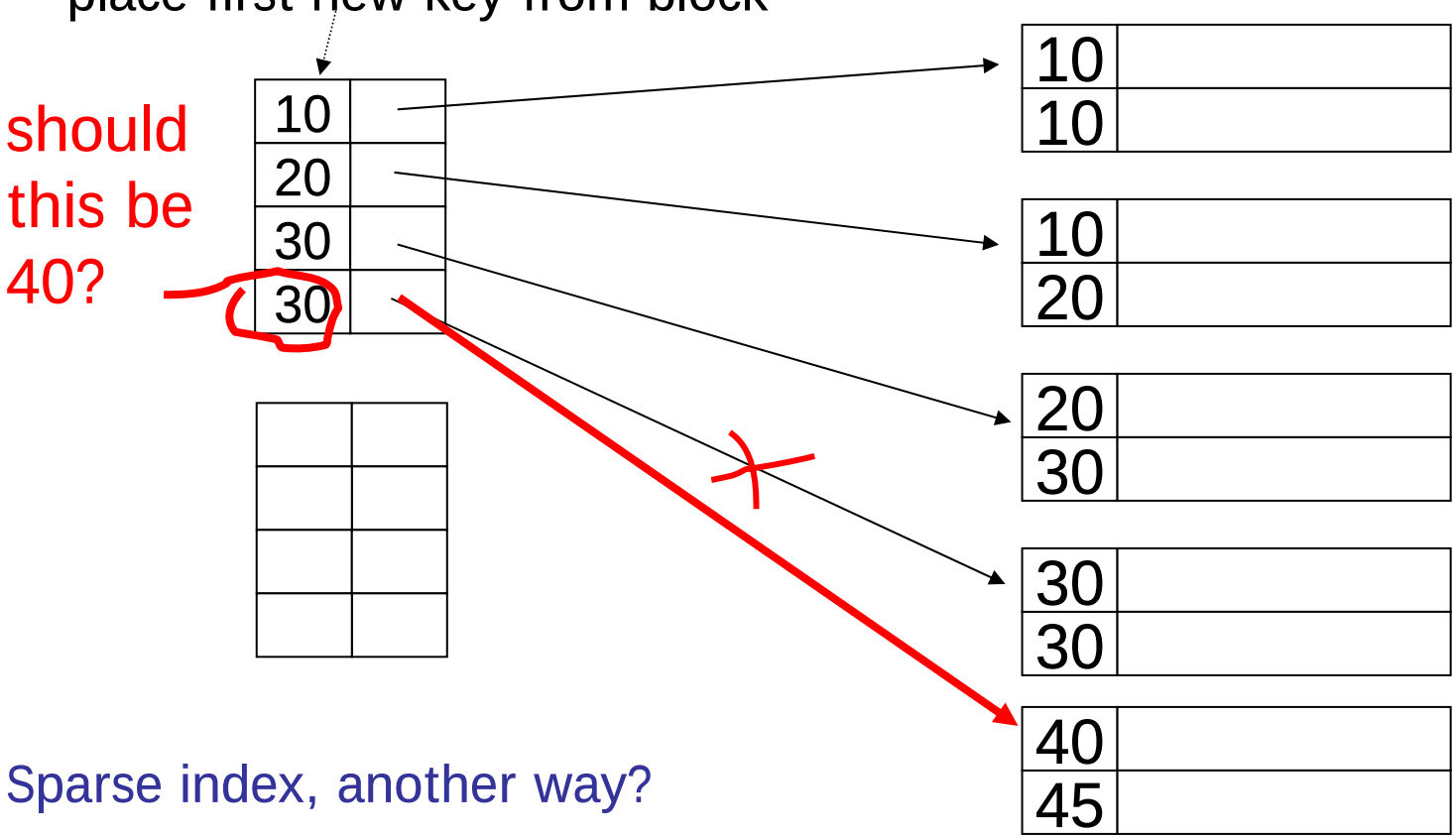
Sparse index, one way?

careful if looking for 20 or 30!



... -- Duplicate keys ...

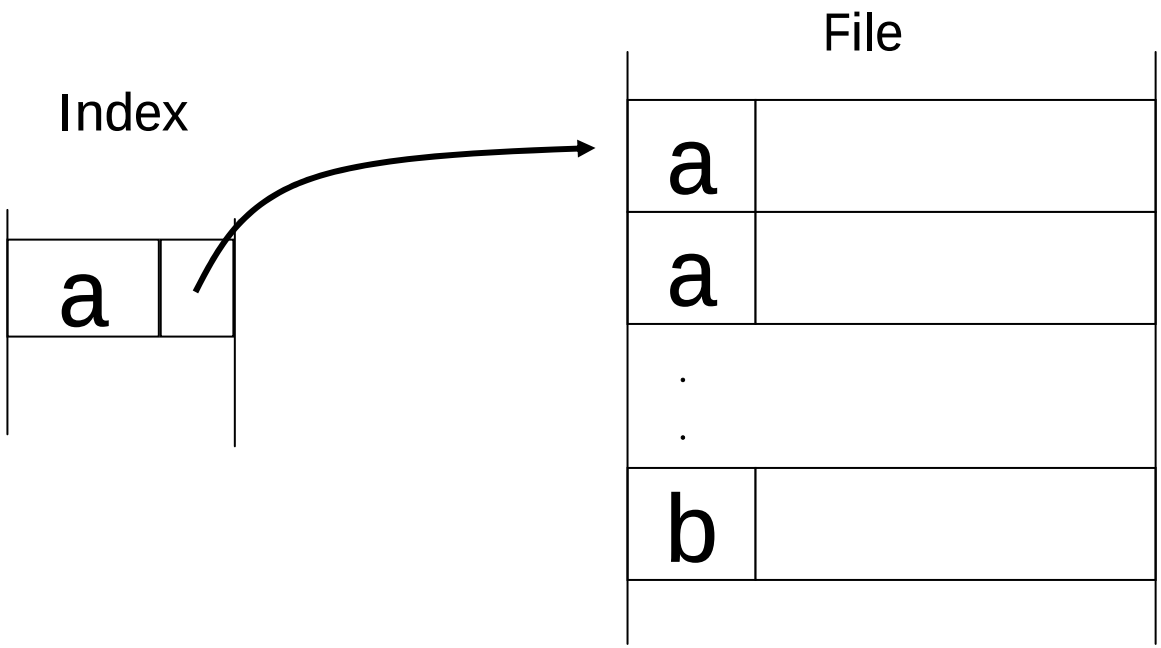
– place first new key from block

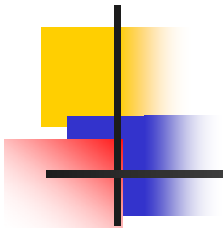


Sparse index, another way?

... -- Duplicate keys

- Incase of primary index may point to first instance of each value only

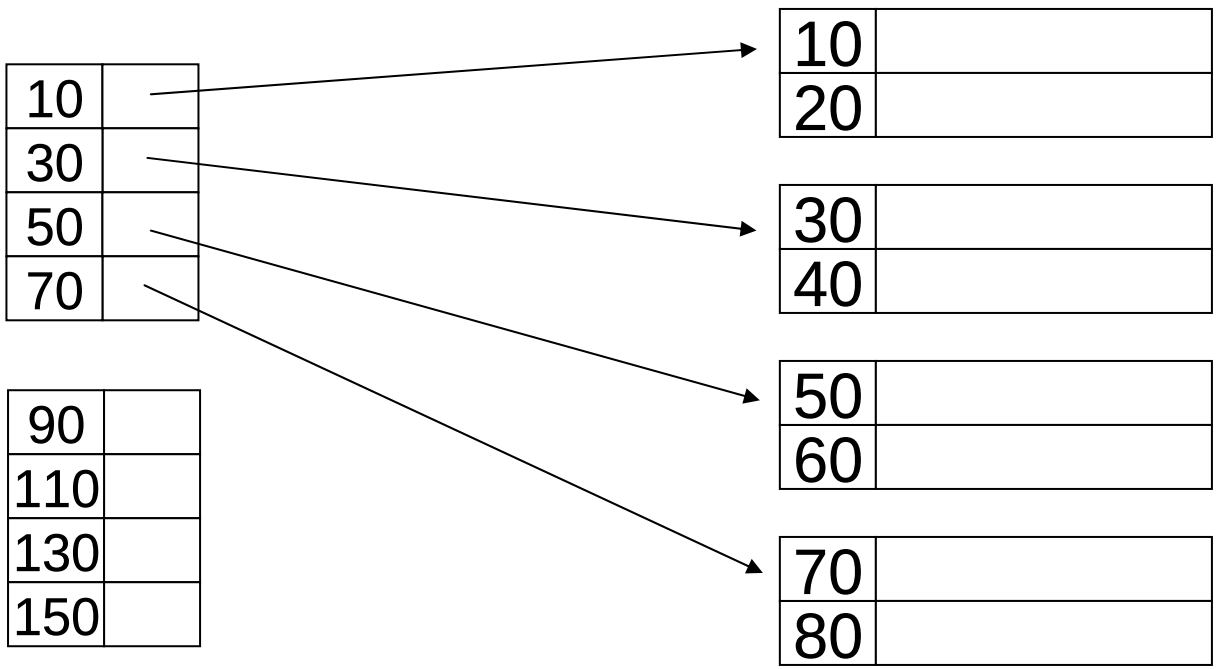




-- Managing Indexes During Data Modification

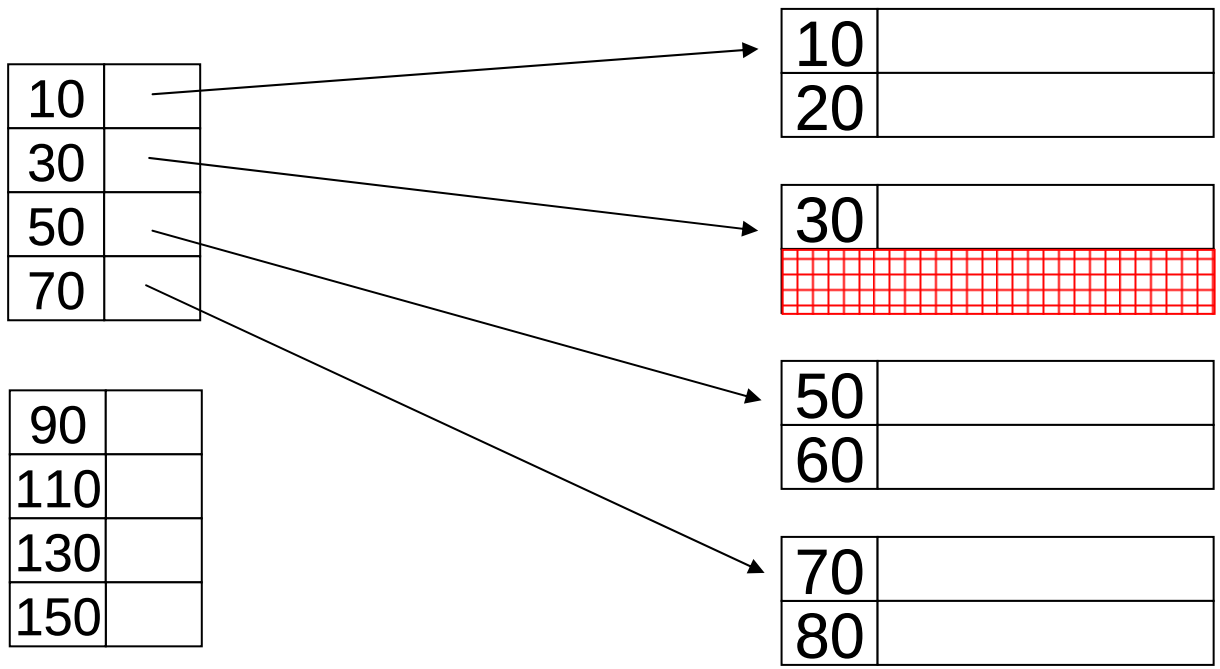
- Deletion from Sparse Index
- Insertion into Sparse Index

--- Deletion from sparse index ...



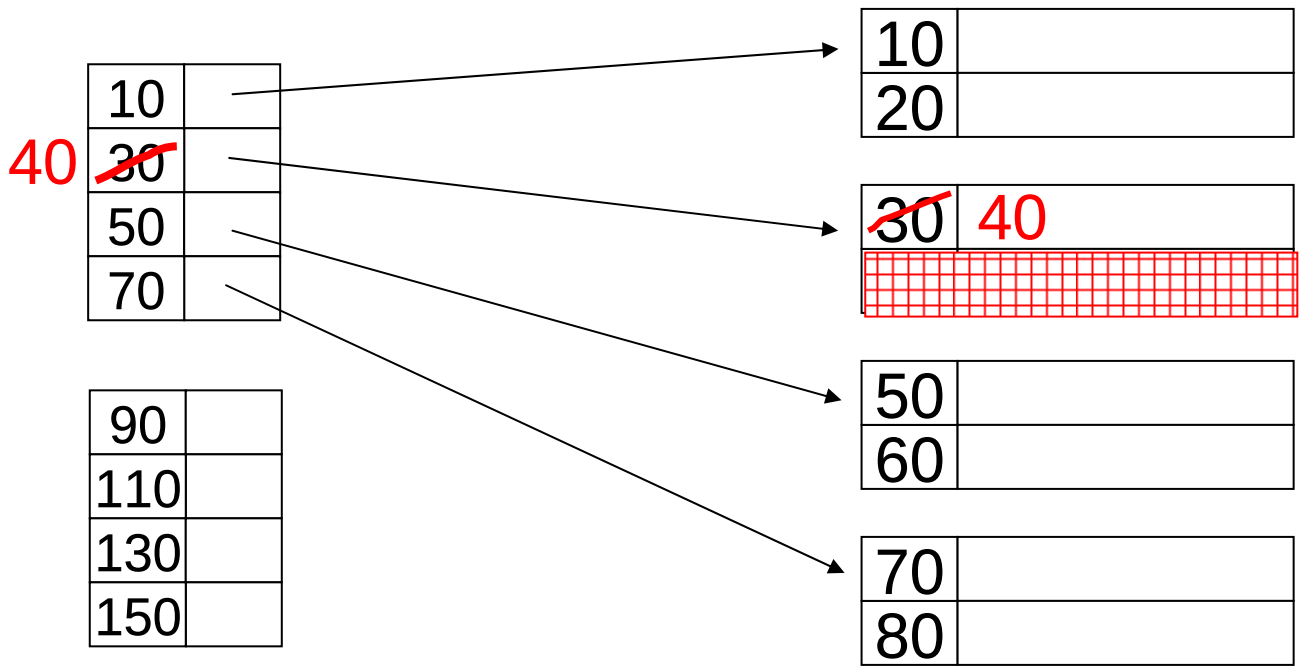
... --- Deletion from sparse index ...

– delete record 40



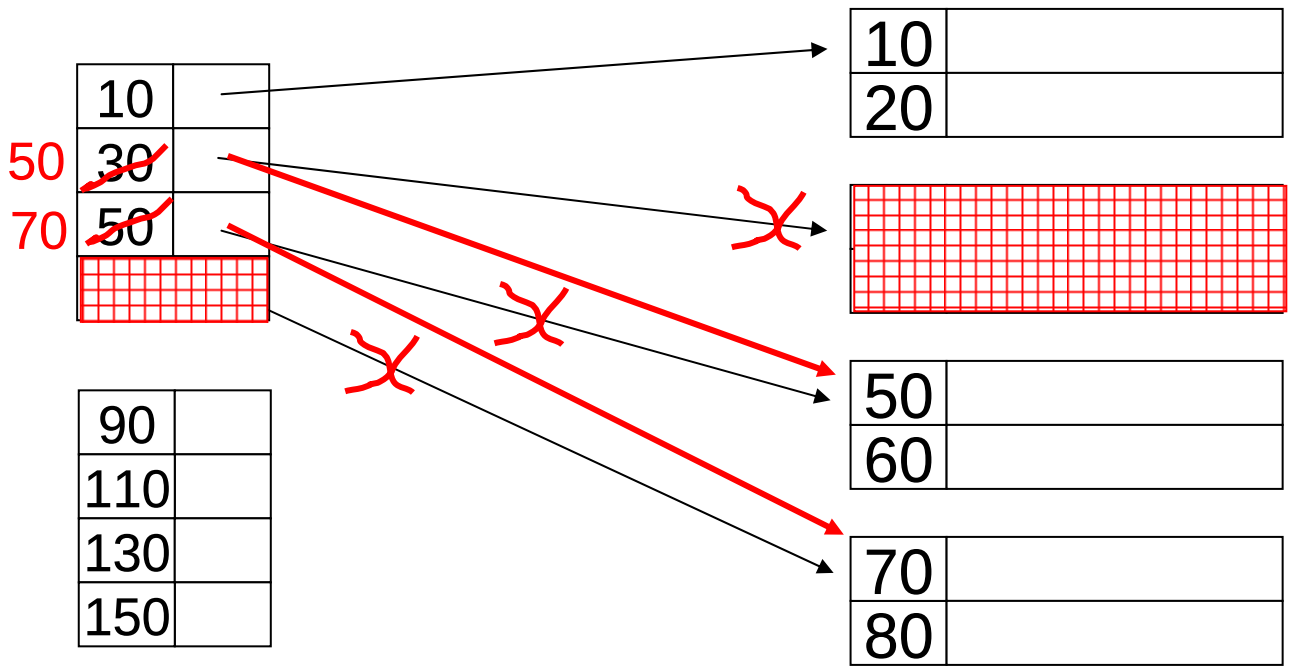
... --- Deletion from sparse index ...

– delete record 30

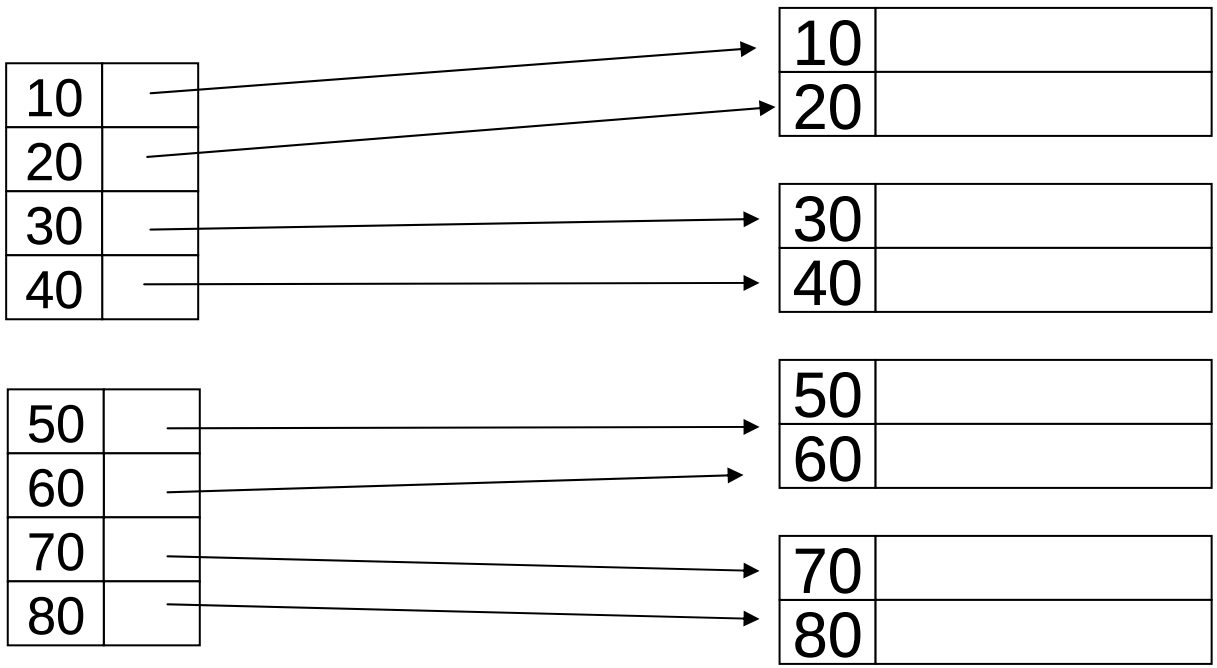


... ---- Deletion from sparse index ...

– delete records 30 & 40

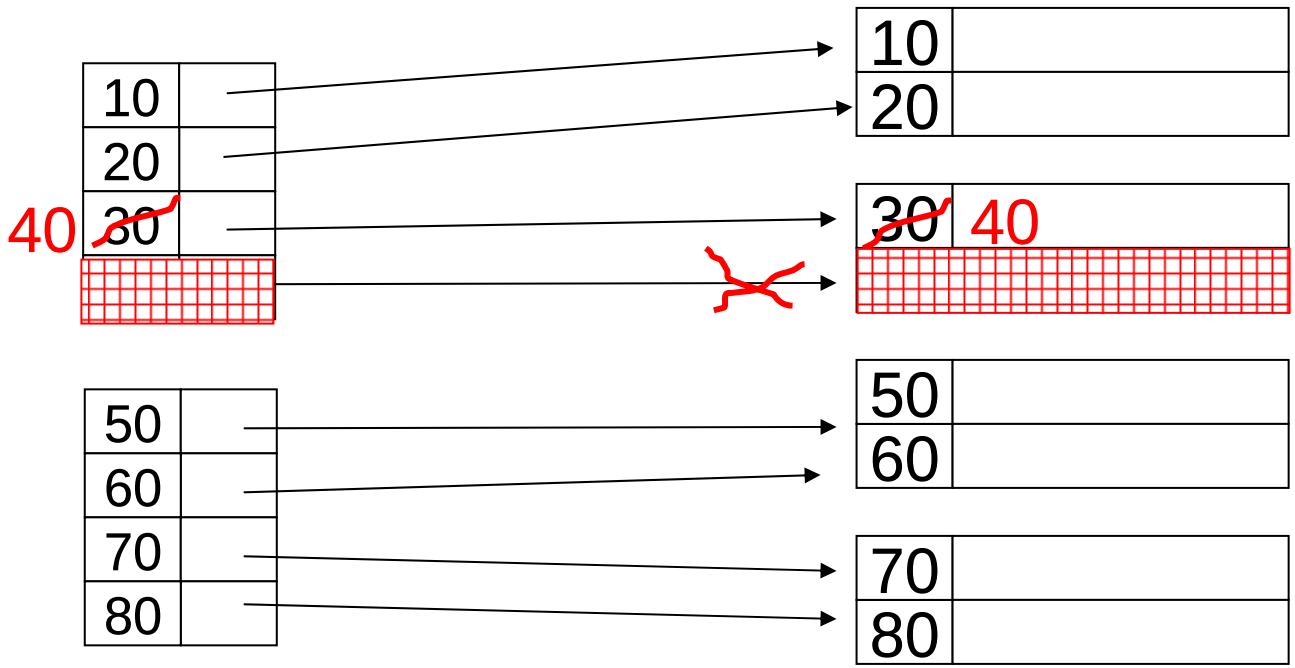


... --- Deletion from dense index ...

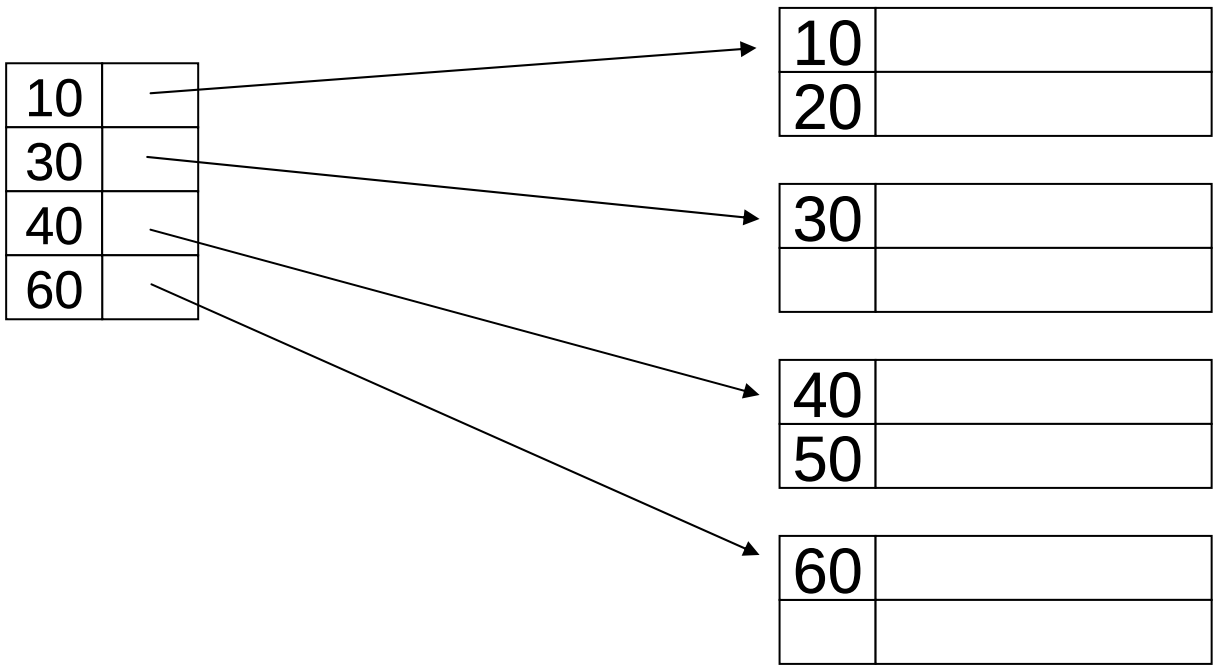


... --- Deletion from dense index

– delete record 30

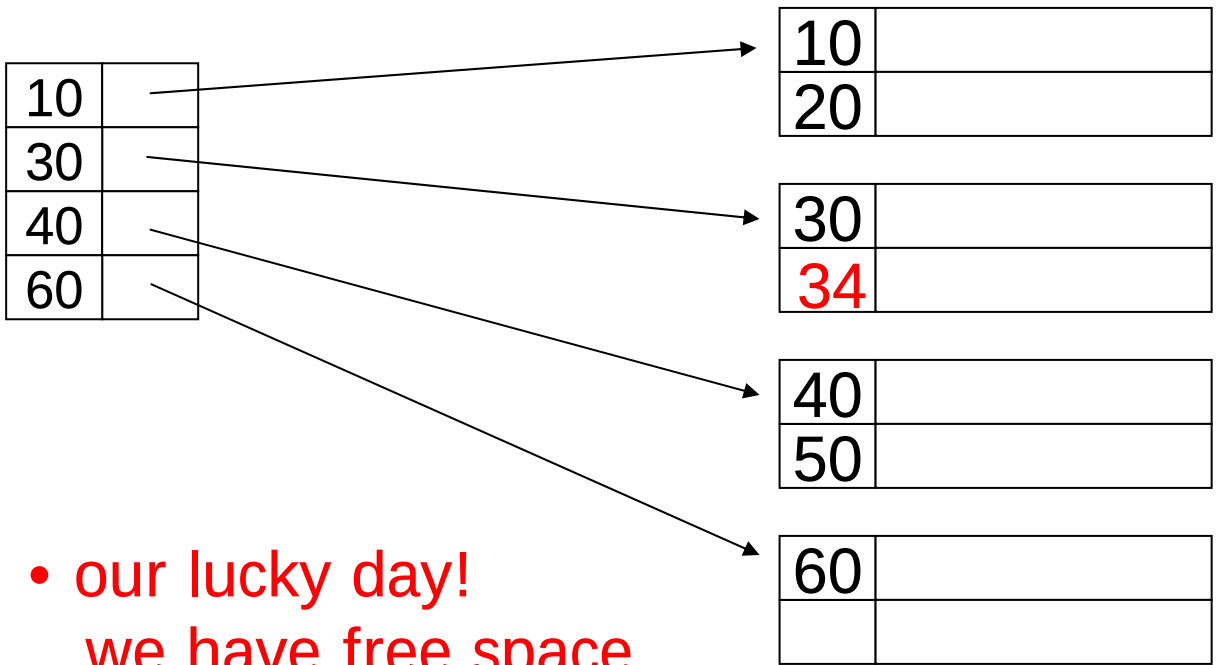


--- Insertion, sparse index case ...



... --- Insertion, sparse index case ...

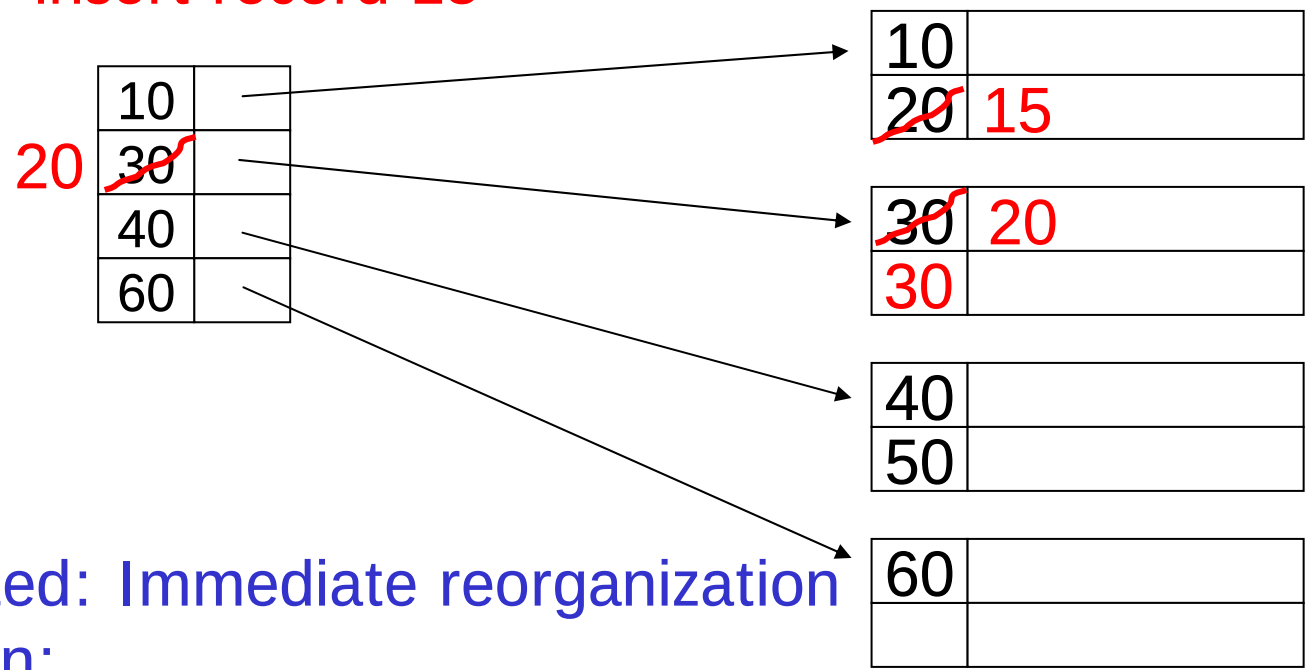
– insert record 34



- our lucky day!
  we have free space
  where we need it!

... --- Insertion, sparse index case ...

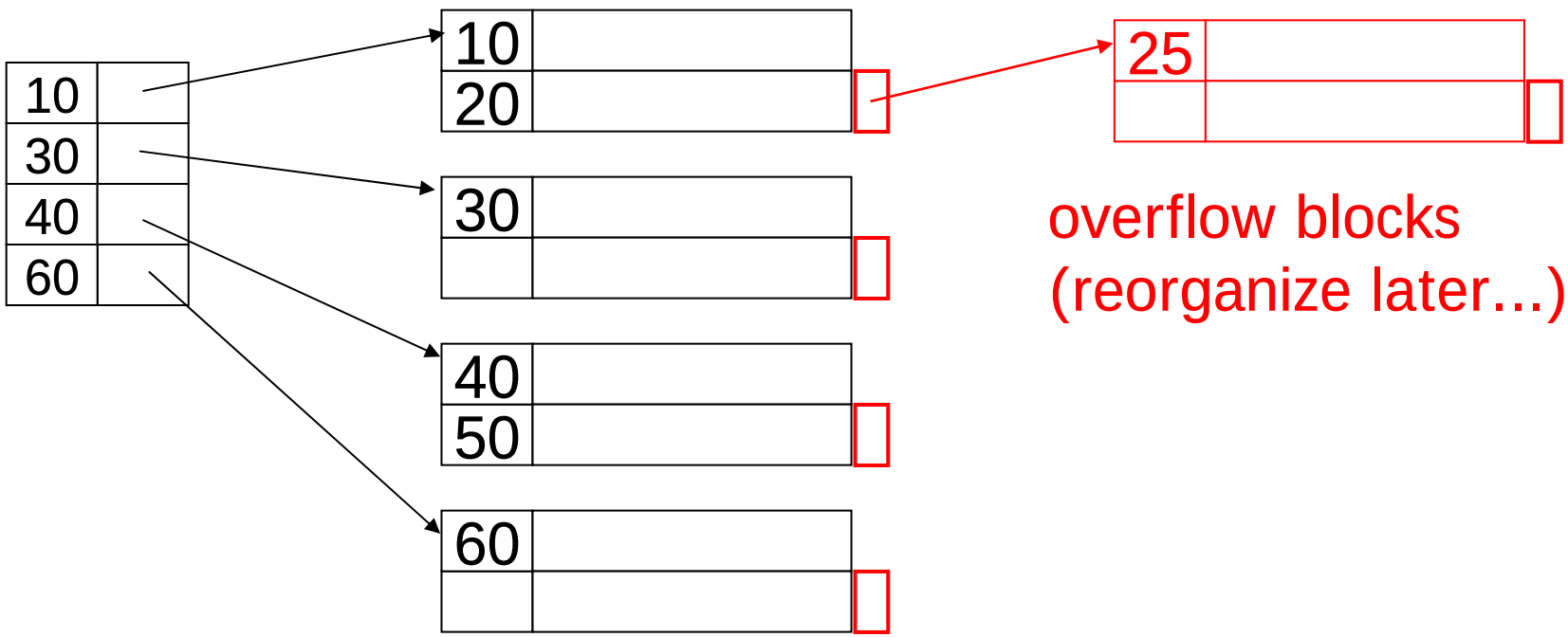
– insert record 15

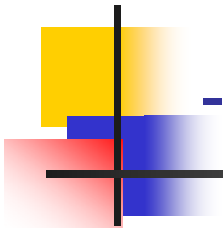


- Illustrated: Immediate reorganization
- Variation:
  - insert new block (chained file)
  - update index

... --- Insertion, sparse index case

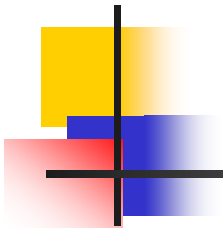
– insert record 25





--- Insertion, dense index case

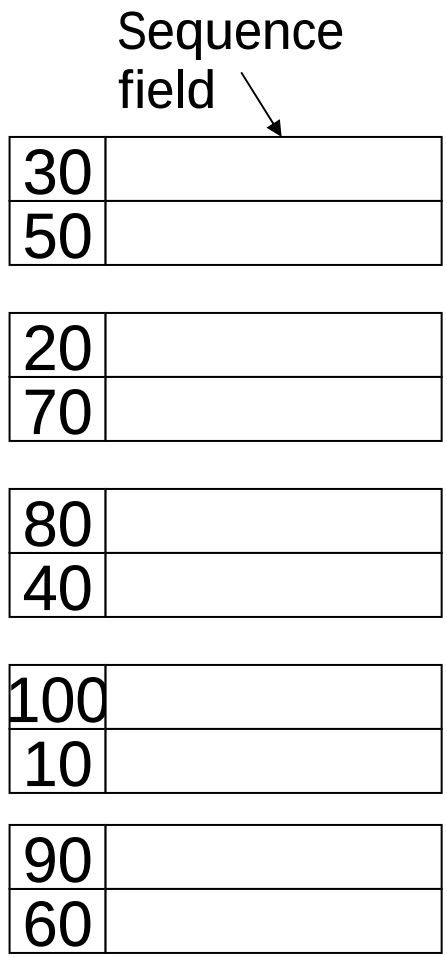
- Similar
- Often more expensive . . .



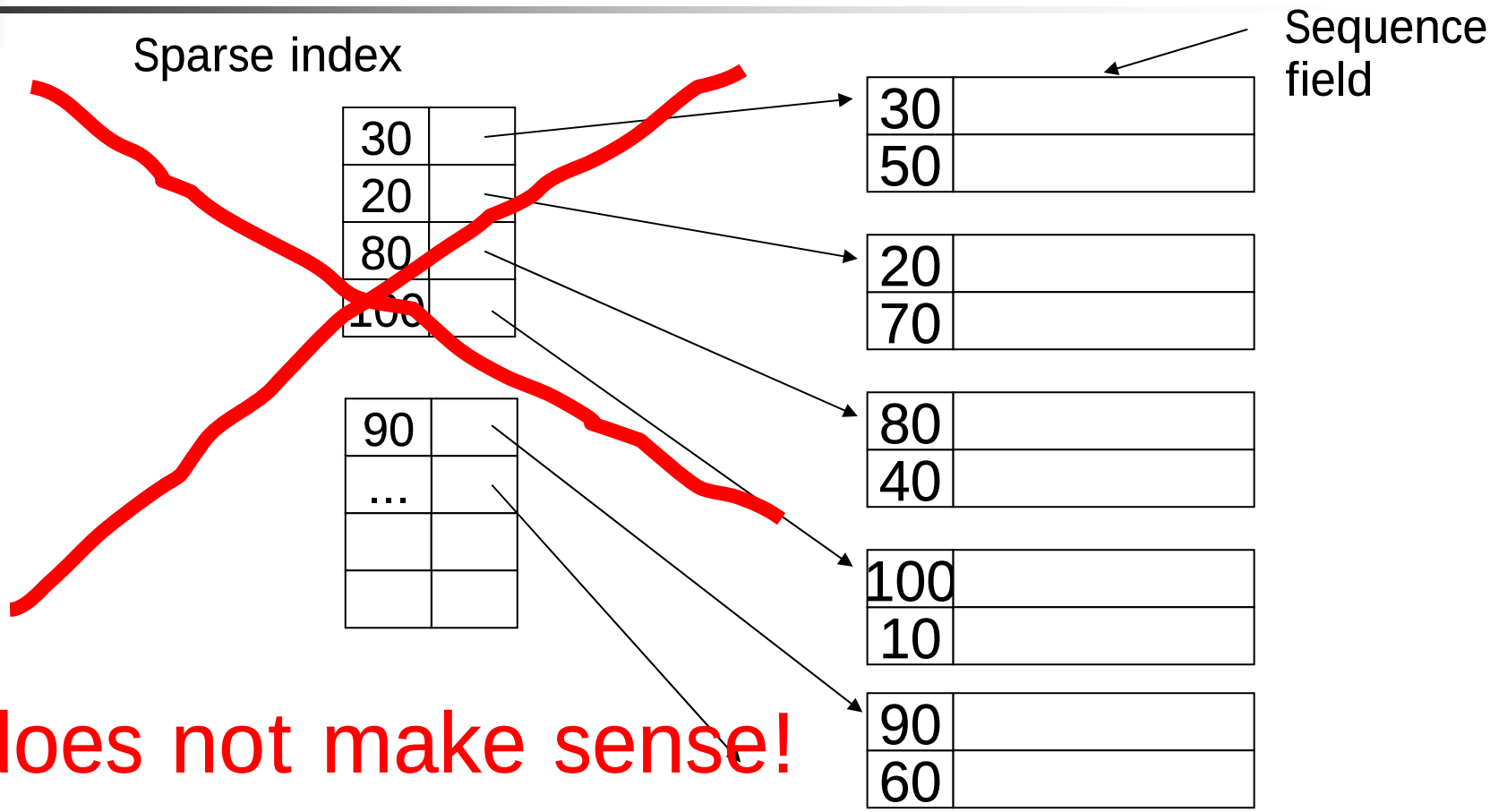
- Secondary Indexes

- Design of Secondary Indexes
- Duplicate Values and Secondary Indexes
- Applications of Secondary Indexes
- Indirection in Secondary Indexes

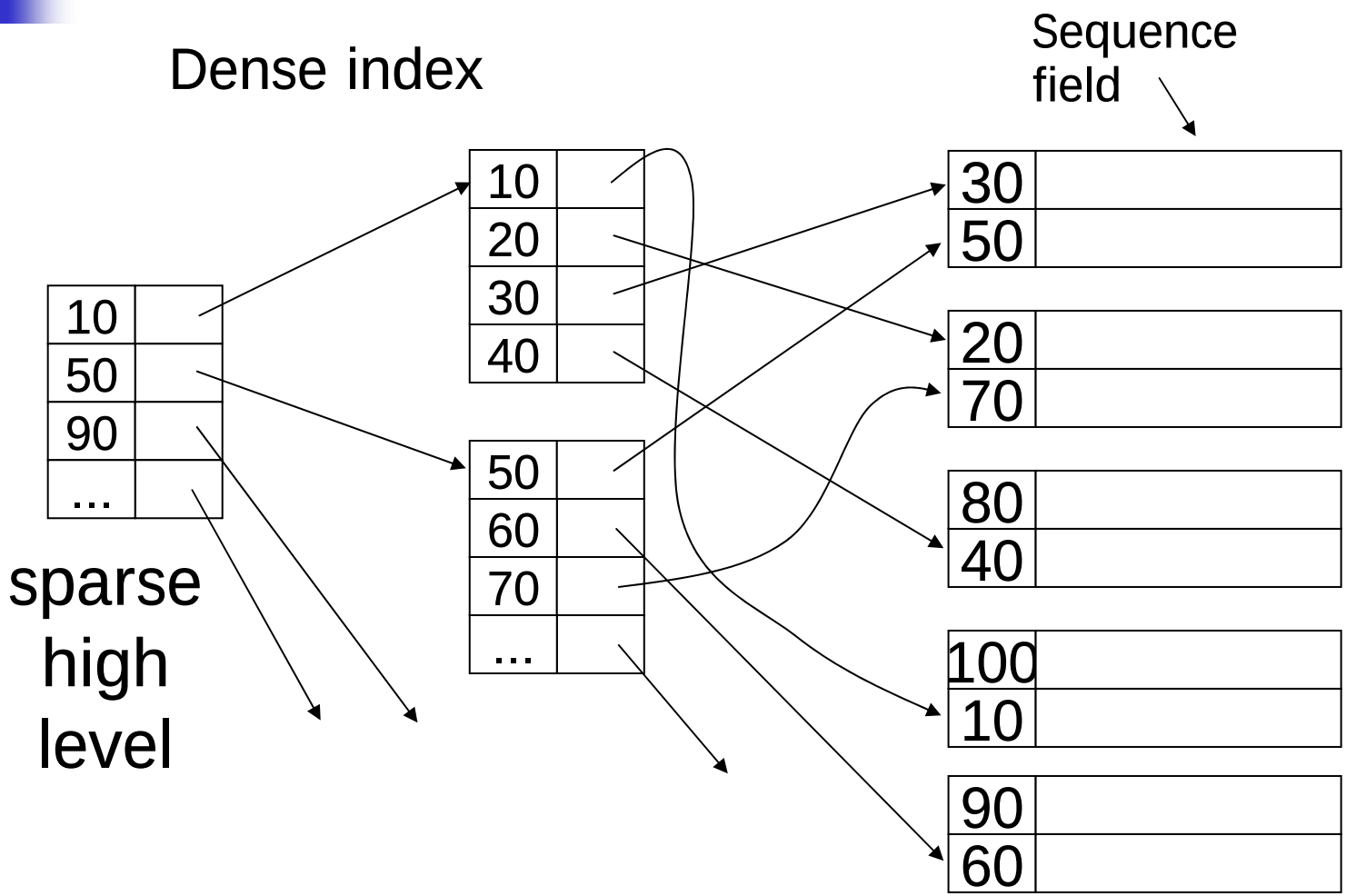
-- Design of Secondary Indexes ...

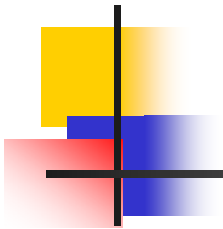


... -- Design of Secondary Indexes ...



... -- Design of Secondary Indexes ...

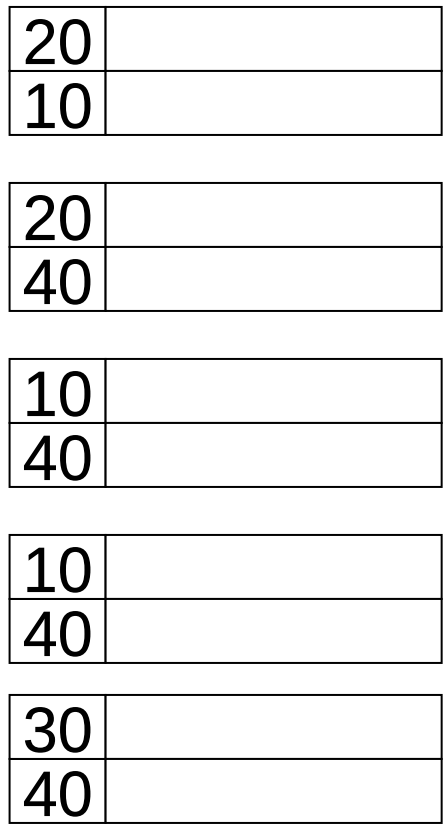




... -- Design of Secondary Indexes

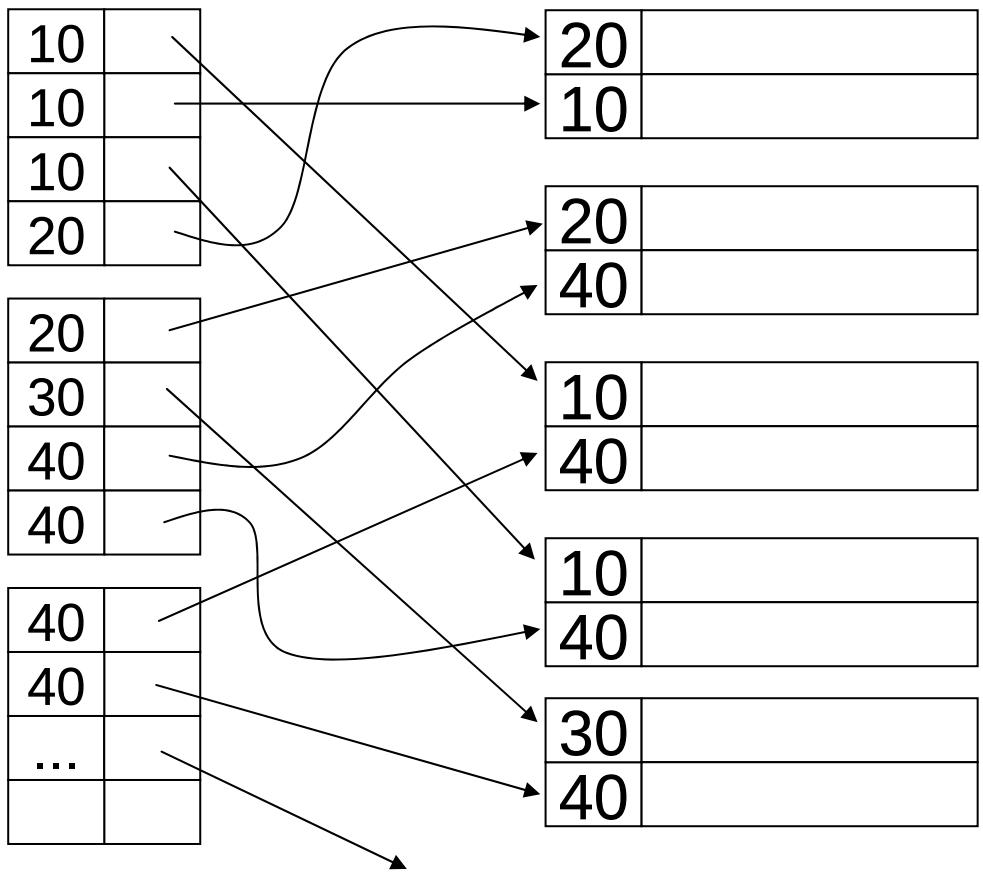
- Lowest level is dense
  - Record pointers
- Other levels are sparse
  - Block pointers

-- Duplicate Values and Secondary Indexes ...



-- Duplicate Values and Secondary Indexes ...

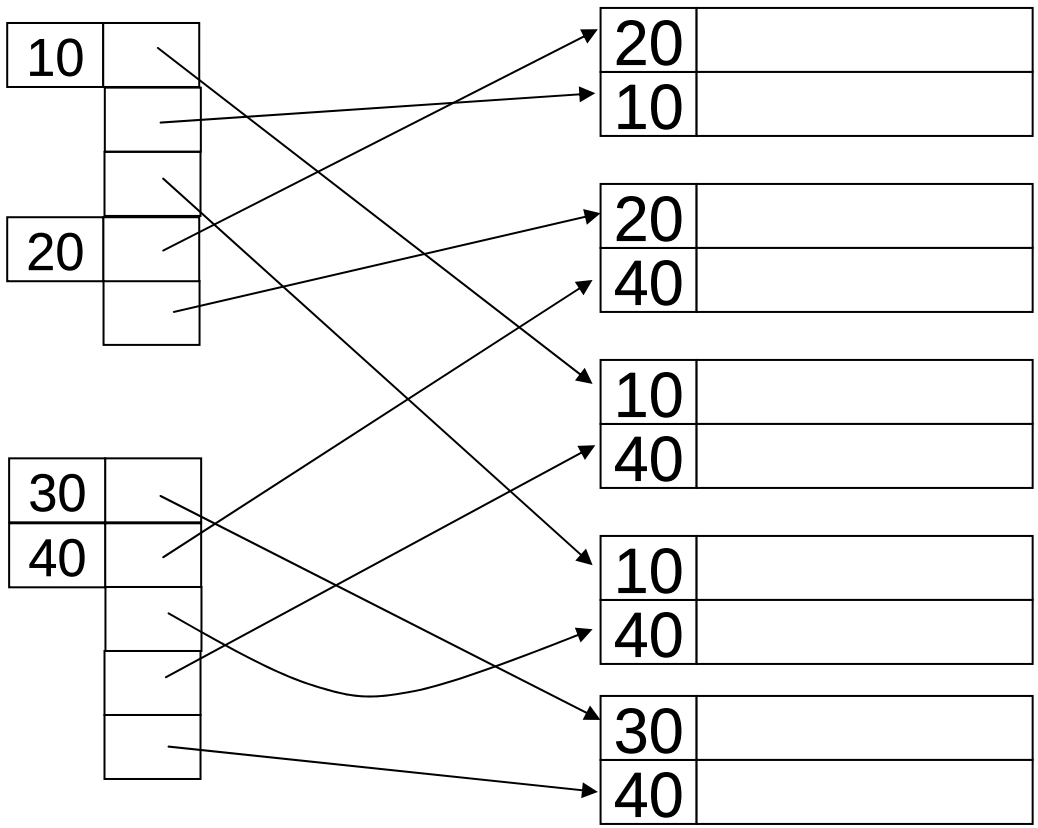
- one option...



Problem:
excess overhead!
- disk space
- search time

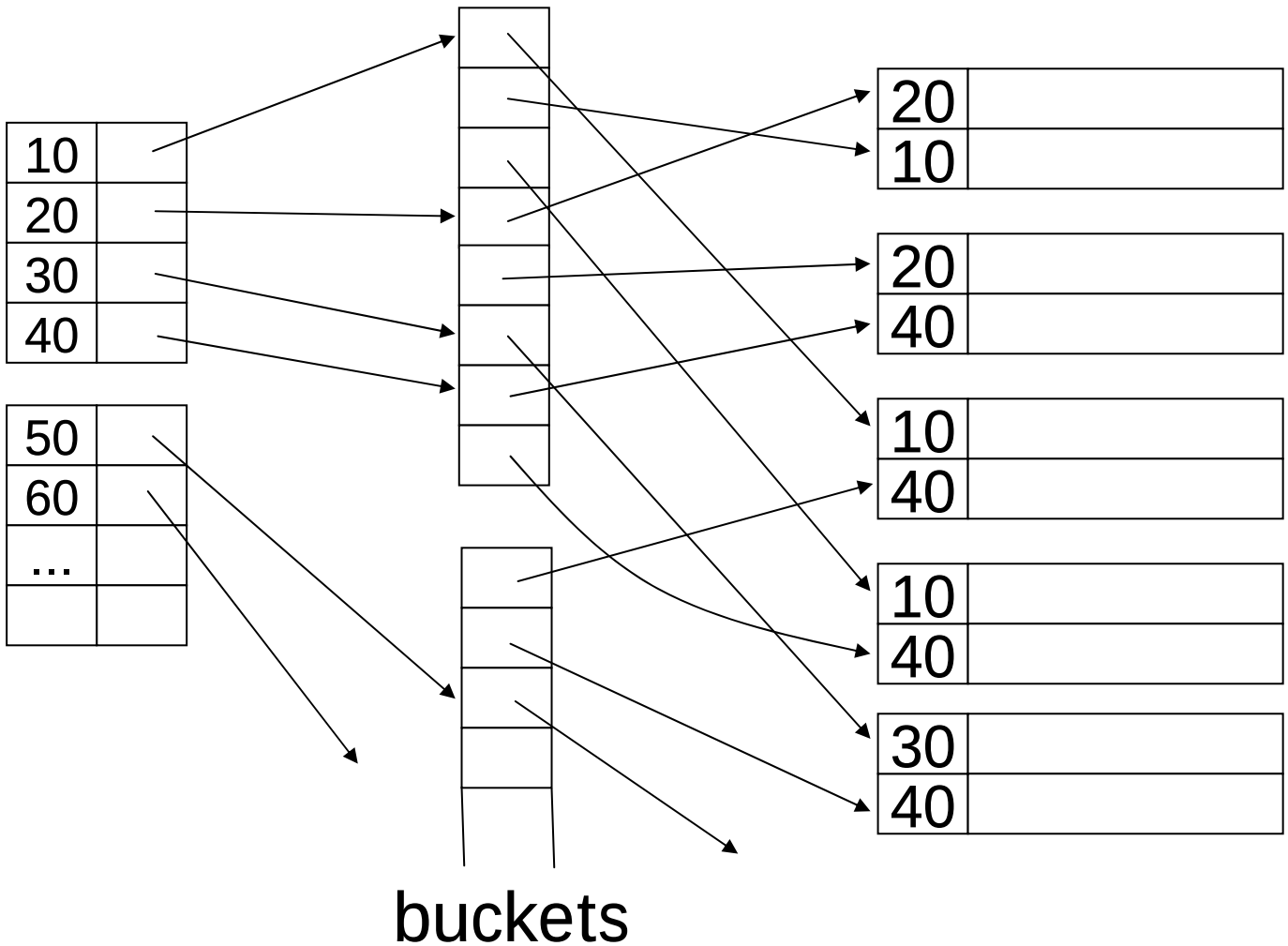
-- Duplicate Values and Secondary Indexes ...

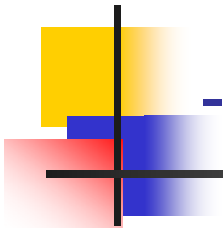
- another option...



Problem:
variable size records in index!

-- Duplicate Values and Secondary Indexes ...



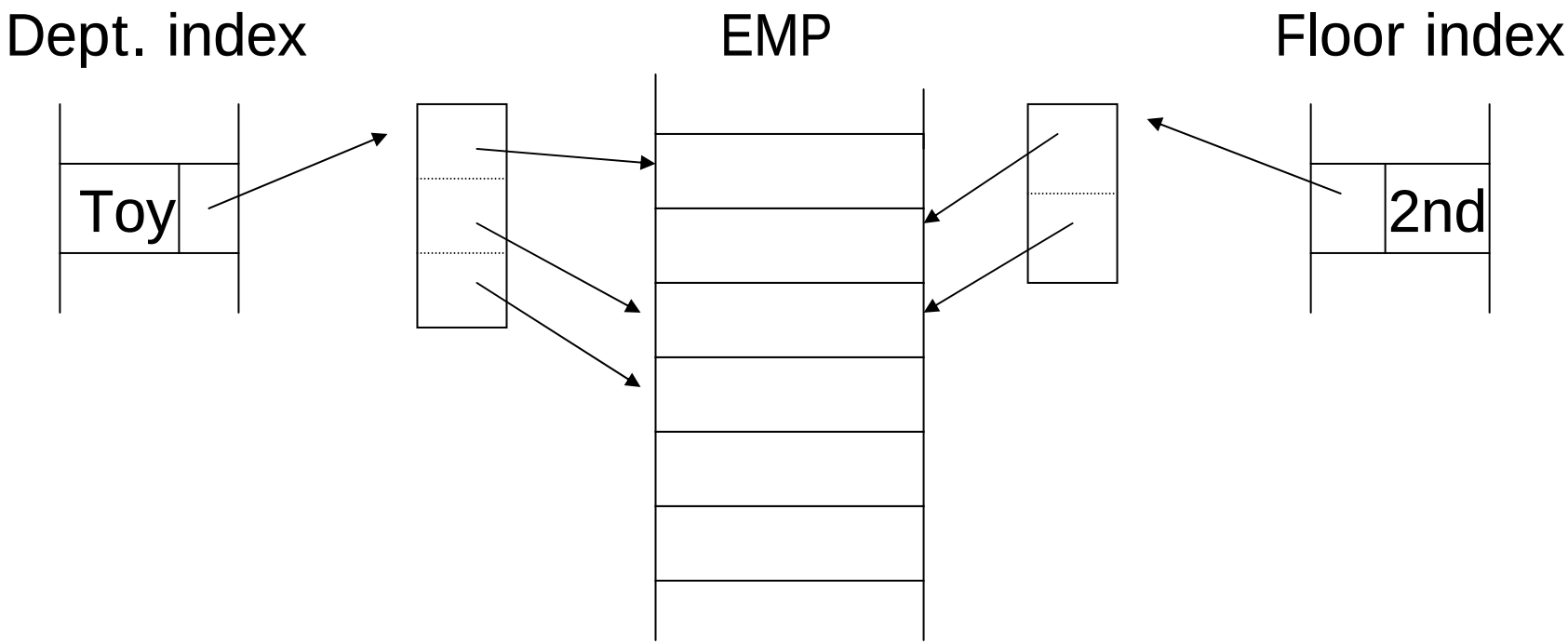


---Why “bucket” idea is useful ...

Indexes	Records
Name: primary	EMP (name,dept,floor,...)
Dept: secondary	
Floor: secondary	

... ---Why "bucket" idea is useful ...

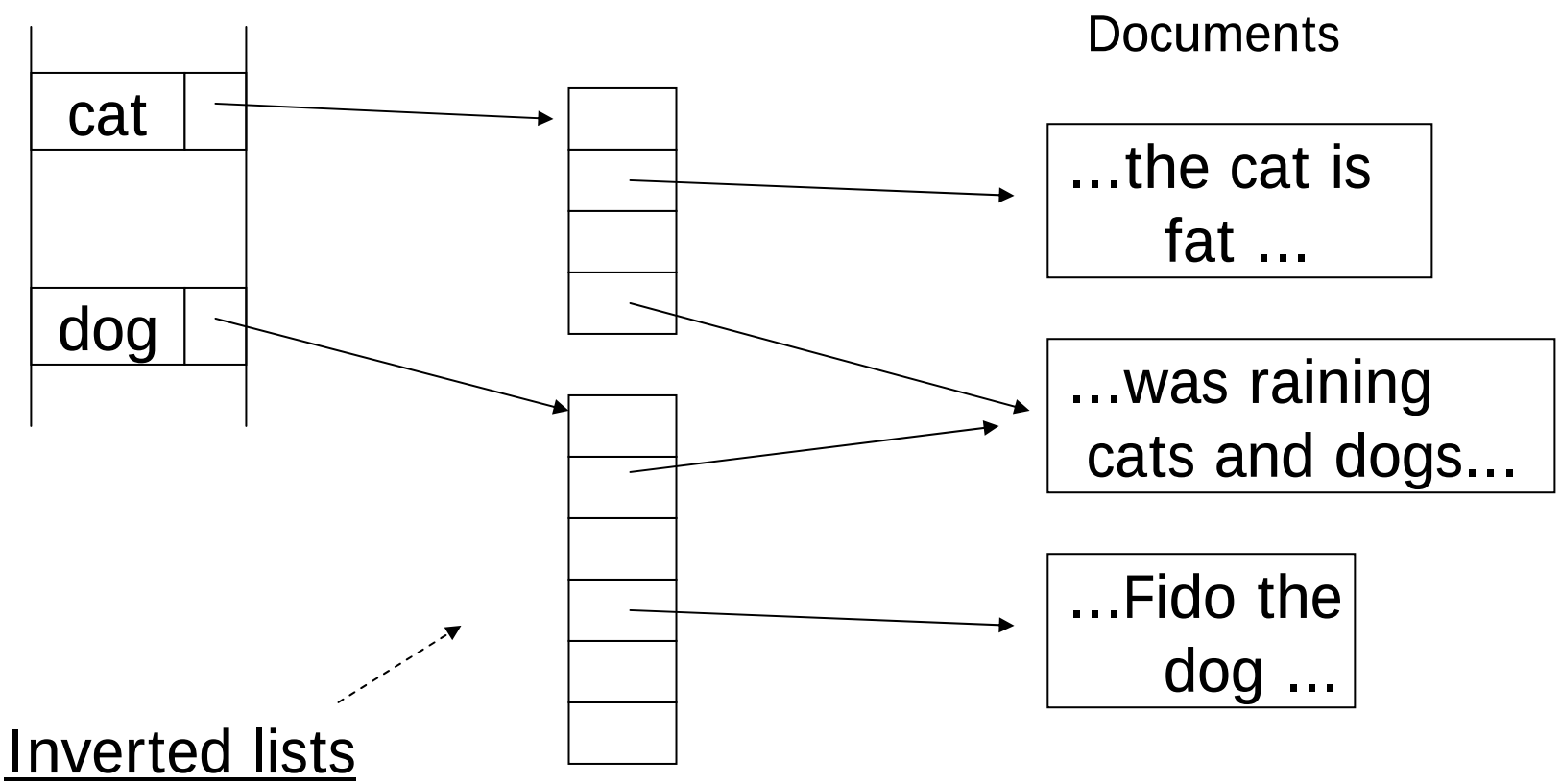
Query: Get employees in (Toy Dept) ∧ (2nd floor)

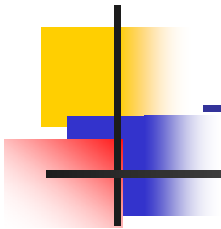


→Intersect toy bucket and 2nd Floor bucket to get set of matching EMP's. Used for text retrieval

... ---Why “bucket” idea is useful

This idea used in text information retrieval.





-- Conventional indexes

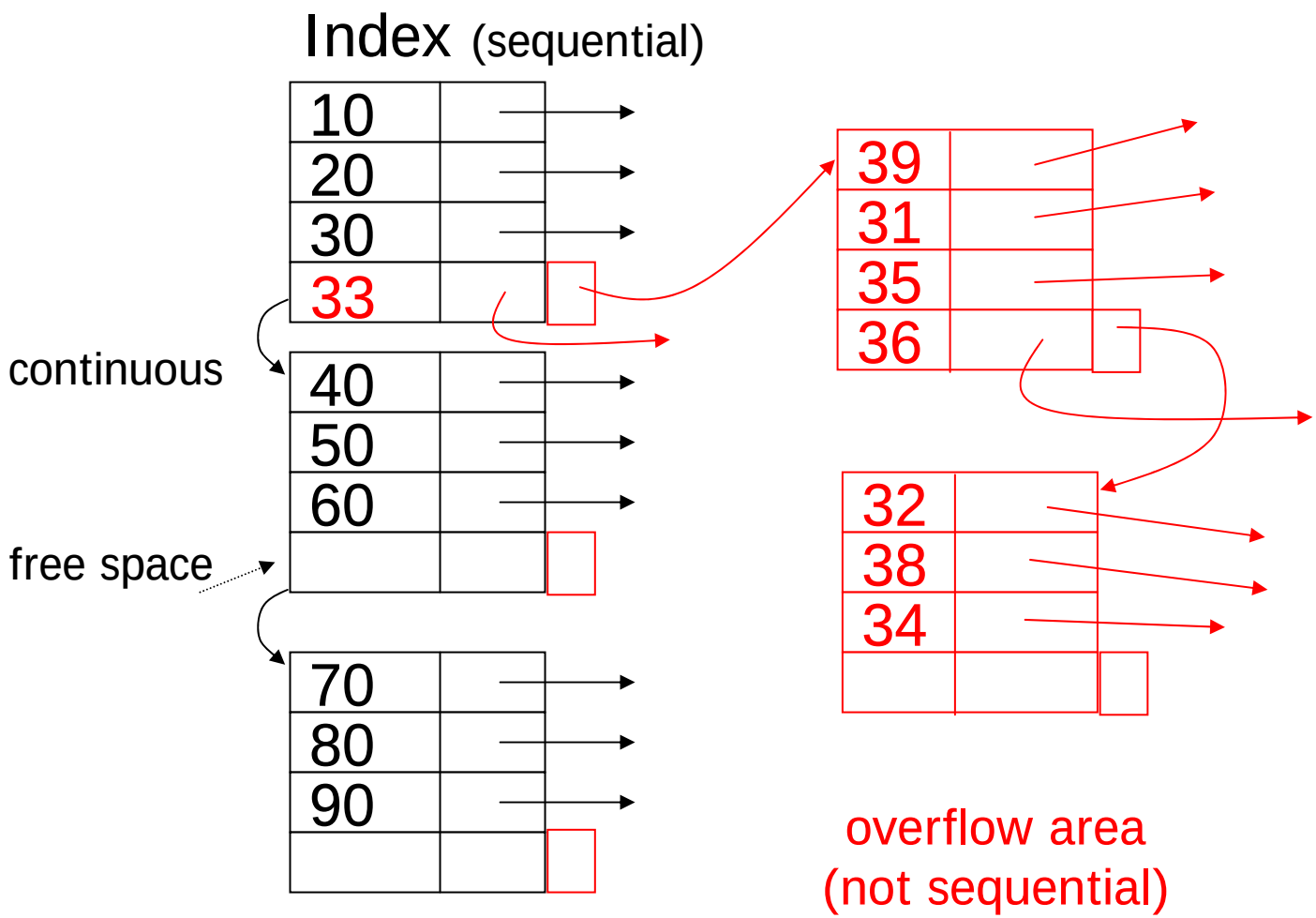
Advantage:

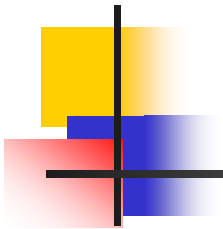
- Simple
- Index is sequential file good for scans

Disadvantage:

- Inserts expensive, and/or
- Lose sequentiality & balance

--- Example

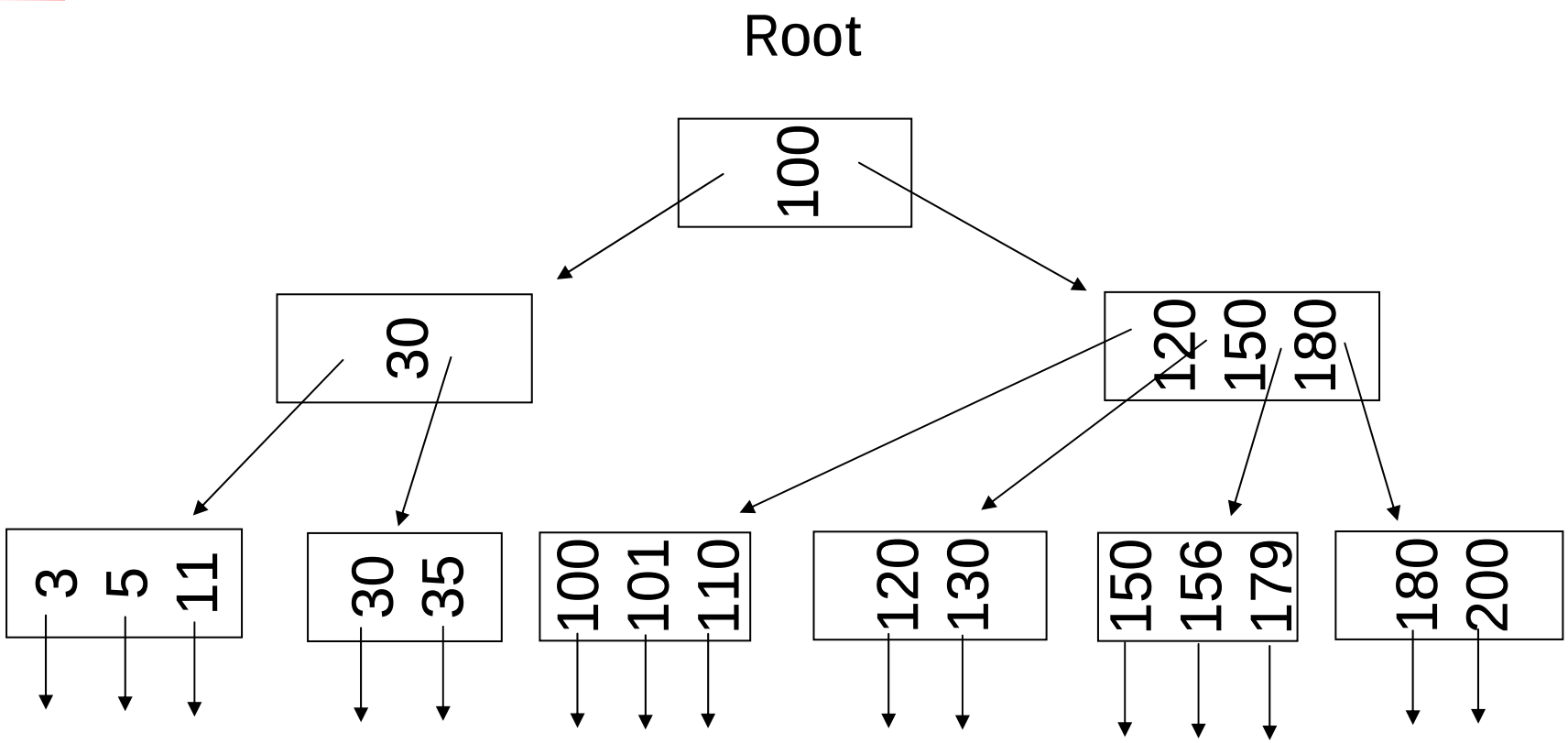

Index (sequential)
10
20
30
33
continuous
40
50
60
free space
70
80
90
39
31
35
36
32
38
34
overflow area
(not sequential)



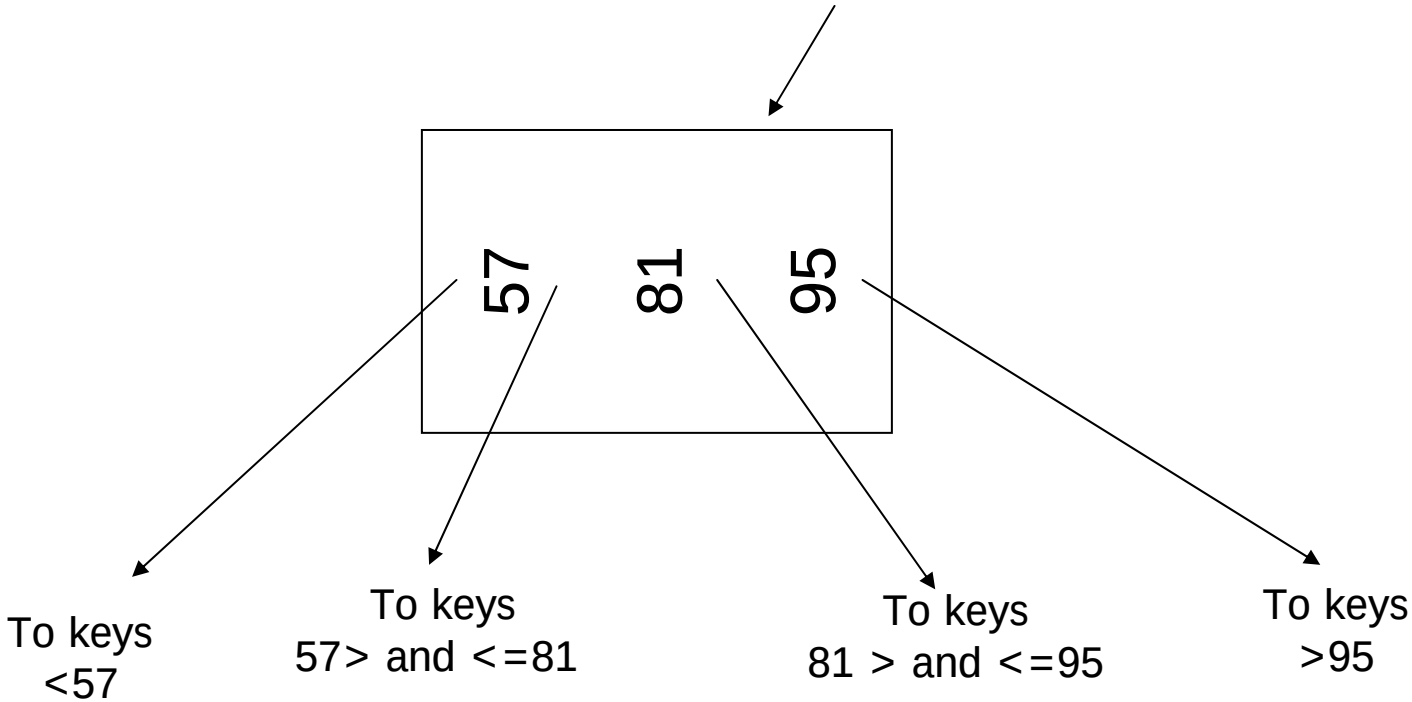
-- Next

- Another type of index
  - Give up on sequentiality of index
  - Try to get “balance”
- Btree
- Has Schemes

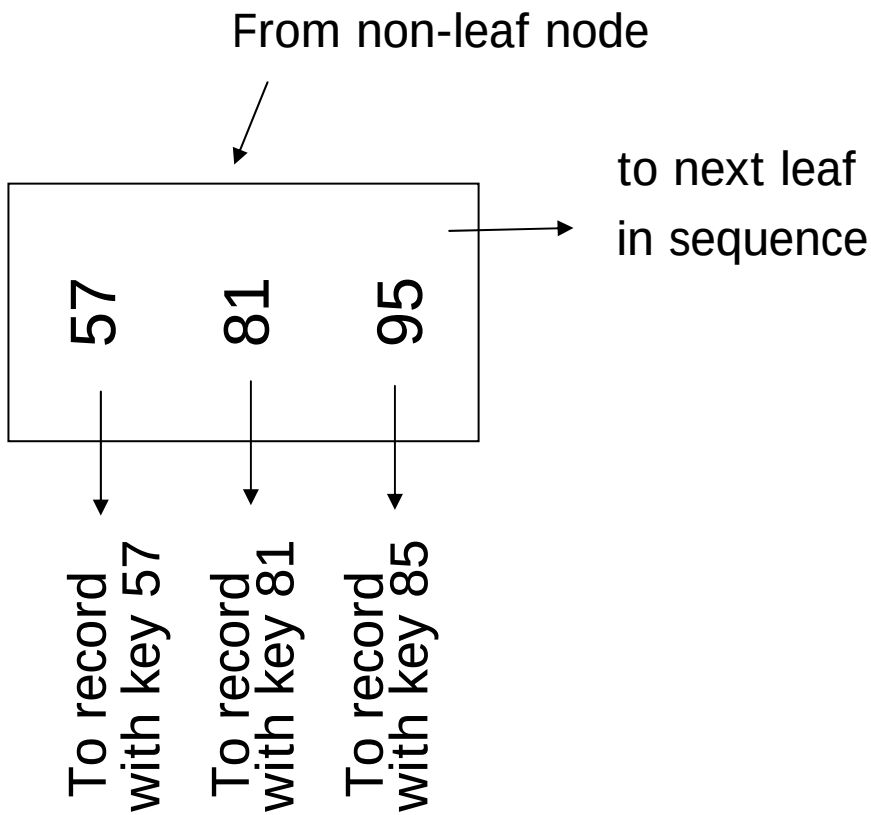
B+Tree Example n=3

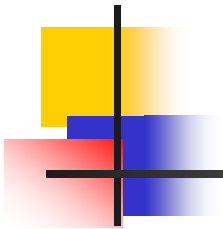


-- Sample non-leaf



-- Sample leaf node:





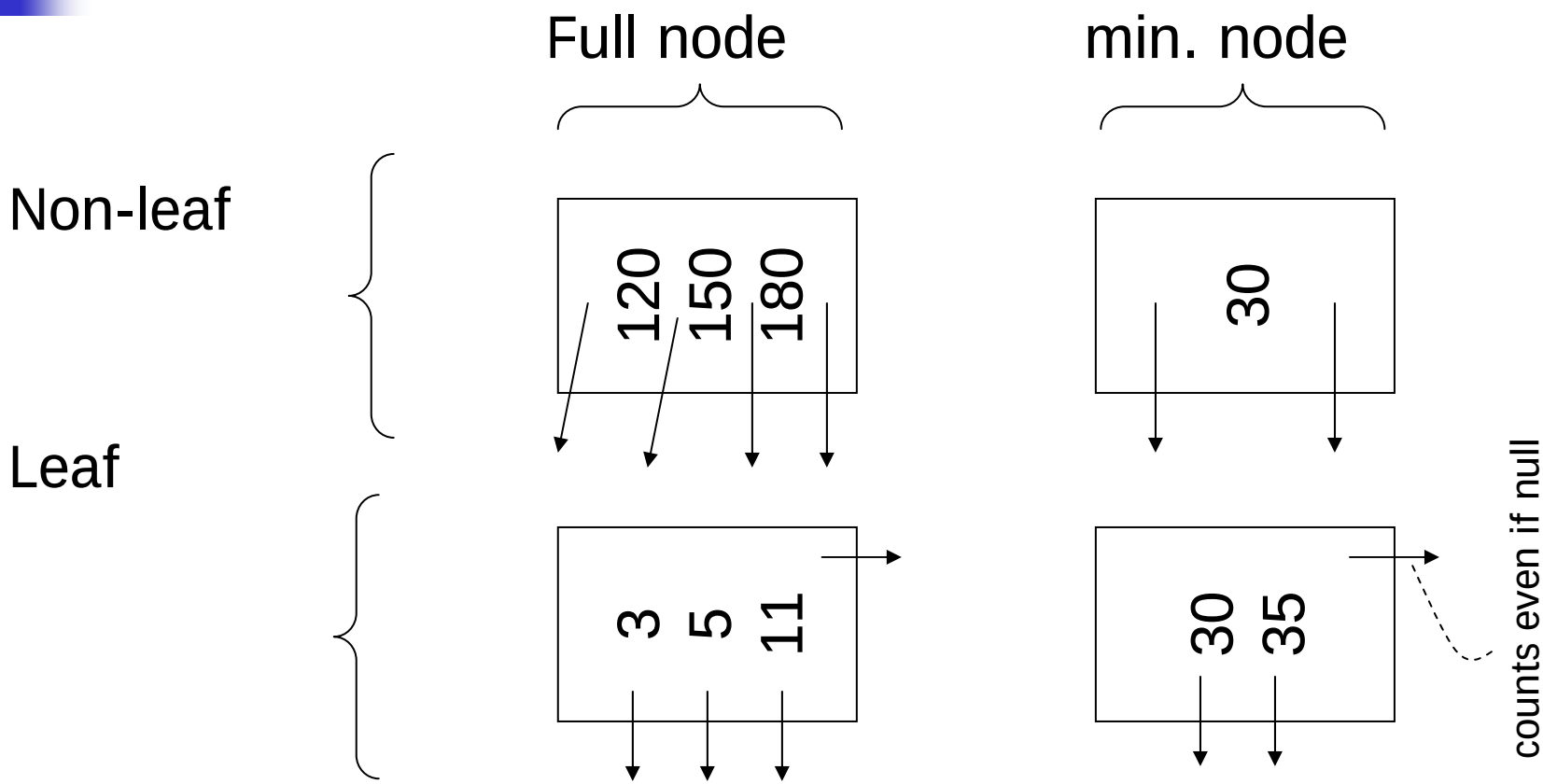
-- Size of Nodes

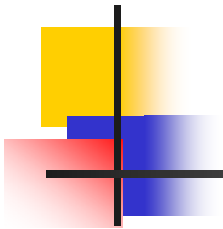
- n keys
- n + 1 Pointers
- Use at least

  Non-leaf: $\lceil (n+1)/2 \rceil$ pointers

  Leaf: $\lfloor (n+1)/2 \rfloor$ pointers to data

--- Example: n = 3

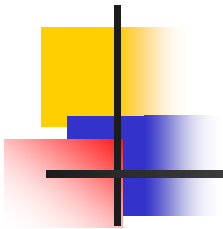

Full node
min. node
Non-leaf
Leaf
120
150
180
30
3
5
11
30
35
counts even if null



-- B+tree rules tree of order *n*

(1) All leaves at same lowest level
(balanced tree)

(2) Pointers in leaves point to records
except for “sequence pointer”



-- Insert into B+tree

(a) simple case

- space available in leaf

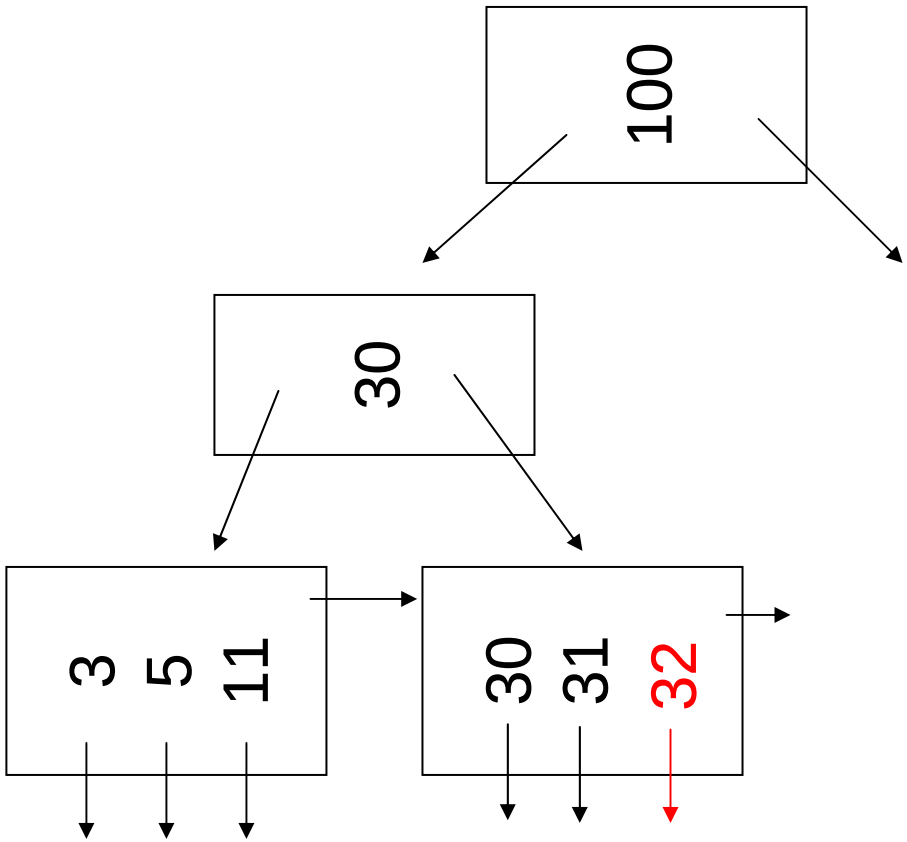
(b) leaf overflow

(c) non-leaf overflow

(d) new root

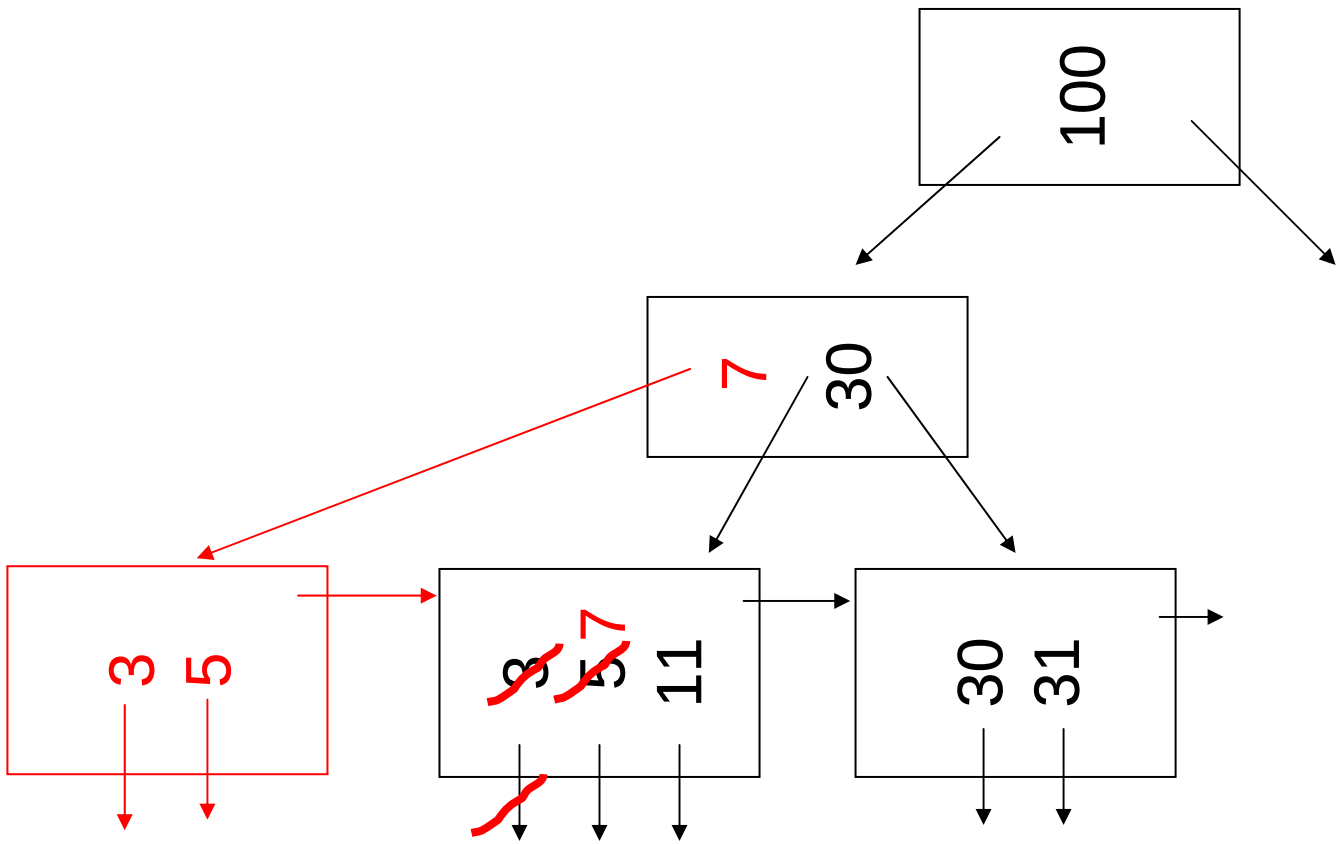
--- Insert key = 32

n=3



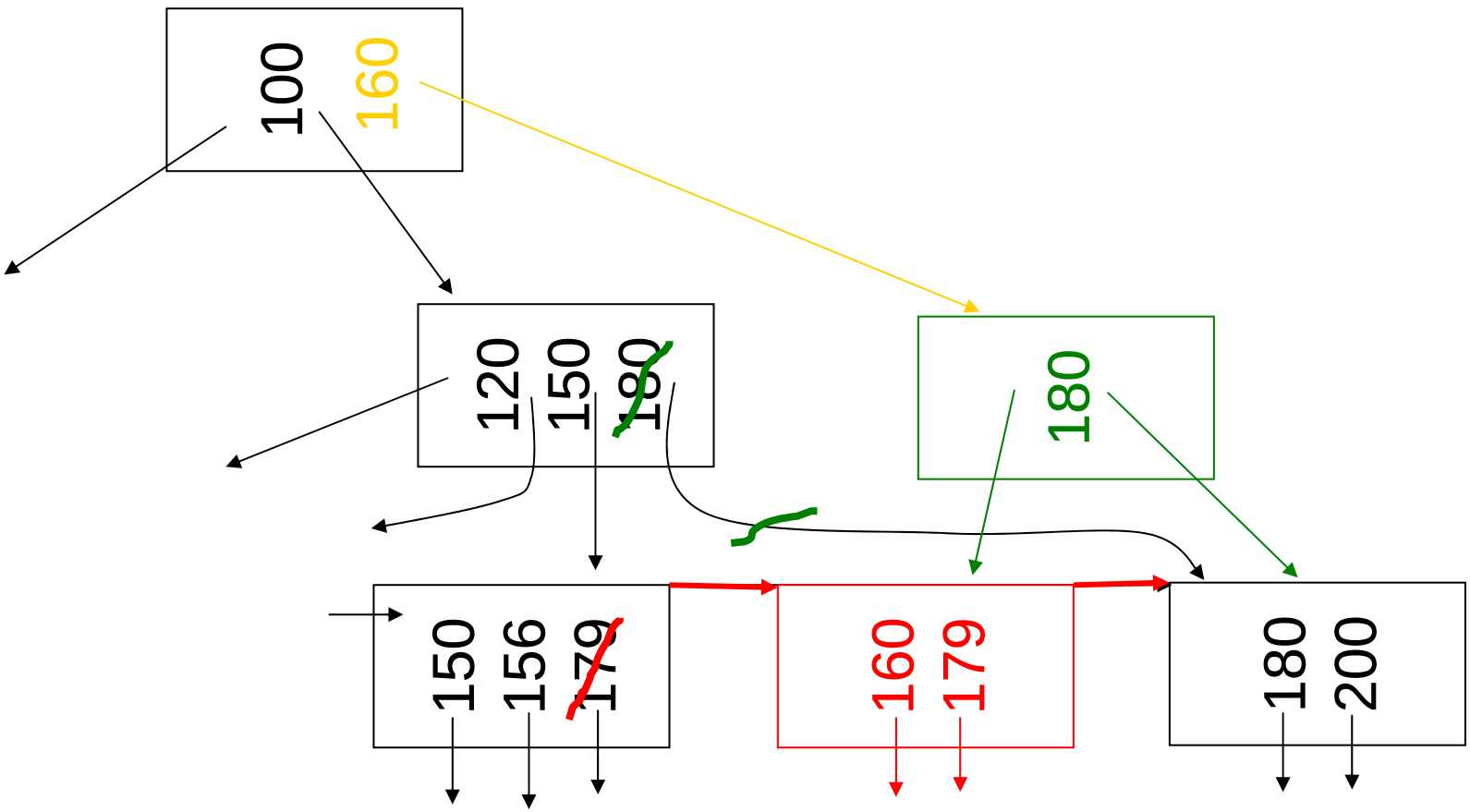
--- Insert key = 7

n=3



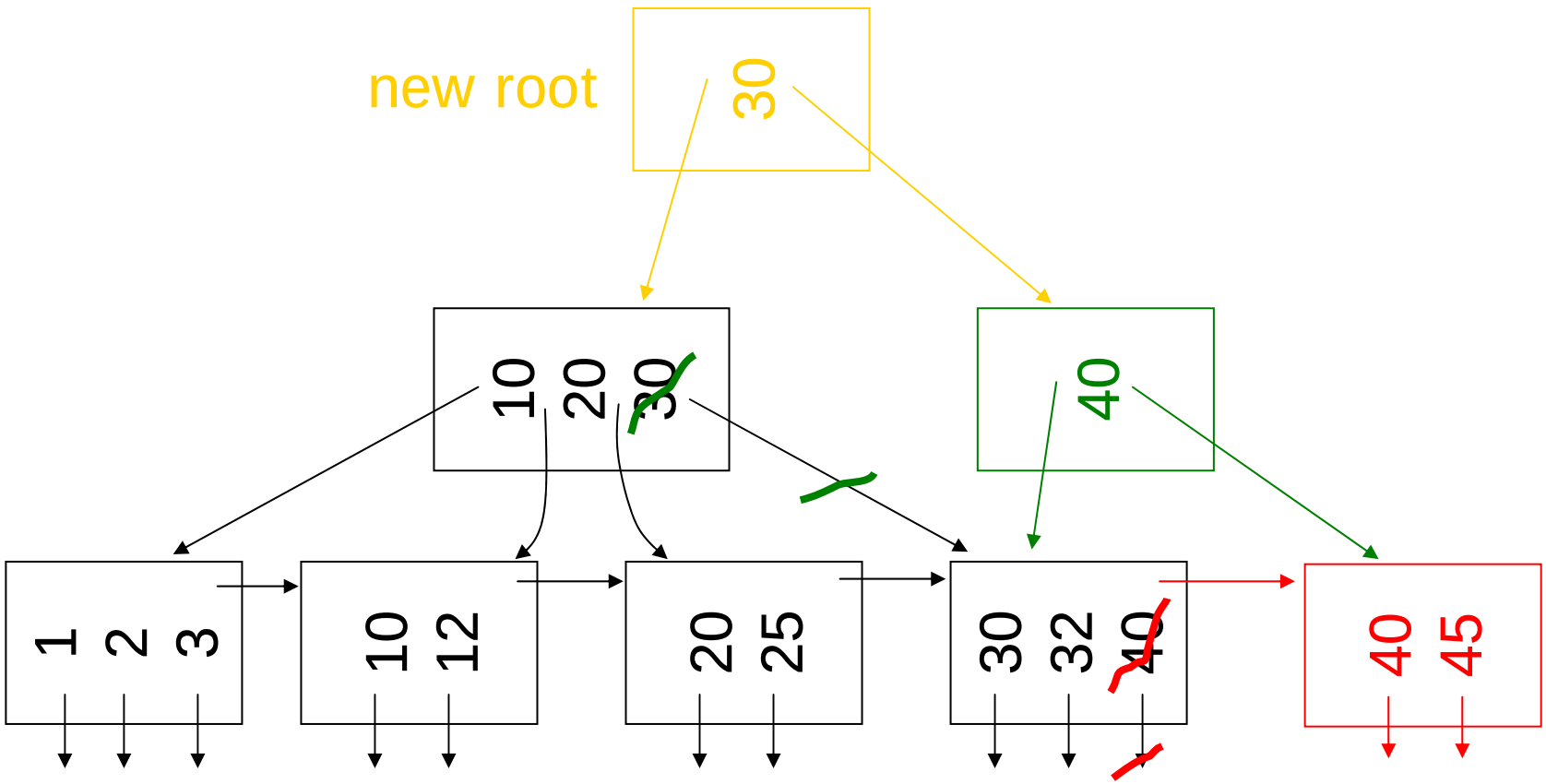
--- Insert key = 160

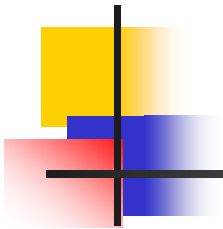
n=3



--- New root, insert 45

n=3





-- Deletion from B+tree

(a) Simple case - no example

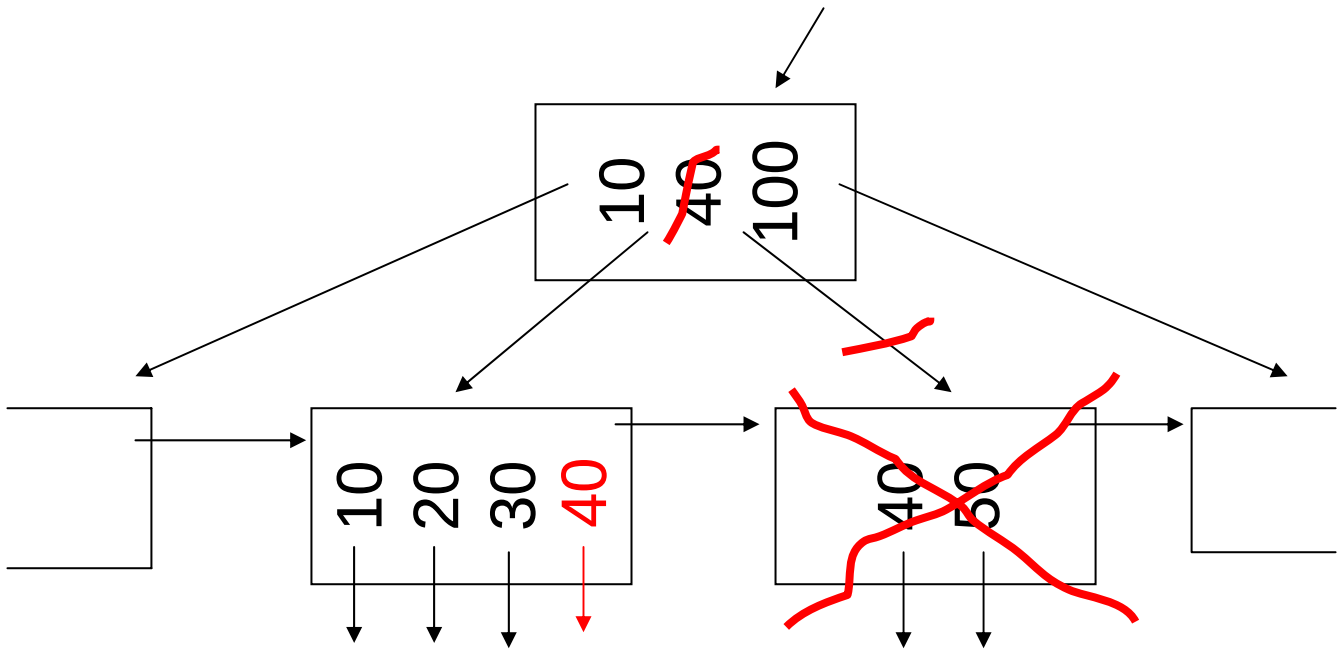
(b) Coalesce with neighbor (sibling)

(c) Re-distribute keys

(d) Cases (b) or (c) at non-leaf

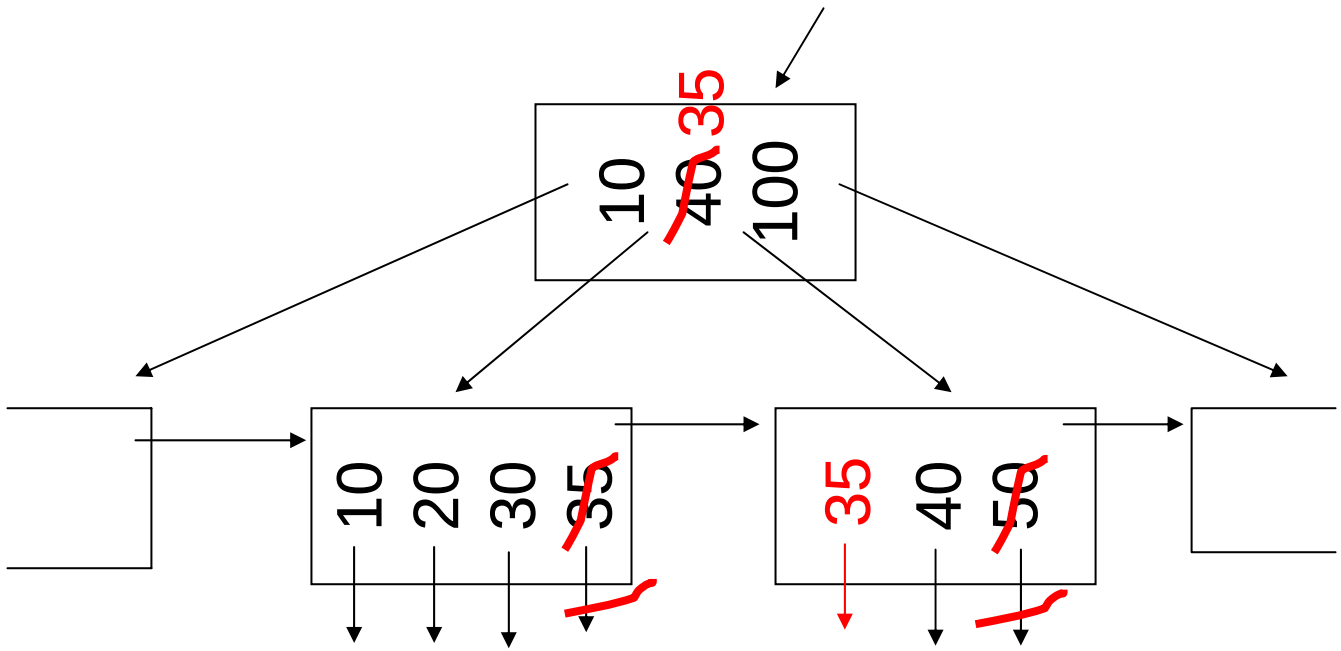
--- Coalesce with sibling: Delete 50

n=4



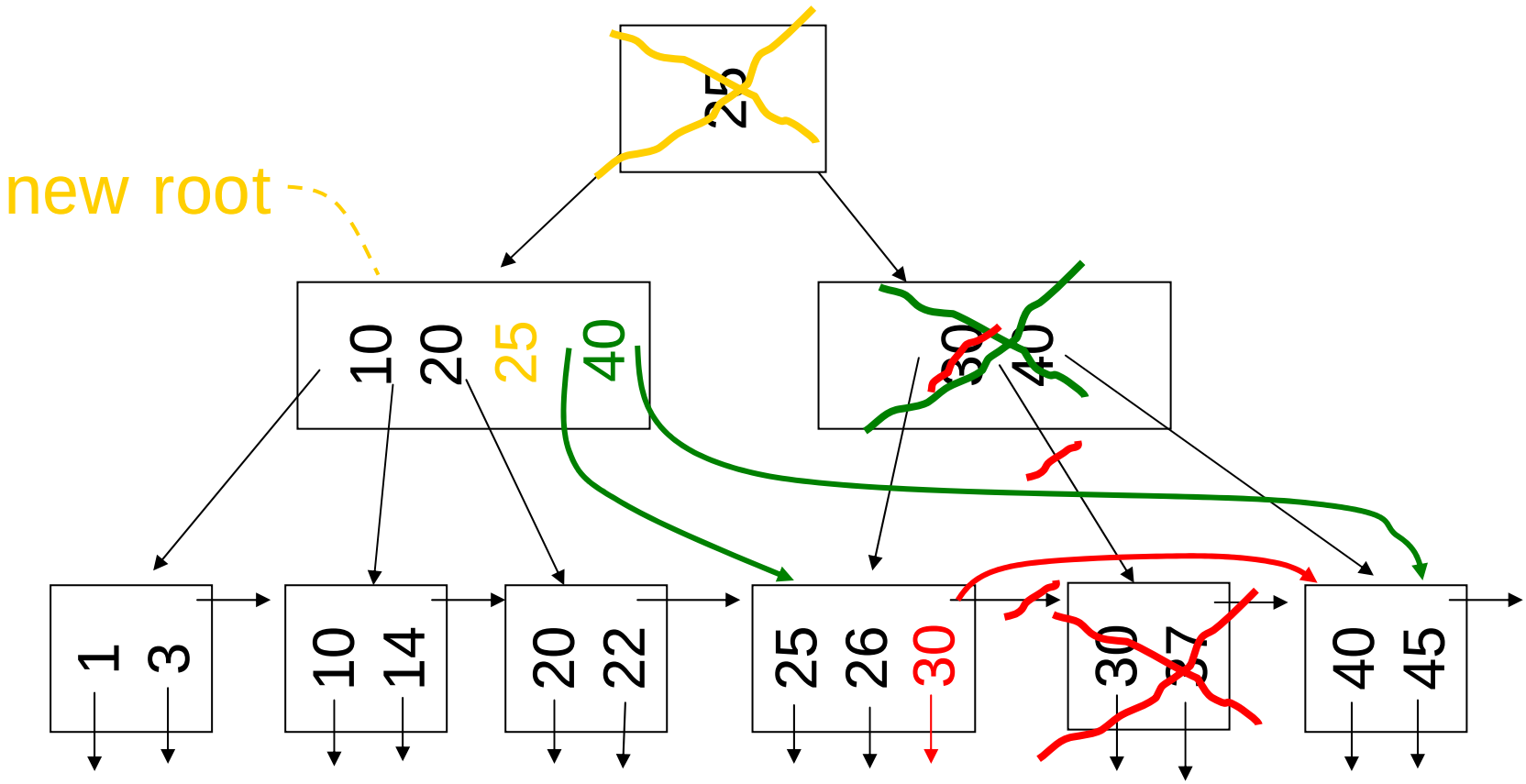
--- Redistribute key: Delete 50

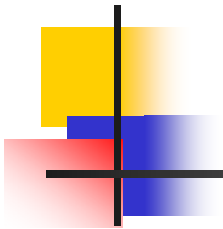
n=4



--- Non-leaf coalesce: Delete 37

n=4



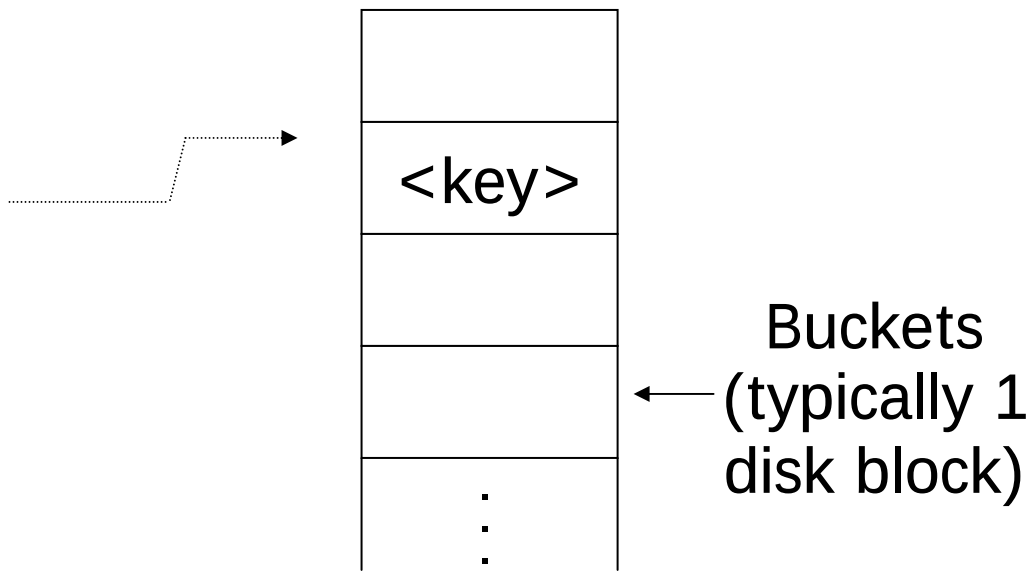


--- B+tree deletions in practice

- Often, coalescing is not implemented
  - Too hard and not worth it!

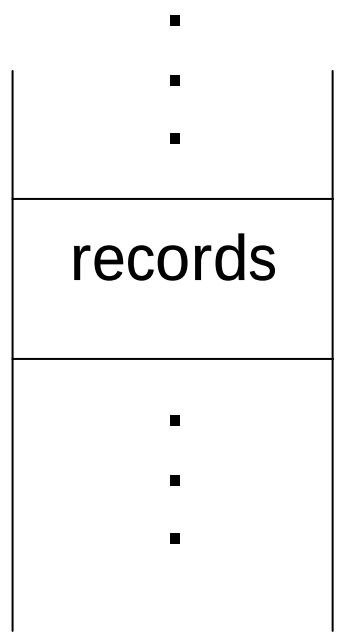
- Hashing

key $\rightarrow$ h(key)



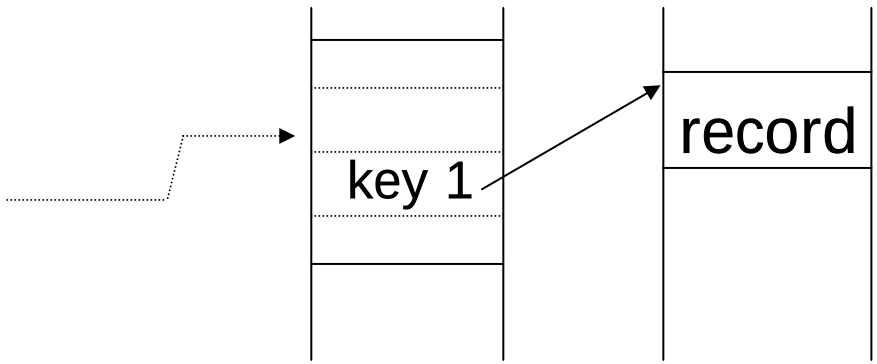
-- Two alternatives ...

(1) key $\rightarrow$ h(key)

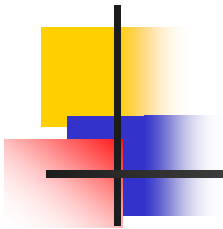


... -- Two alternatives

(2) key $\rightarrow$ h(key)

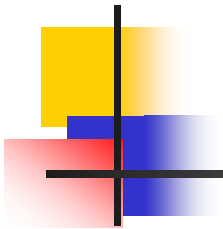


- Alt (2) for “secondary” search key



-- Example hash function ...

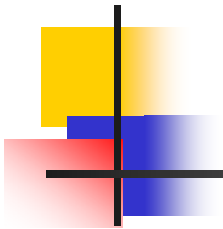
- Key = ‘x_1 x_2 ... x_n’ *n* byte character string
- Have *b* buckets
- h: add x_1 + x_2 + ... X_n
  - compute sum modulo *b*



... -- Example hash function

- This may not be best function ...
- Read Knuth Vol. 3 if you really need to select a good function.

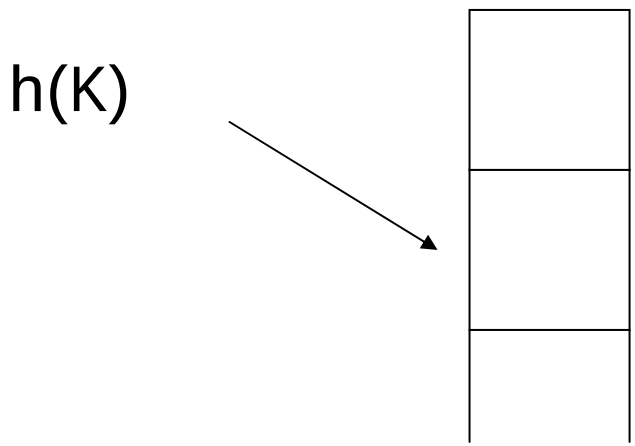
Good hash function: ☞ Expected number of keys/bucket is the same for all buckets



-- Within a bucket

- Do we keep keys sorted?
  - Yes
    - if CPU time critical, and
    - Inserts/Deletes not too frequent

-- Example to illustrate inserts, overflows, deletes



... -- Example to illustrate insert ...

INSERT:

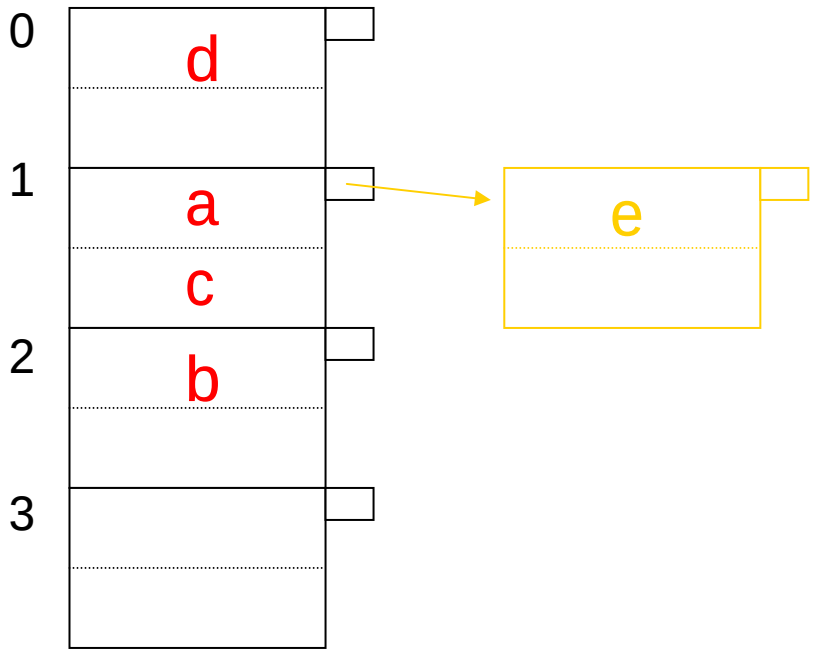
h(a) = 1

h(b) = 2

h(c) = 1

h(d) = 0

h(e) = 1



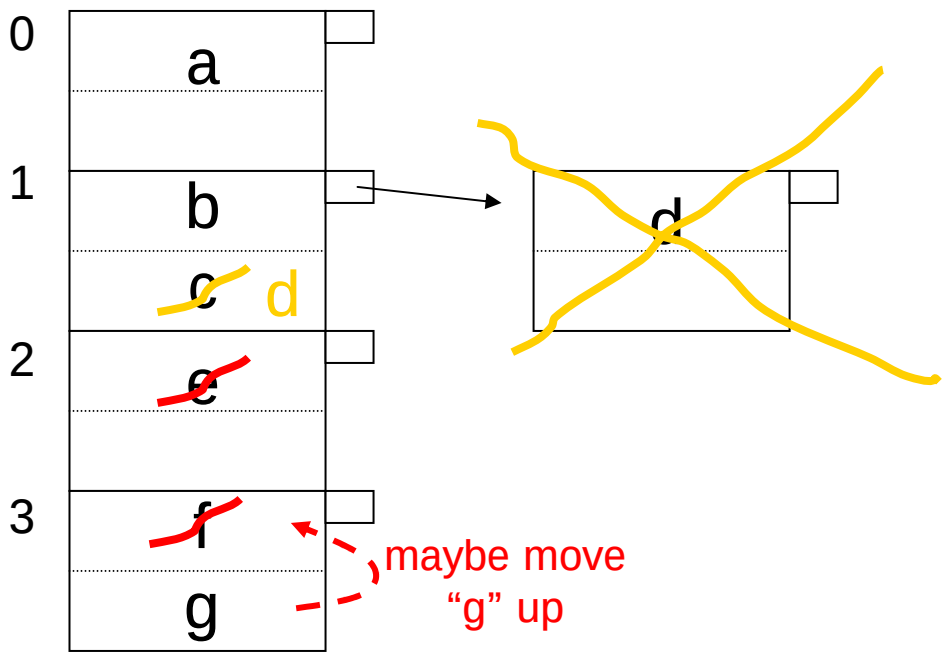
... -- Example to illustrate delete ...

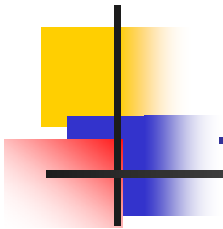
Delete:

e

f

c



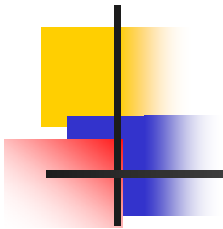


--- Rule of thumb

- Try to keep space utilization between 50% and 80%

  $$\text{Utilization} = \frac{\text{\# keys used}}{\text{total \# keys that fit}}$$

  - If < 50%, wasting space
  - If > 80%, overflows significant
    depends on how good hash function is & on # keys/bucket

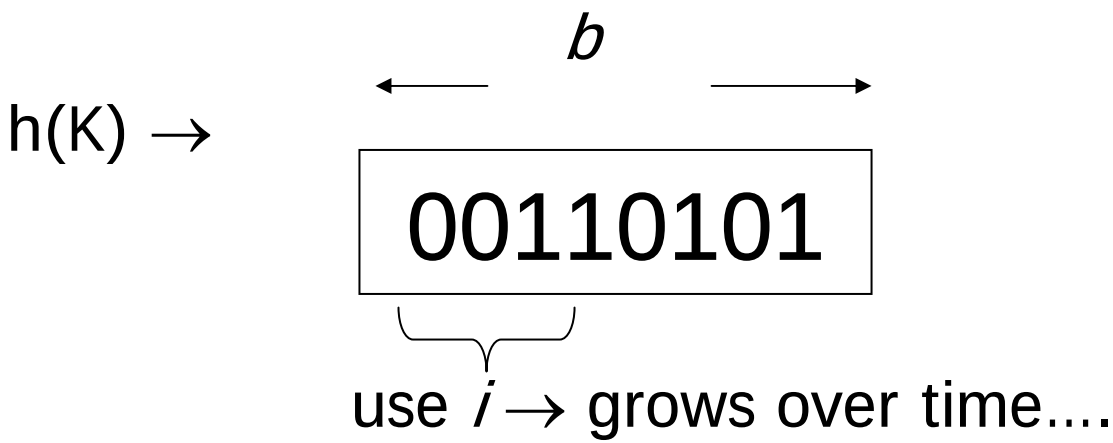


-- How do we cope with growth?

- Static hashing
  - Overflows and reorganizations
    - Very expensive
- Solution
  - Dynamic hashing
    - Extensible
    - Linear

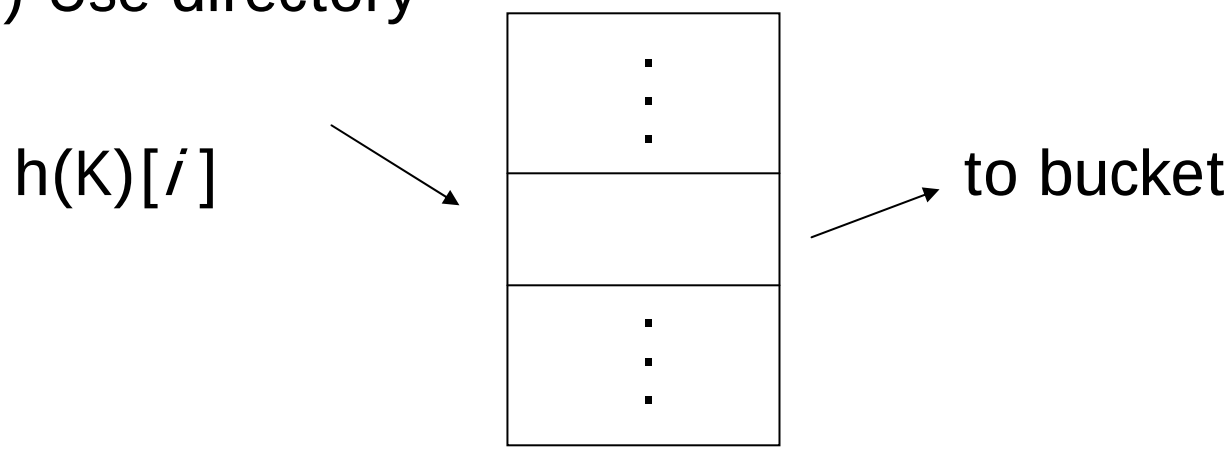
-- Extensible hashing: two ideas ..

(a) Use *i* of *b* bits output by hash function

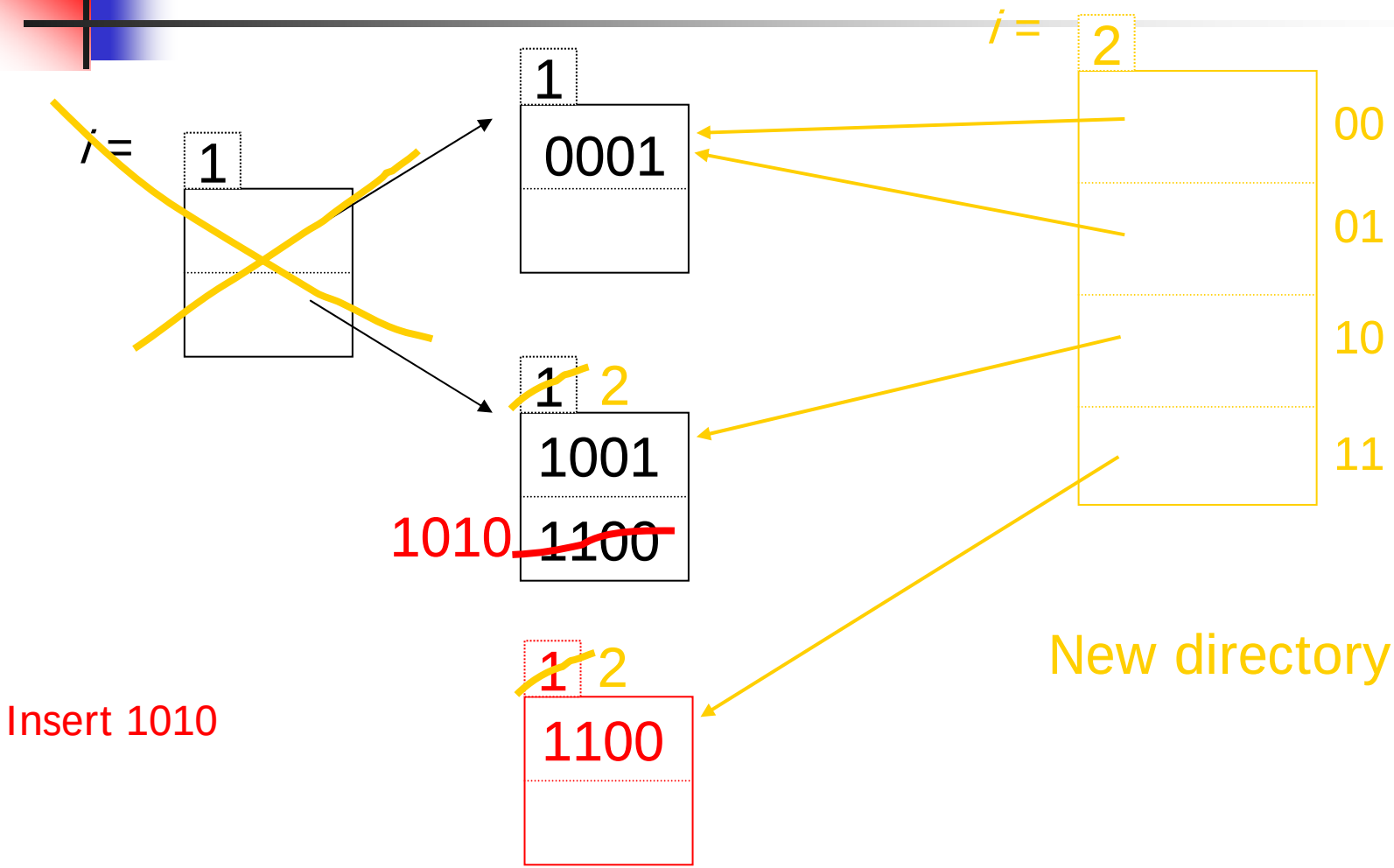


... -- Extensible hashing: two ideas

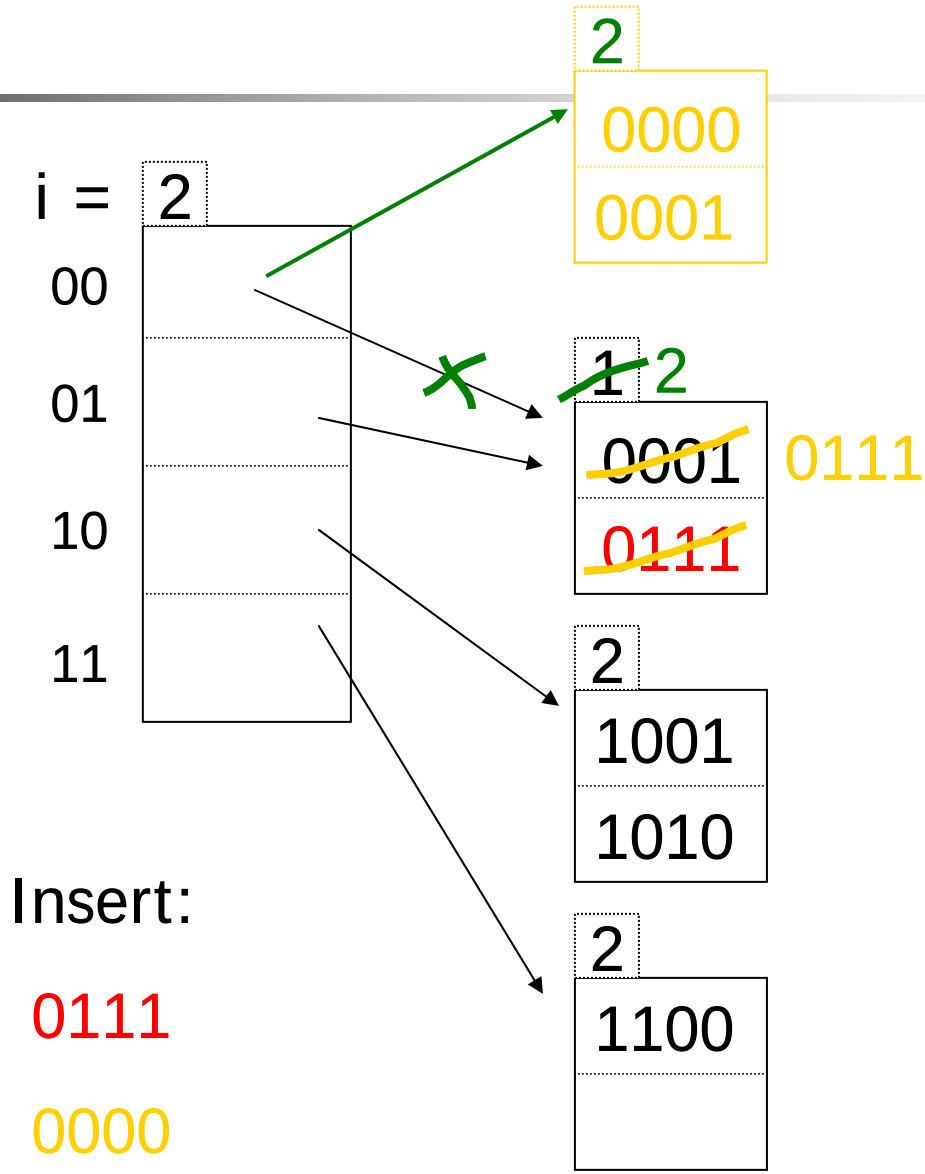
(b) Use directory



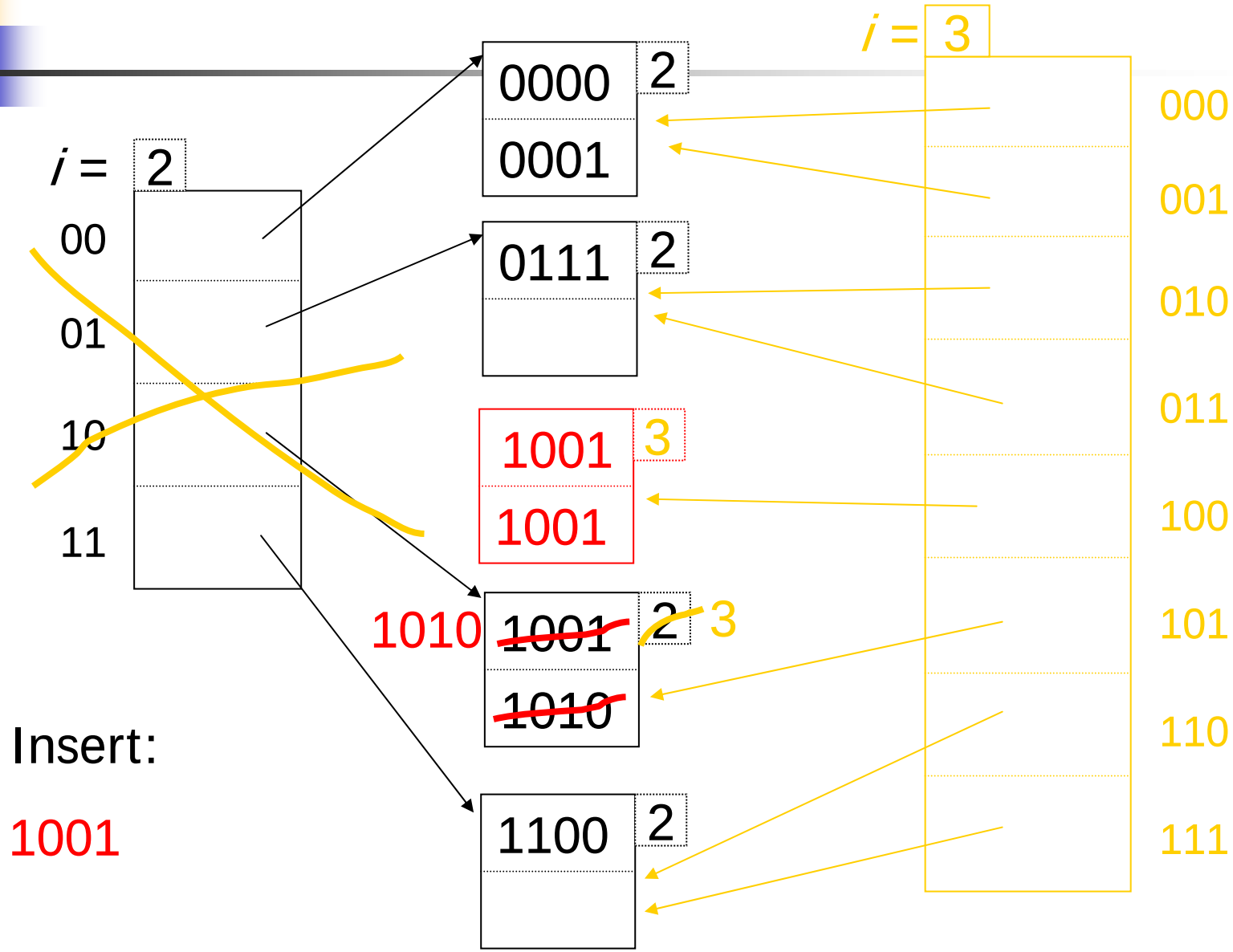
--- Example: h(k) is 4 bits; 2 keys/bucket ...

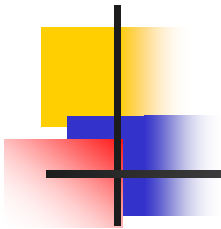


... --- Example: h(k) is 4 bits; 2 keys/bucket ...



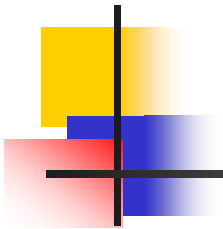
... --- Example: h(k) is 4 bits; 2 keys/bucket





--- Extensible hashing: deletion

- Two option:
  - No merging of blocks
  - Merge blocks and cut directory if possible



---- Deletion example

- Run thru insert example in reverse!

-- Summary: Extensible hashing

(+) Can handle growing files

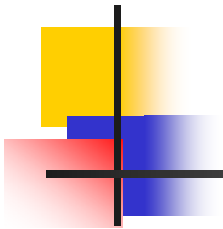
- No full reorganizations

(-) Indirection

(Not bad if directory in memory)

(-) Directory doubles in size

(Now it fits, now it does not)



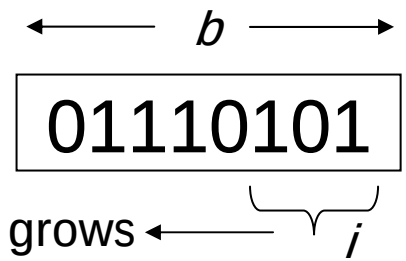
-- Summary: Extensible hashing

- Advantage
  - No reorganization is needed
  - One disk access per record
- Disadvantage
  - Doubling bucket array is expensive
  - The size of the bucket array may no longer fit into memory
  - The number of bucket may be much bigger than the blocks
    - Example: Splitting records can only be done in higher bits.

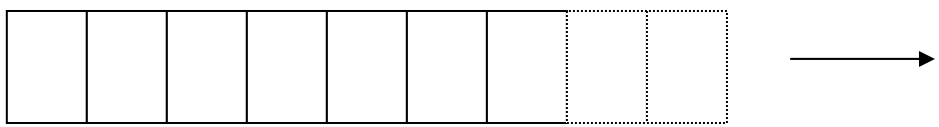
-- Linear hashing

- Another dynamic hashing scheme
- Two ideas:

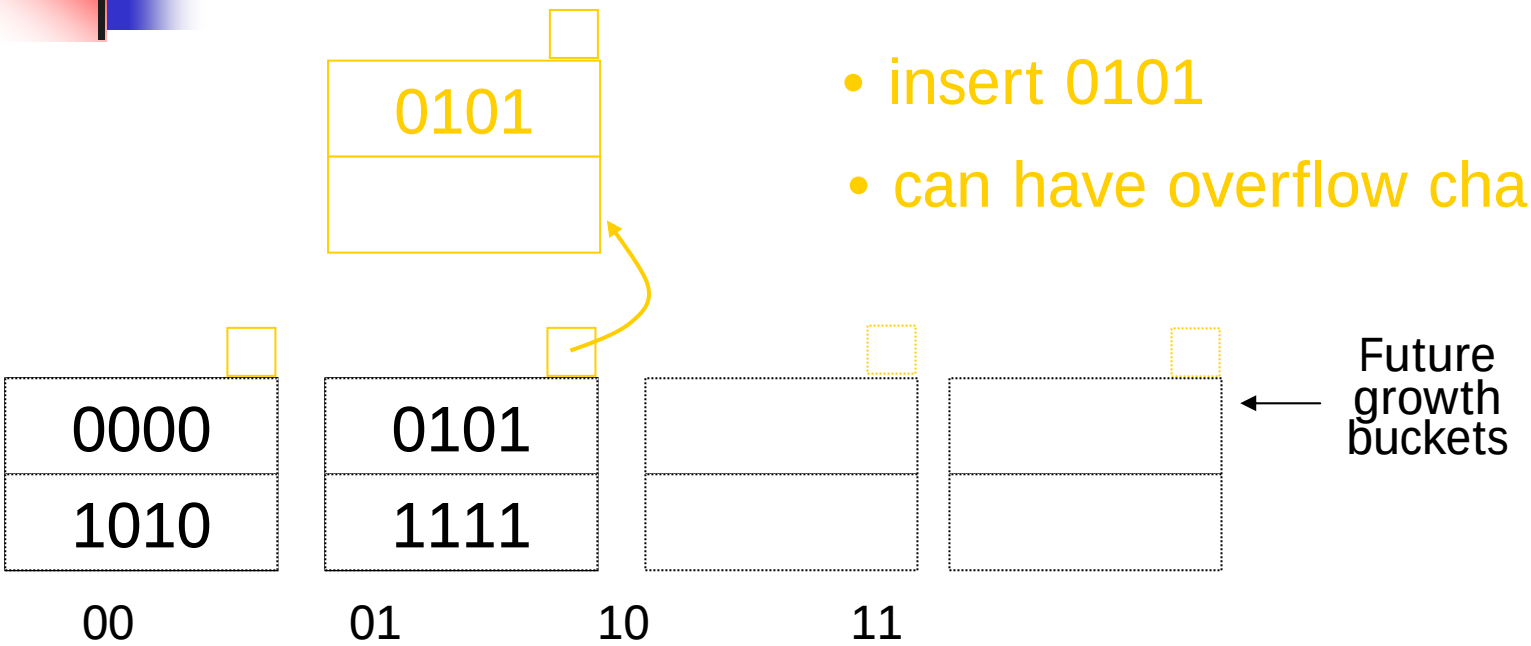
(a) Use *i* low order bits of hash



(b) File grows linearly



Example b=4 bits, i=2, 2 keys/bucket



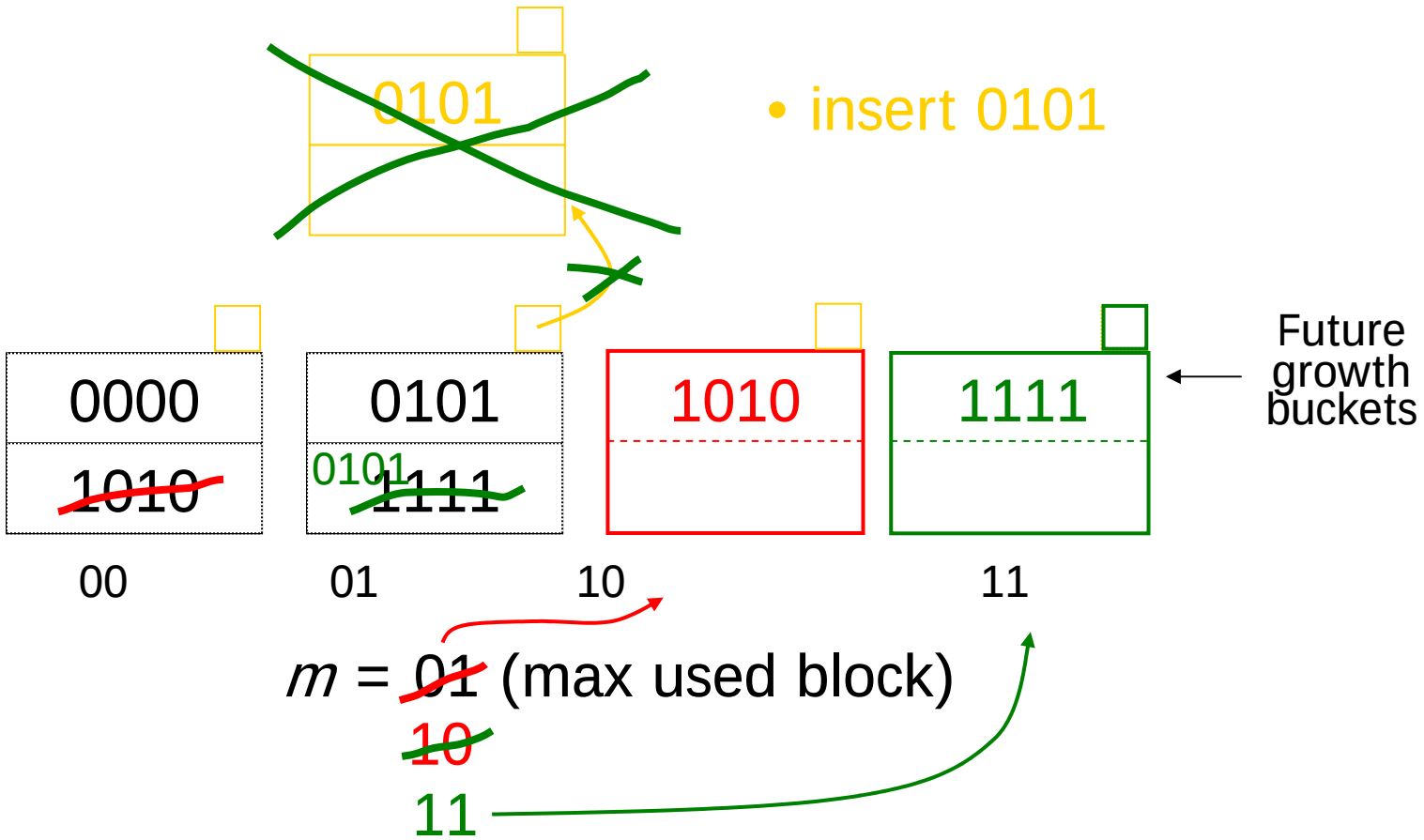
- insert 0101
- can have overflow chains!

m = 01 (max used block)

Rule

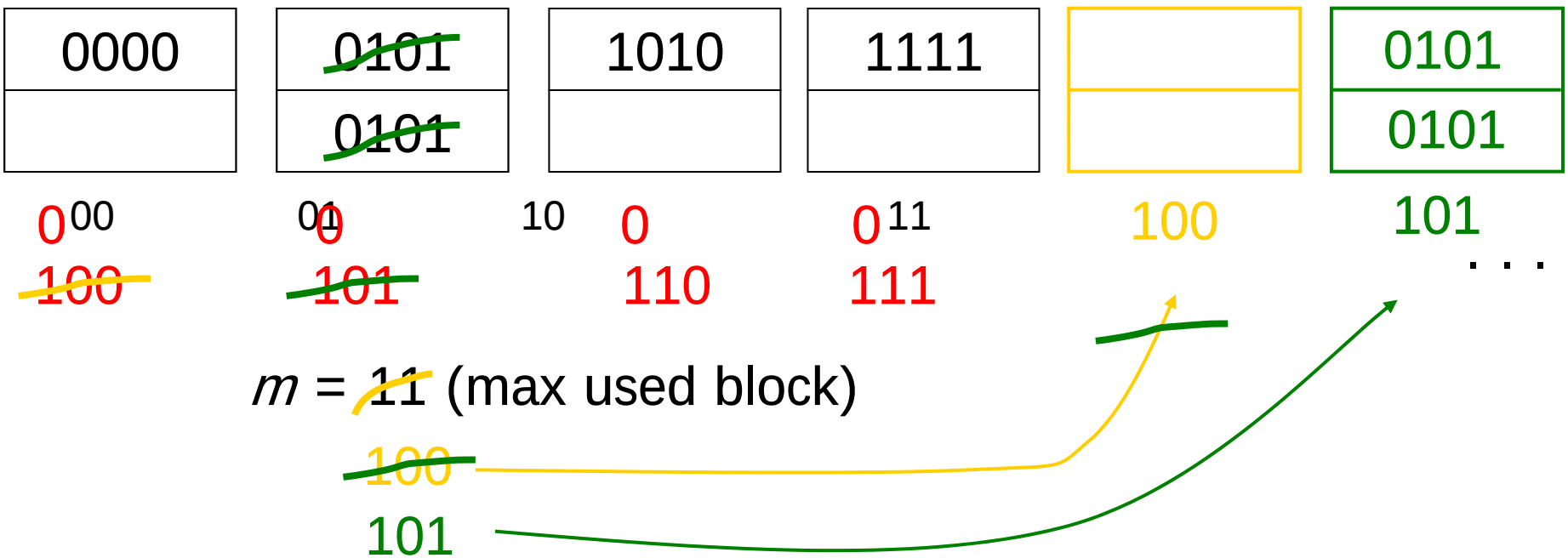
If h(k)[i] ≤ m, then
look at bucket h(k)[i]
else, look at bucket h(k)[i] - 2^{i-1}

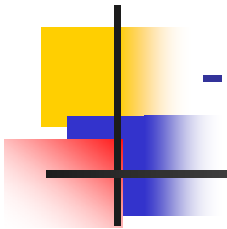
Example b=4 bits, i=2, 2 keys/bucket



Example Continued: How to grow beyond this?

i = ~~2~~ 3



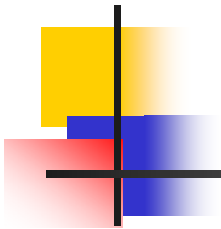


--- When do we expand file?

- Keep track of: $\frac{\text{\# used slots}}{\text{total \# of slots}} = U$

- If U > threshold then increase *m*

  (and maybe *i*)



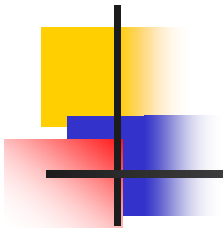
-- Summary: Linear Hashing

(+) Can handle growing files

- with less wasted space
- with no full reorganizations

(+) No indirection like extensible hashing

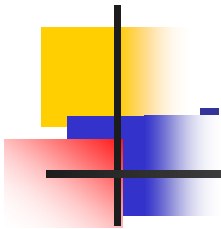
(-) Can still have overflow chains



-- Hashing Summary

Hashing

- How it works
- Dynamic hashing
  - Extensible
  - Linear



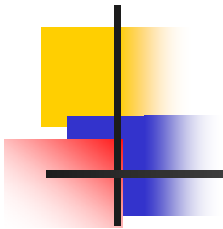
-- Indexing Vs Hashing ...

- Hashing good for probes given key

  e.g., SELECT ...

  FROM R

  WHERE R.A = 5



... -- Indexing vs Hashing

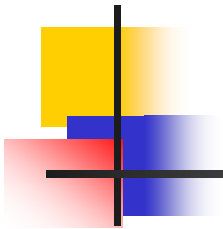
- INDEXING (Including B Trees) good for

  Range Searches:

  e.g., SELECT

  FROM R

  WHERE R.A > 5



- Reading list

- Chapter 13 of GUW
- B-tree (WebCt)

END