Data Storage & Indexes

Instructor: Matei Zaharia

cs245.stanford.edu

Fixed-Length Items

Integer: fixed # of bytes (e.g., 2 bytes)

e.g., 35 is | 00000000 | 00100011 |

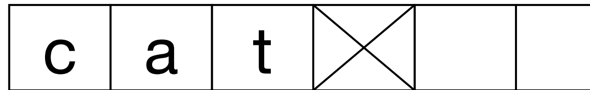
Floating-point: n-bit mantissa, m-bit exponent

Character: encode as integer (e.g. ASCII)

Variable-Length Items

String of characters:

» Null-terminated



» Length + data

» Fixed-length

Bag of bits:

Representing Nothing

NULL concept in SQL (not same as 0 or “”)

Physical representation options:

» Special “sentinel” value in fixed-length field

» Boolean “is null” flag

» Just skip the field in a sparse record format

Pretty common in practice!

Bigger Collections

Data Items

Records

Blocks

Files

Record: Set Data Items (Fields)

E.g. employee record:

» name field
» salary field
» date-of-hire field
» ...

Record Encodings

Fixed vs variable **format**

Fixed vs variable **length**

Fixed Format

A **schema** for all records in table specifies:

- # of fields
- type of each field
- order in record
- meaning of each field

Example: Fixed Format & Length

Employee record

(1) EID, 2 byte integer

(2) Name, 10 chars

(3) Dept, 2 byte code

} Schema

55	s m i t h	02
83	j o n e s	01

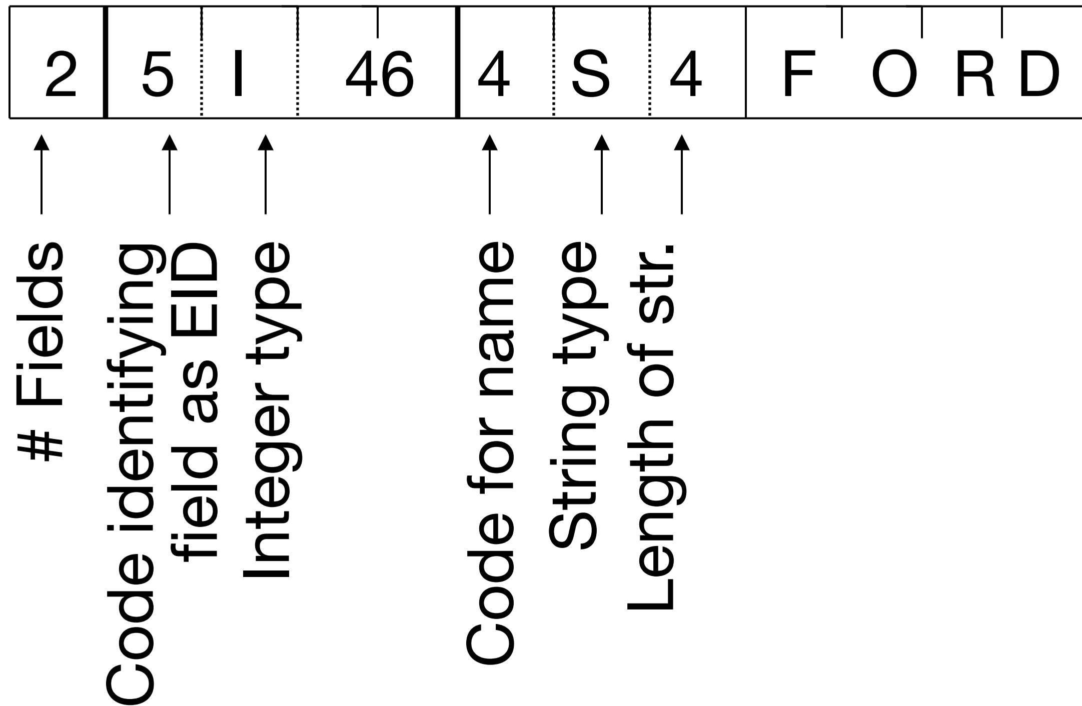
} Records

Variable Format

Record itself contains format

“Self-describing”

Example: Variable Format & Length



Variable Format Useful For

"Sparse" records

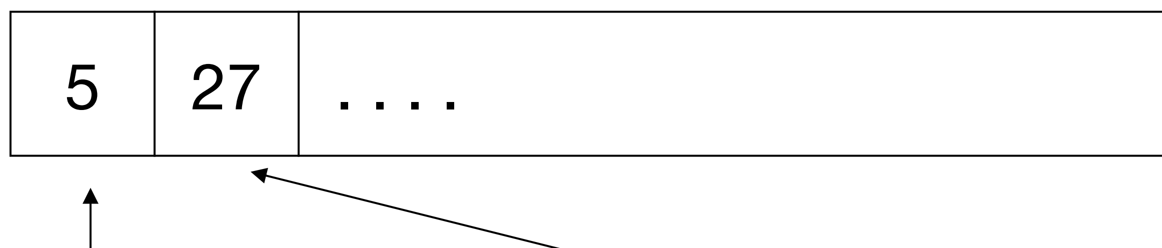
Repeating fields

Evolving formats

But may waste space...

Many Variants Between Fixed and Variable Format

Example: Include a **record type** in record



Type is a pointer to one of several schemas

Outline

Overview

Record encoding

Collection storage

Indexes

Collection Storage Questions

How do we place data items and records for efficient access?

» **Locality** and **searchability**

How do we physically encode records in blocks and files?

Placing Data for Efficient Access

Locality: which items are accessed together

» When you read one field of a record, you're likely to read other fields of the same record

» When you read one field of record 1, you're likely to read the same field of record 2

Searchability: quickly find relevant records

» E.g. sorting the file lets you do binary search

Locality Example: Row Stores vs Column Stores

Row Store

name	age	state
Alex	20	CA
Bob	30	CA
Carol	42	NY
David	21	MA
Eve	26	CA
Frances	56	NY
Gia	19	MA
Harold	28	AK
Ivan	41	CA

Fields stored contiguously in one file

Column Store

name
Alex
Bob
Carol
David
Eve
Frances
Gia
Harold
Ivan

age
20
30
42
21
26
56
19
28
41

state
CA
CA
NY
MA
CA
NY
MA
AK
CA

Each column in a different file

Locality Example: Row Stores vs Column Stores

Row Store

name	age	state
Alex	20	CA
Bob	30	CA
Carol	42	NY
David	21	MA
Eve	26	CA
Frances	56	NY
Gia	19	MA
Harold	28	AK
Ivan	41	CA

Fields stored contiguously in one file

Column Store

name
Alex
Bob
Carol
David
Eve
Frances
Gia
Harold
Ivan

age
20
30
42
21
26
56
19
28
41

state
CA
CA
NY
MA
CA
NY
MA
AK
CA

Each column in a different file

Accessing all fields of one record: 1 random I/O for row, 3 for column

Locality Example: Row Stores vs Column Stores

Row Store

name	age	state
Alex	20	CA
Bob	30	CA
Carol	42	NY
David	21	MA
Eve	26	CA
Frances	56	NY
Gia	19	MA
Harold	28	AK
Ivan	41	CA

Fields stored contiguously in one file

Column Store

name
Alex
Bob
Carol
David
Eve
Frances
Gia
Harold
Ivan

age
20
30
42
21
26
56
19
28
41

state
CA
CA
NY
MA
CA
NY
MA
AK
CA

Each column in a different file

Accessing one field of all records: 3x less I/O for column store

Z-Ordering

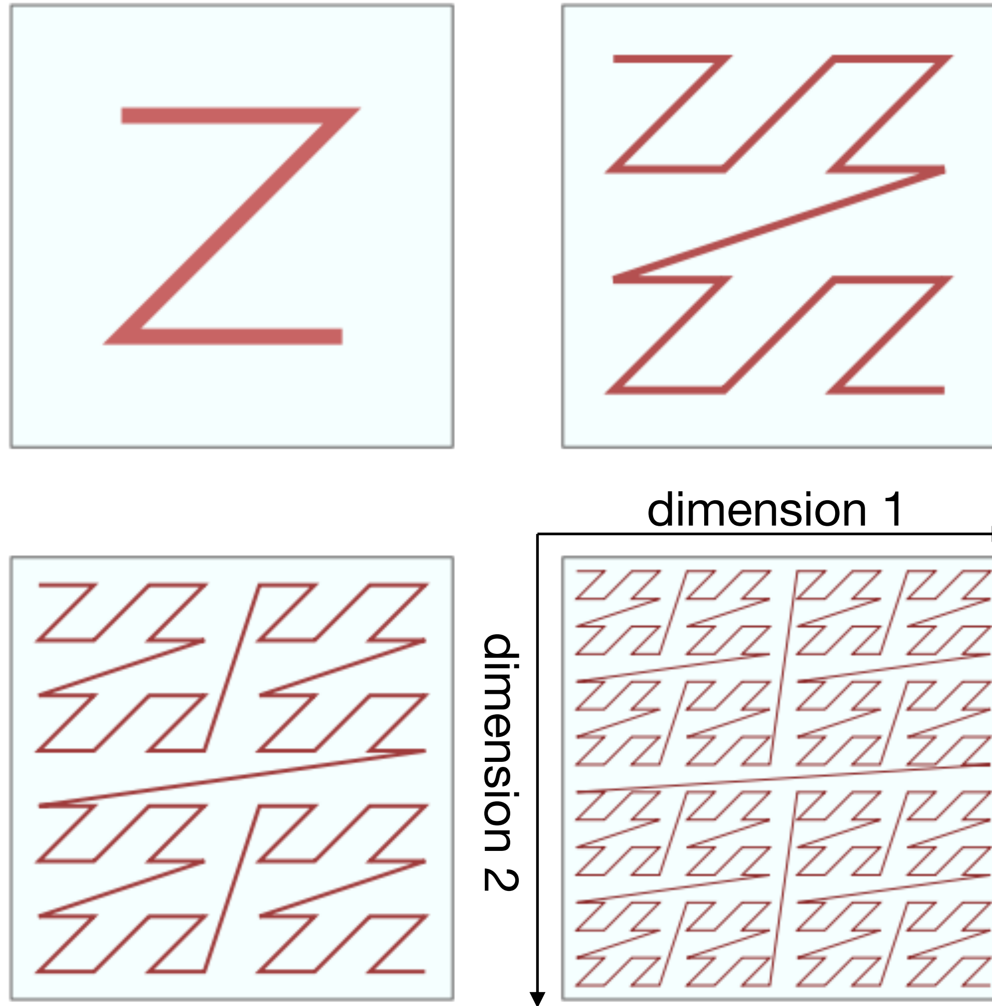
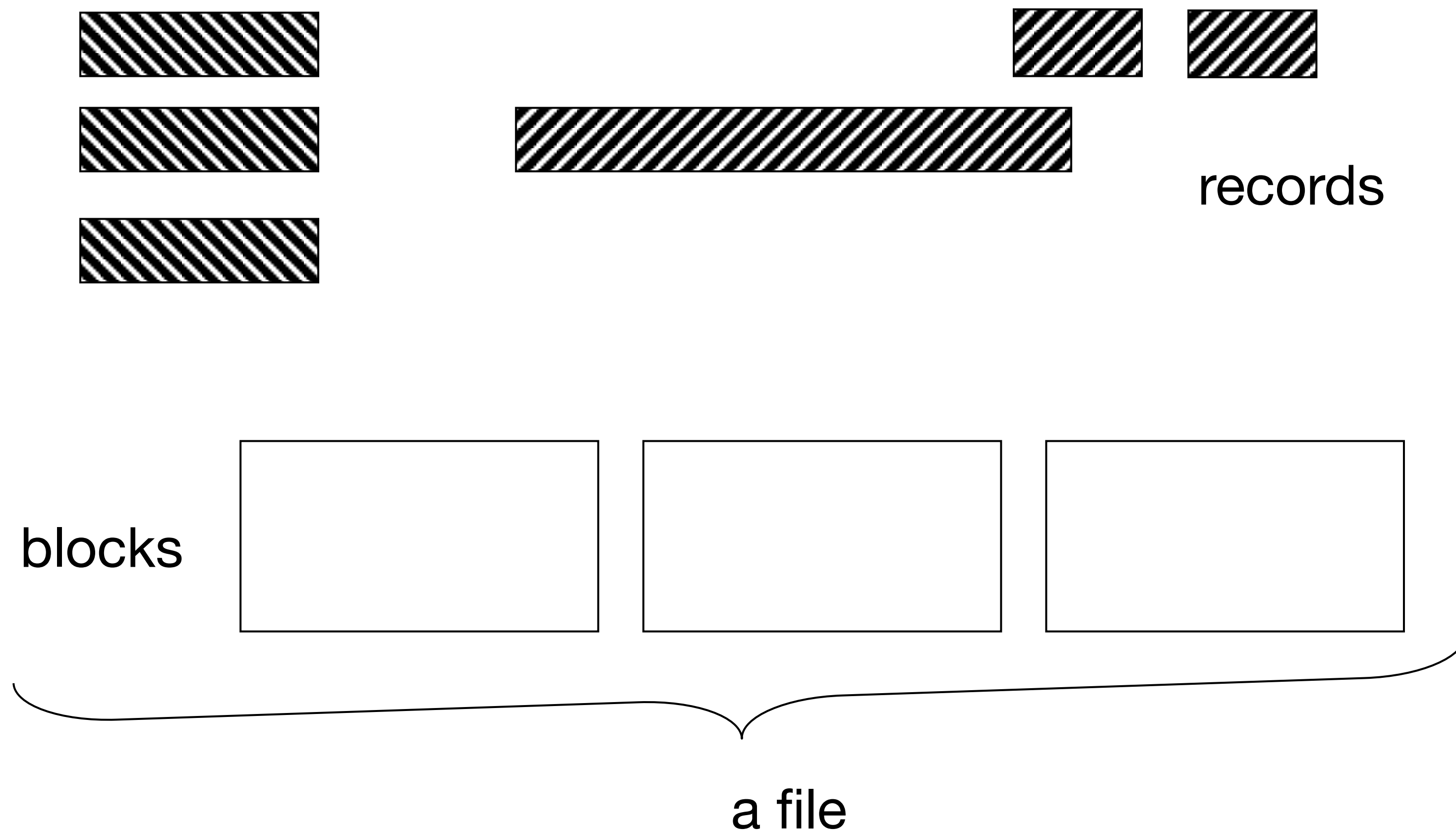
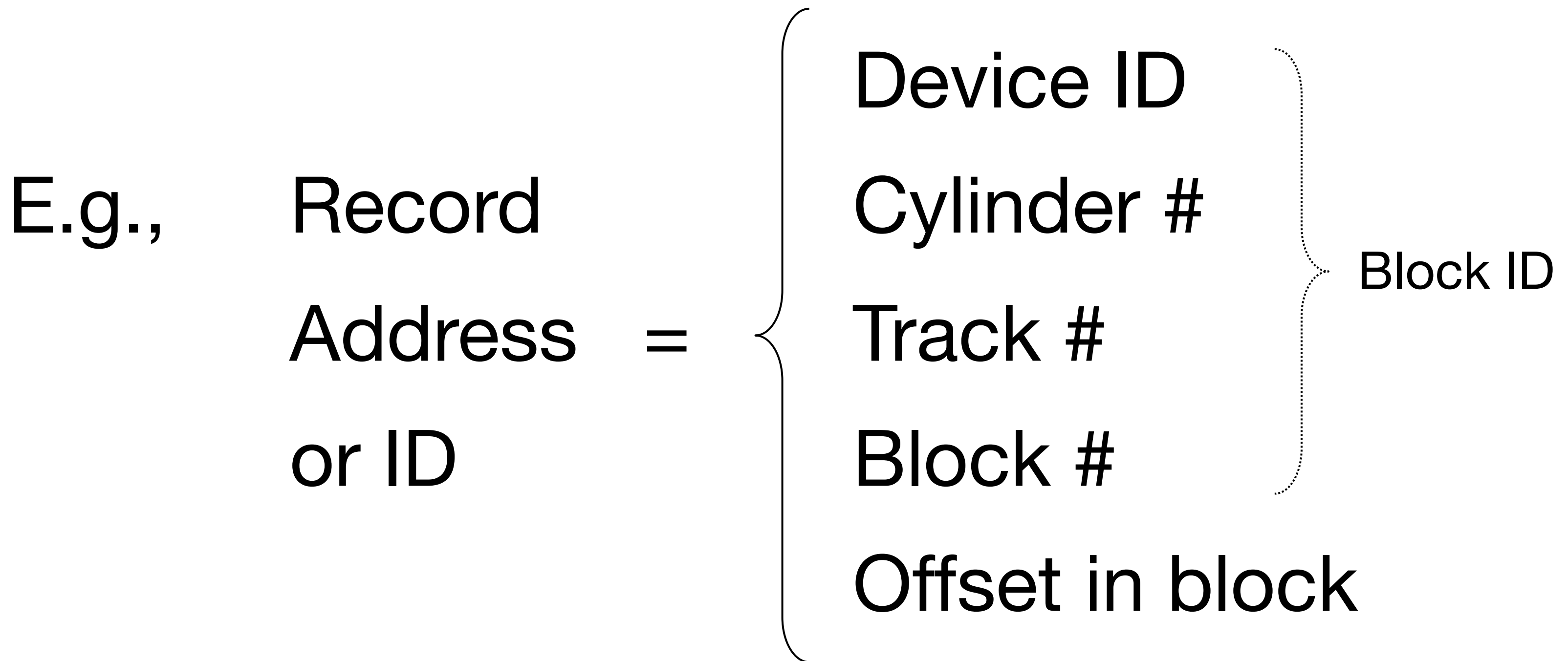


Image source: Wikipedia

How Do We Encode Records into Blocks & Files?

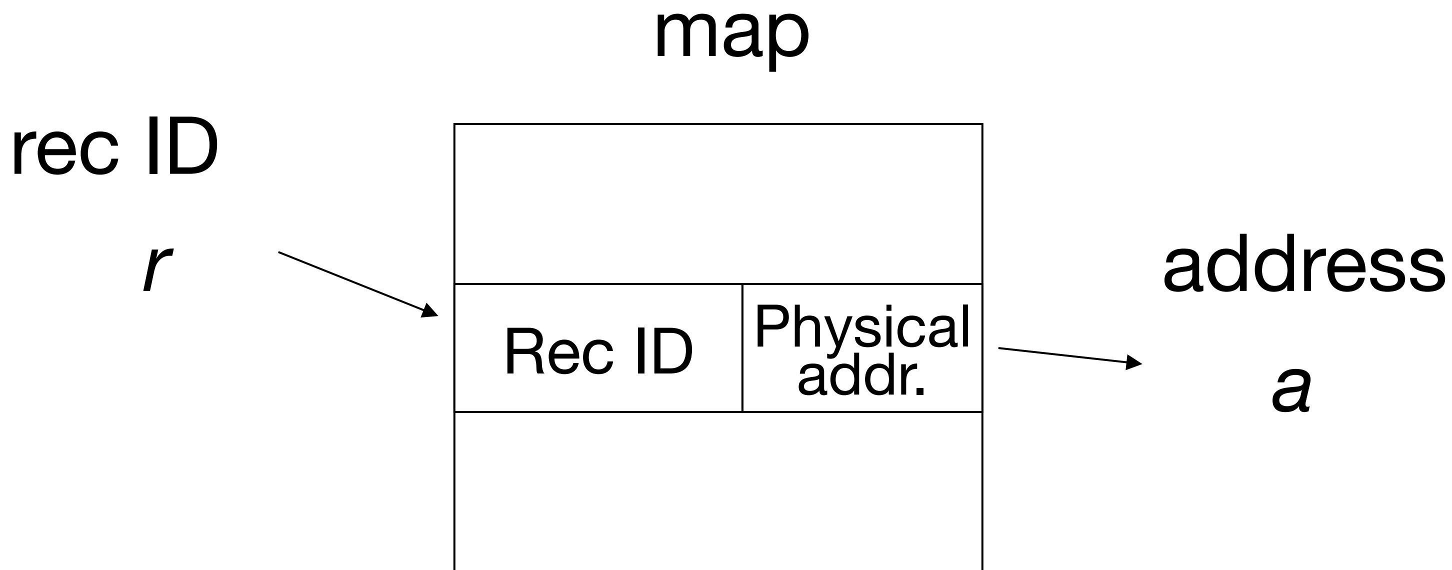


Purely Physical



Fully Indirect

E.g., Record ID is arbitrary bit string



Tradeoff

Flexibility Cost

to move records — of indirection

(for deletions, insertions)

Inserting Records

Easy case: records not ordered

» Insert record at end of file or in a free space

» Harder if records are variable-length

Hard case: records are ordered

» If free space close by, not too bad...

» Otherwise, use an **overflow** area and reorganize the file periodically

Deleting Records

Immediately reclaim space

OR

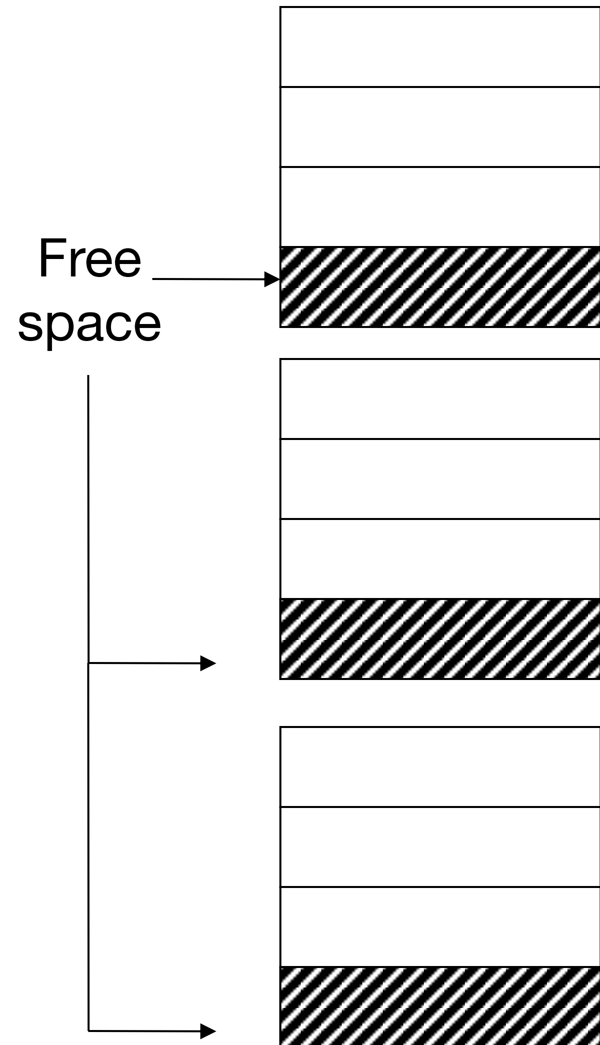
Mark deleted

- And keep track of freed spaces for later use

Interesting Problems

How much free space to leave in each block, track, cylinder, etc?

How often to reorganize file + merge overflow?



Compressing Collections

Usually for a block at a time

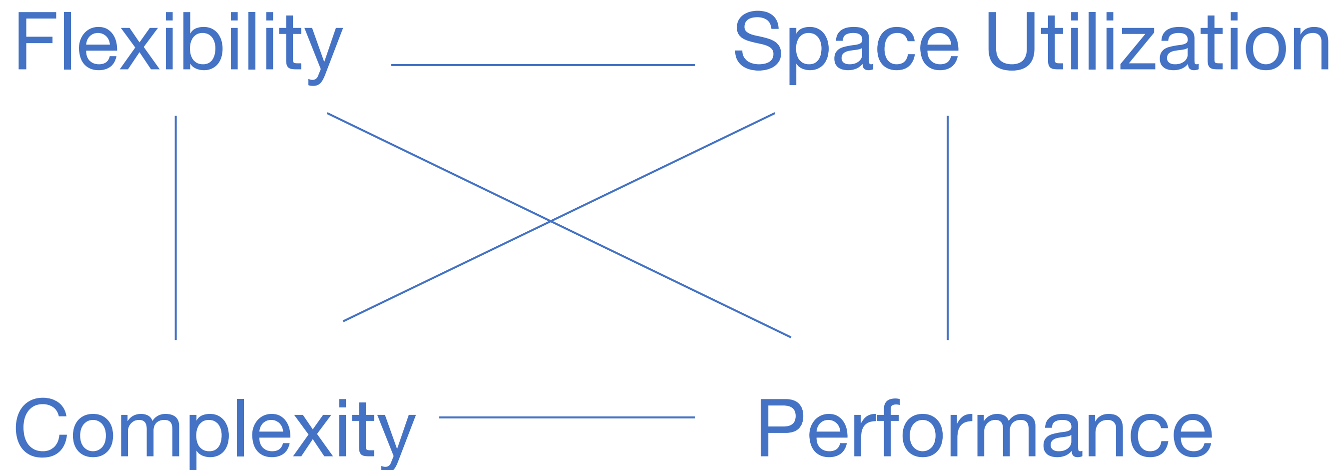
- » Benefits from placing similar items together

Can be integrated with execution (C-Store)

Summary

There are many ways to organize data on disk

Key tradeoffs:



To Evaluate a Strategy, Compute:

Space used for expected data

Expected time to

- fetch record given key
- read whole file
- insert record
- delete record
- update record
- reorganize file
- ...

Data Storage Formats

Instructor: Matei Zaharia

cs245.stanford.edu

Outline

Storage devices wrap-up

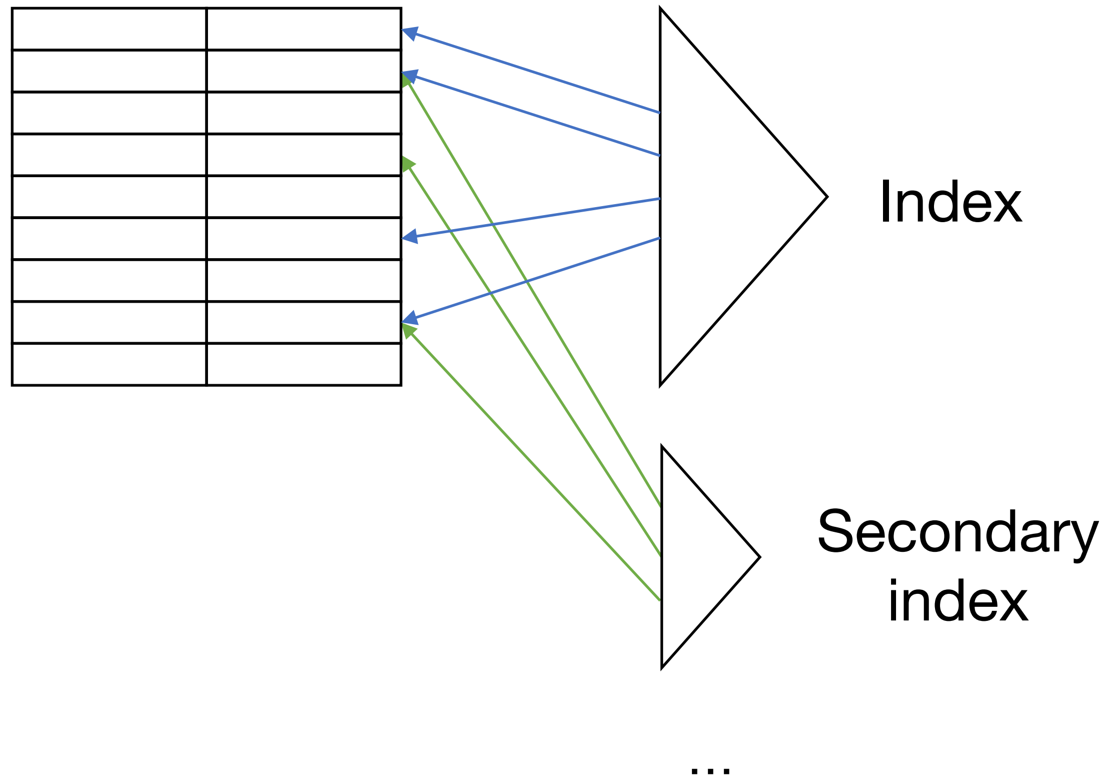
Record encoding

Collection storage

C-Store paper

Indexes

General Setup



What Are the Data Items We Want to Store?

a salary

a name

a date

a picture

What Are the Data Items We Want to Store?

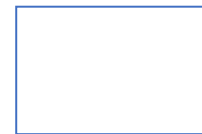
a salary

a name

a date

a picture

What we have available: bytes



Outline

C-Store (codesigning compute & storage)

Indexes

Key Operations on an Index

Find all records with a given **value** for a key

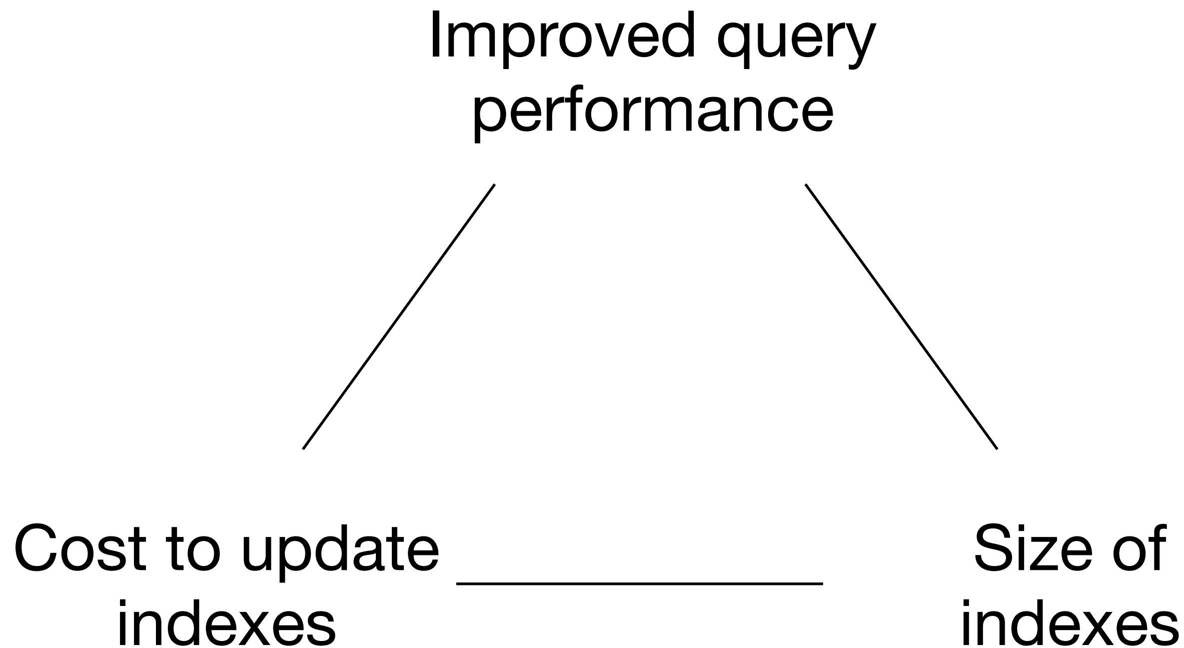
» Key can be one field or a tuple of fields (e.g. `country=“US” AND state=“CA”`)

» In some cases, only one matching record

Find all records with key in a given **range**

Find **nearest neighbor** to a data point?

Tradeoffs in Indexing



Some Types of Indexes

Conventional indexes

B-trees

Hash indexes

Multi-key indexing

Many standard data structures, but adapted to work well on disk

Sequential File

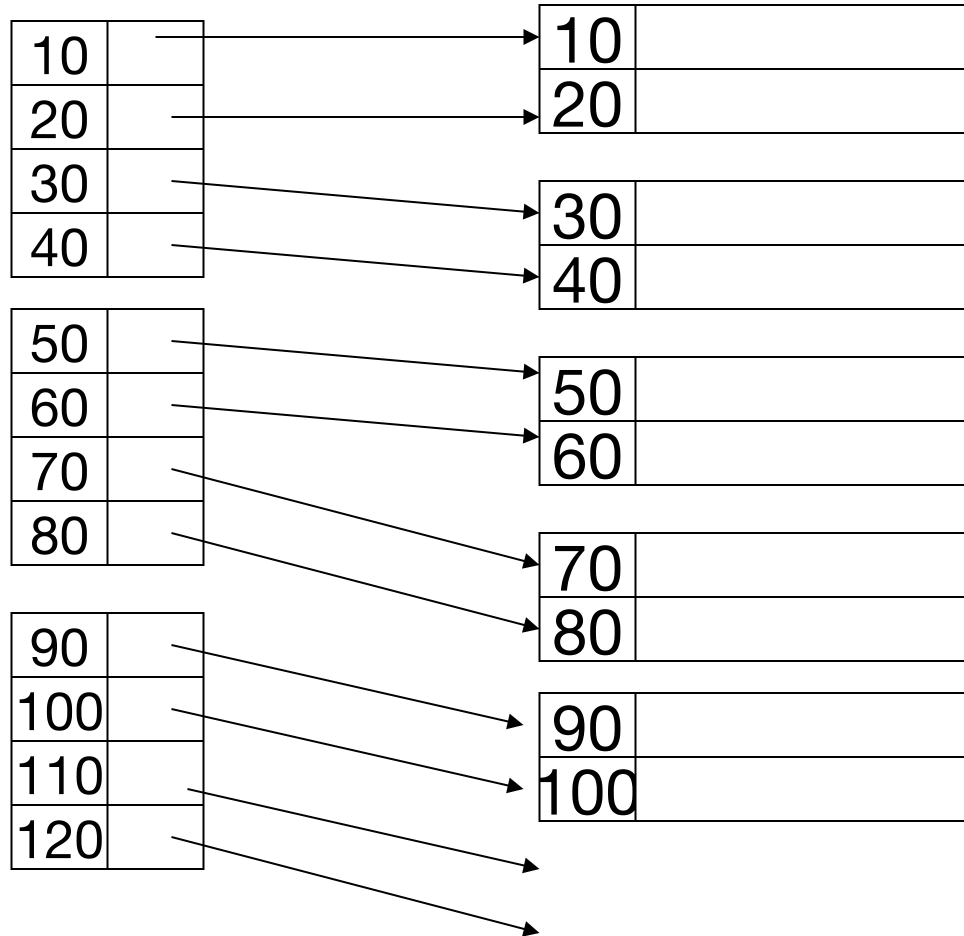
10	
20	

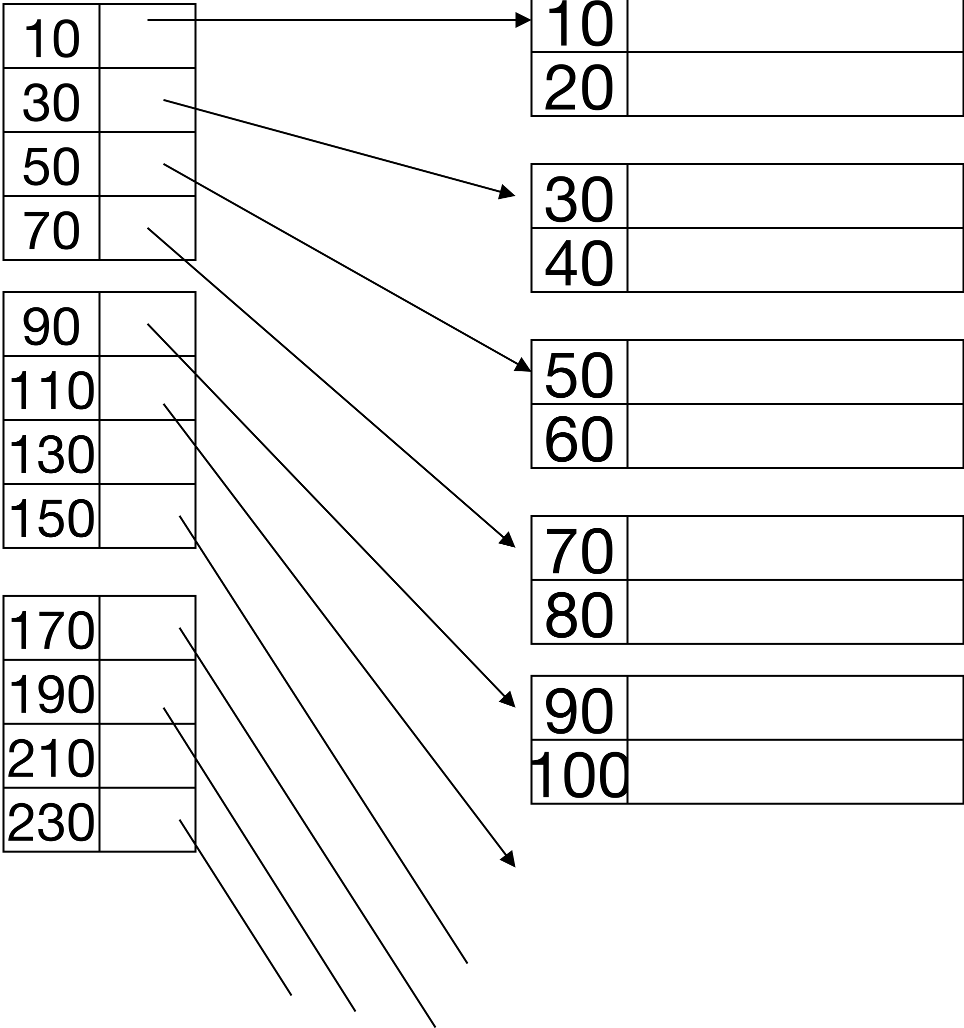
30	
40	

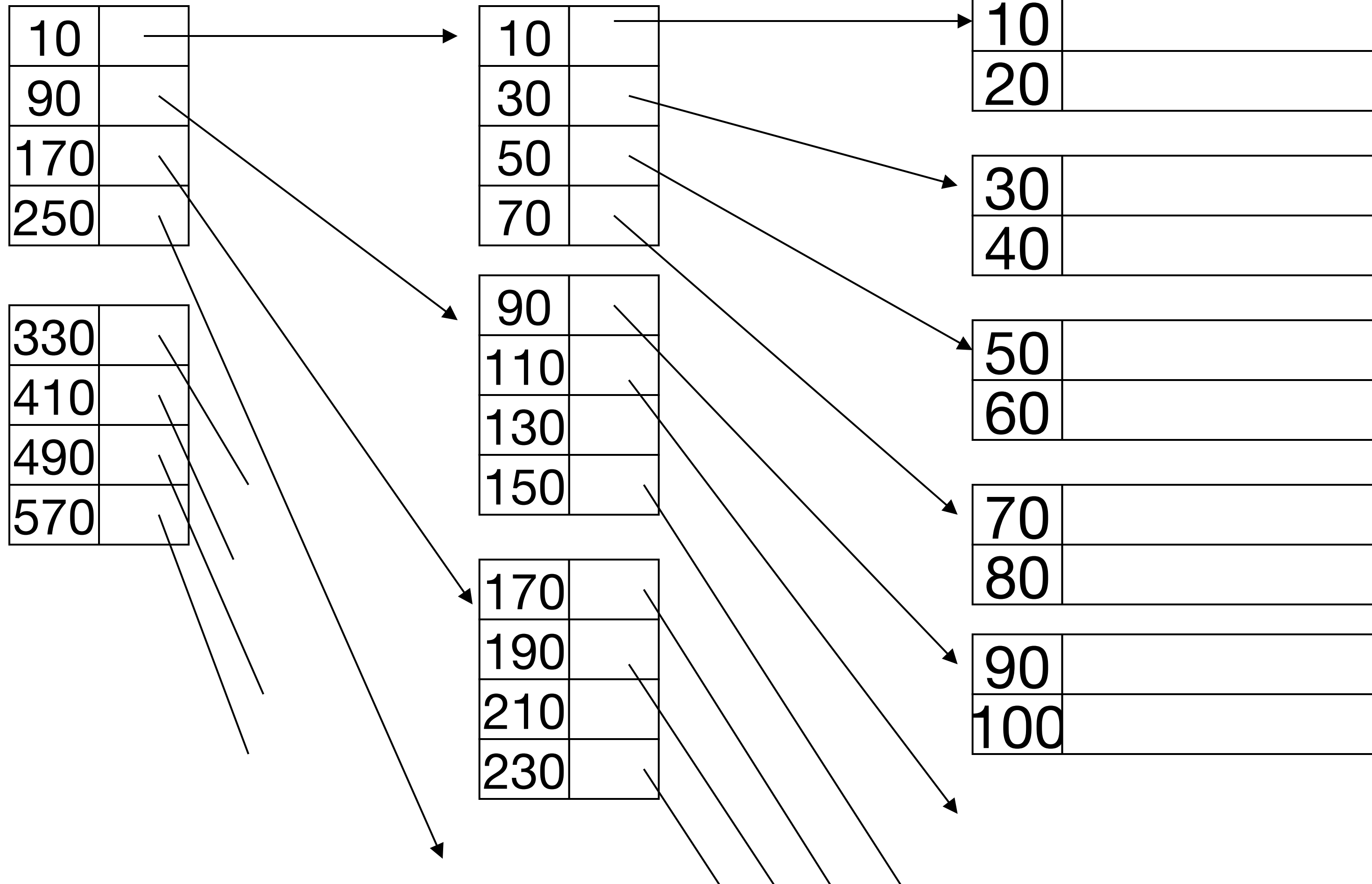
50	
60	

70	
80	

90	
100	


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


Sparse Index
Sequential File
10
30
50
70
90
110
130
150
170
190
210
230
10
20
30
40
50
60
70
80
90
100



File and 2nd level index blocks need not be contiguous on disk

Sparse vs Dense Tradeoff

Sparse: Less space usage, can keep more of index in memory

Dense: Can tell whether a key is present without accessing file

(Later: sparse better for insertions, dense needed for secondary indexes)

Terms

Search key of an index

Primary index (on primary key of ordered files)

Secondary index

Dense index (contains all search key values)

Sparse index

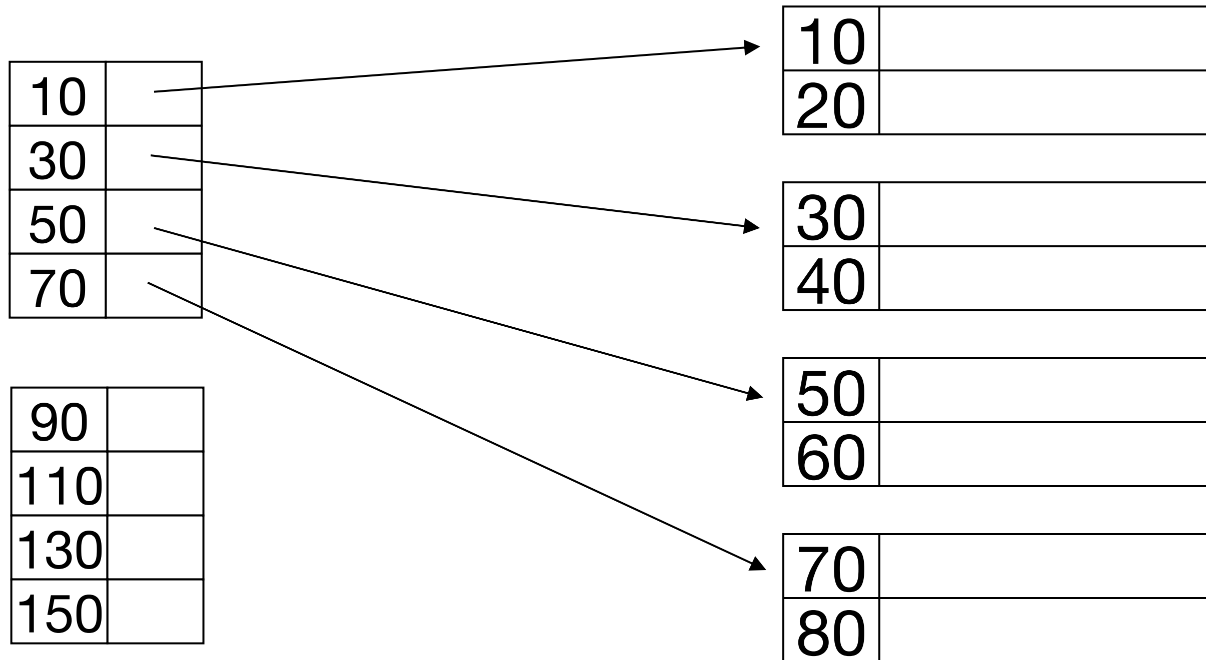
Multi-level index

Handling Duplicate Keys

For a primary index, can point to 1st instance of each item (assuming blocks are linked)

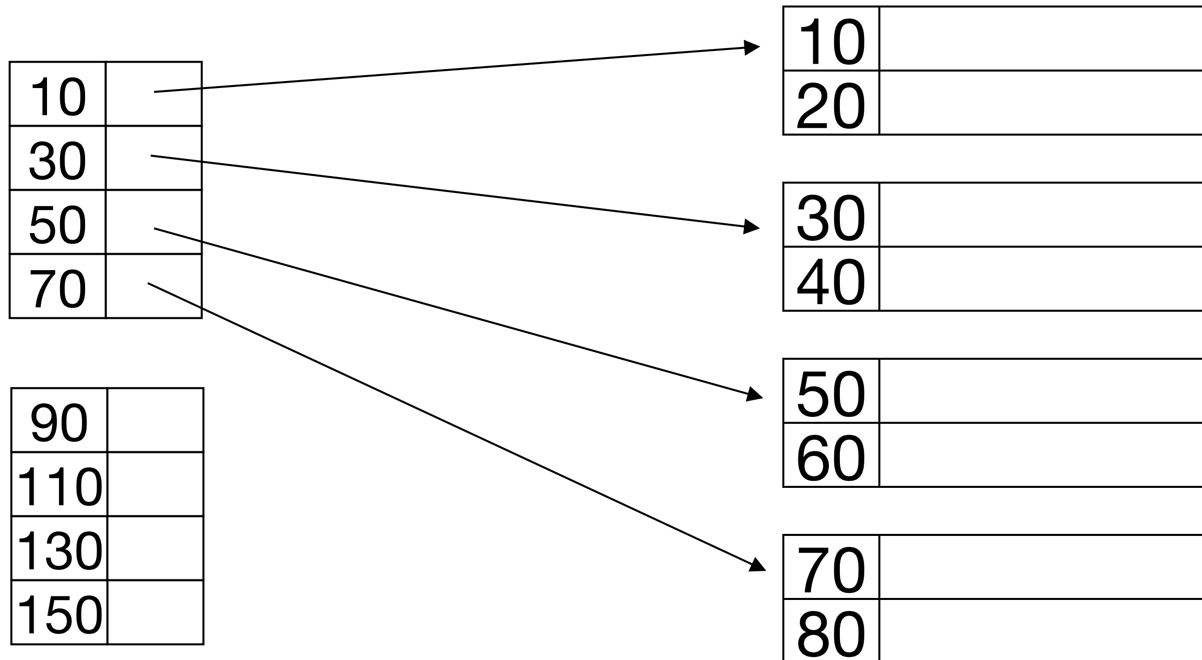
For a secondary index, need to point to a list of records since they can be anywhere

Deletion: Sparse Index



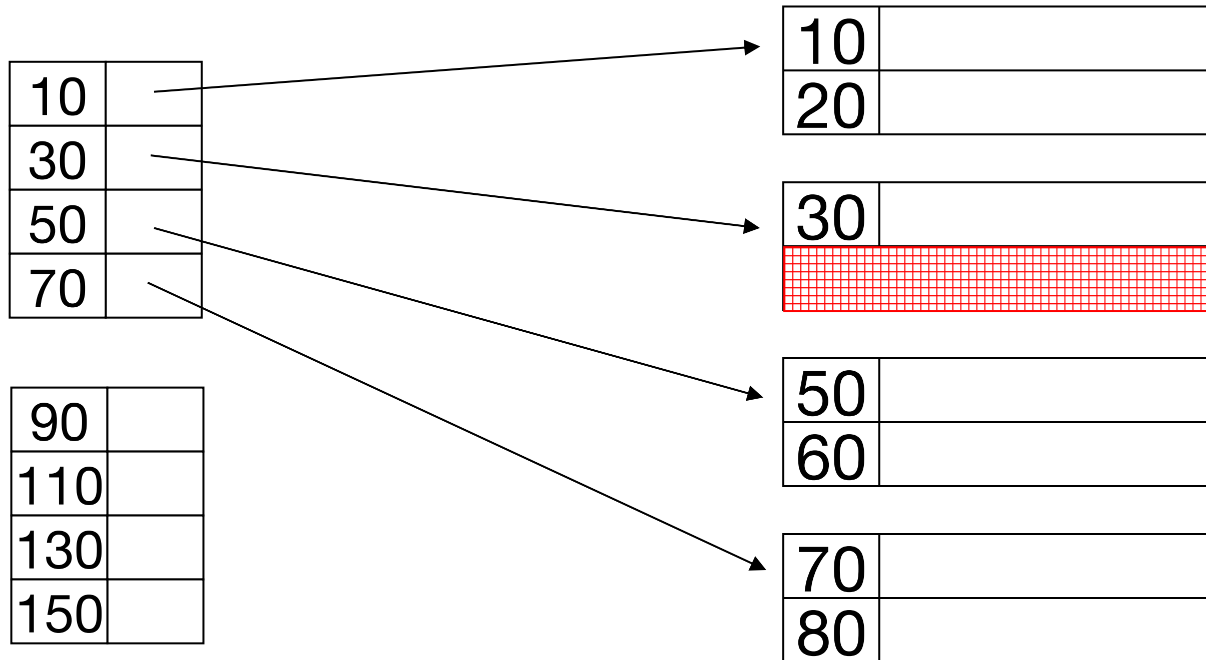
Deletion: Sparse Index

– delete record 40



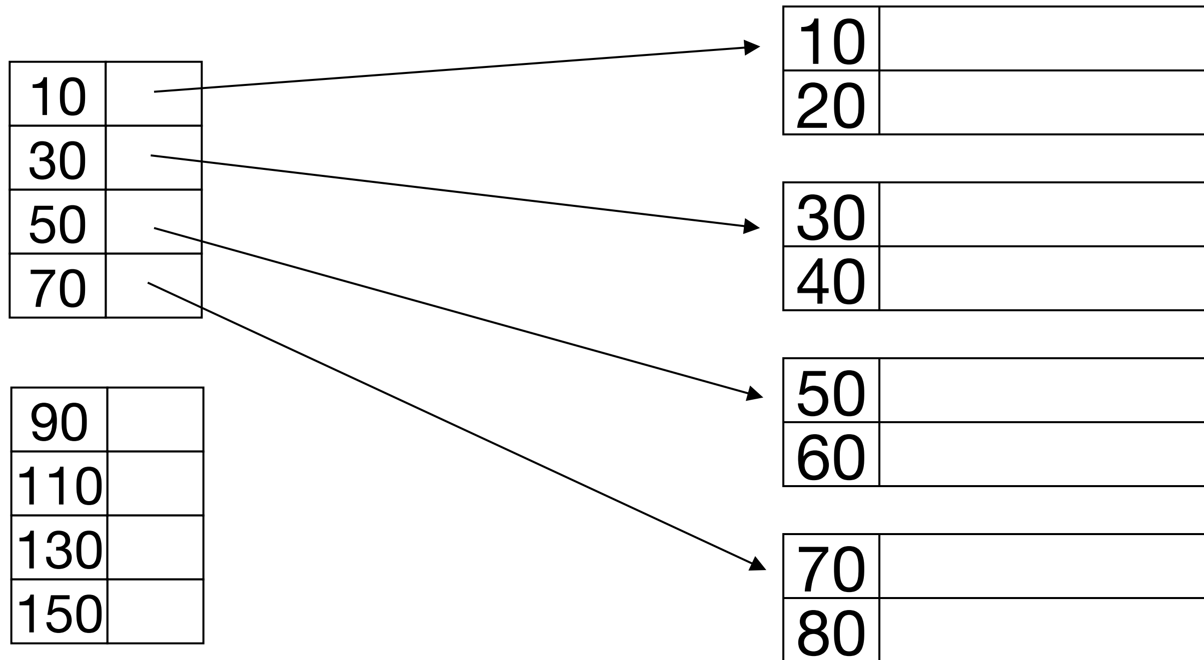
Deletion: Sparse Index

– delete record 40



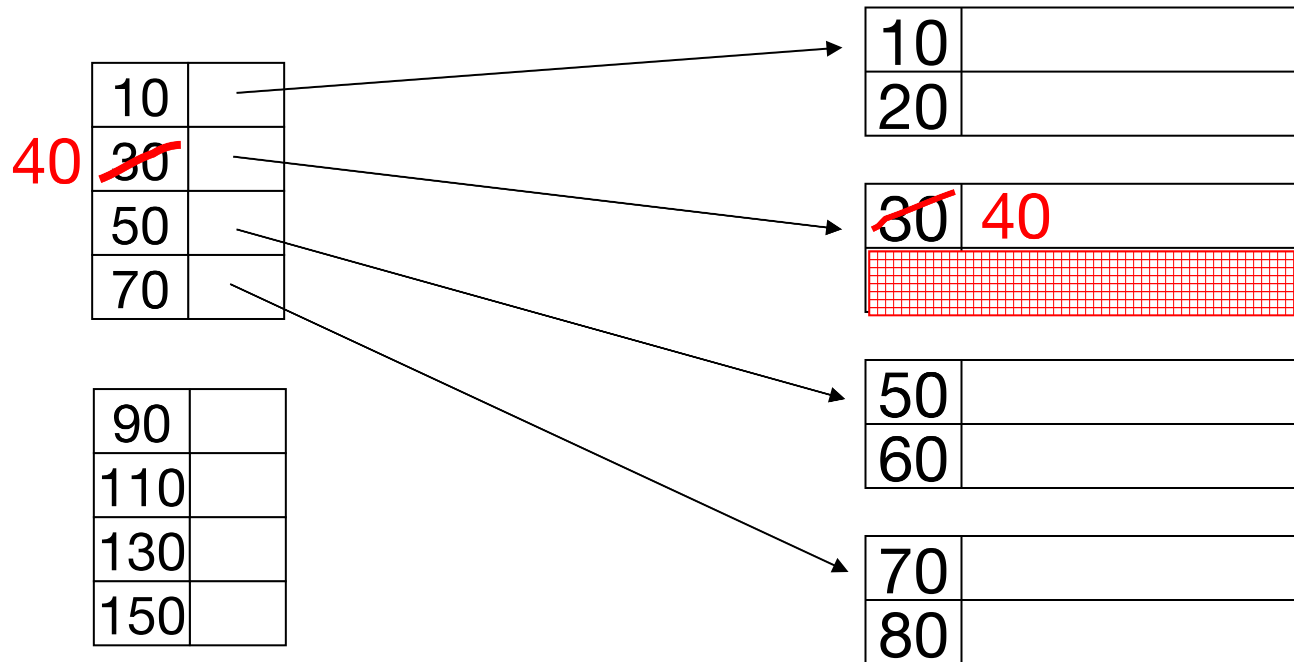
Deletion: Sparse Index

– delete record 30



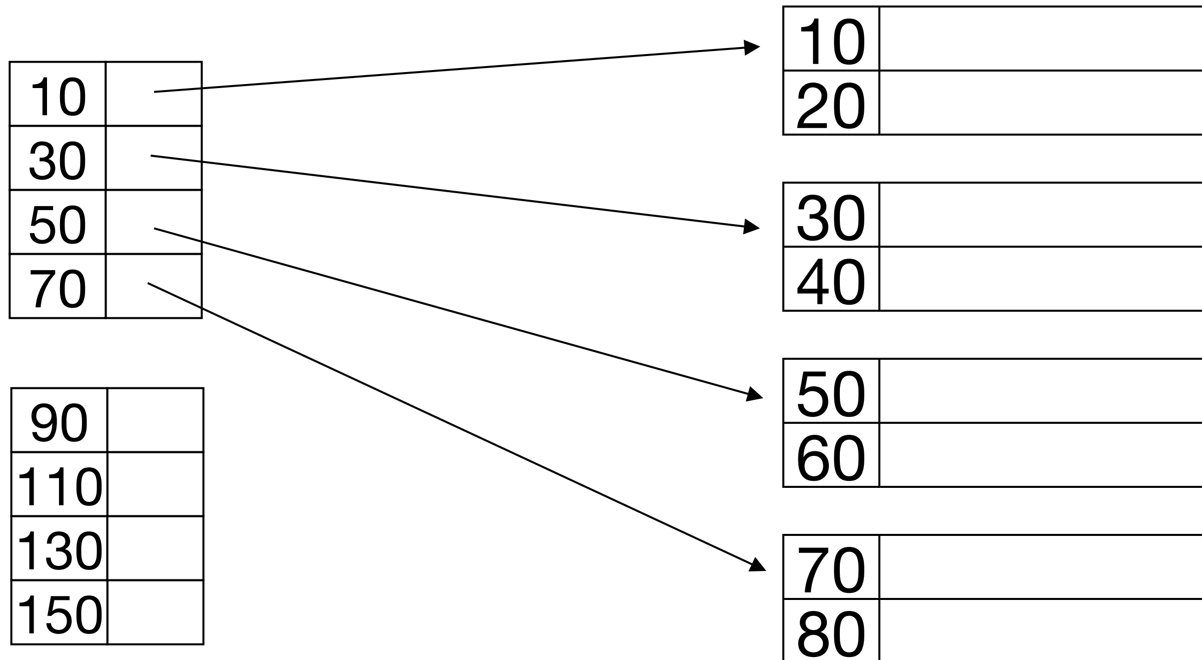
Deletion: Sparse Index

– delete record 30



Deletion: Sparse Index

– delete records 30 & 40



Deletion: Sparse Index

– delete records 30 & 40

10	
30	
50	
70	

90	
110	
130	
150	

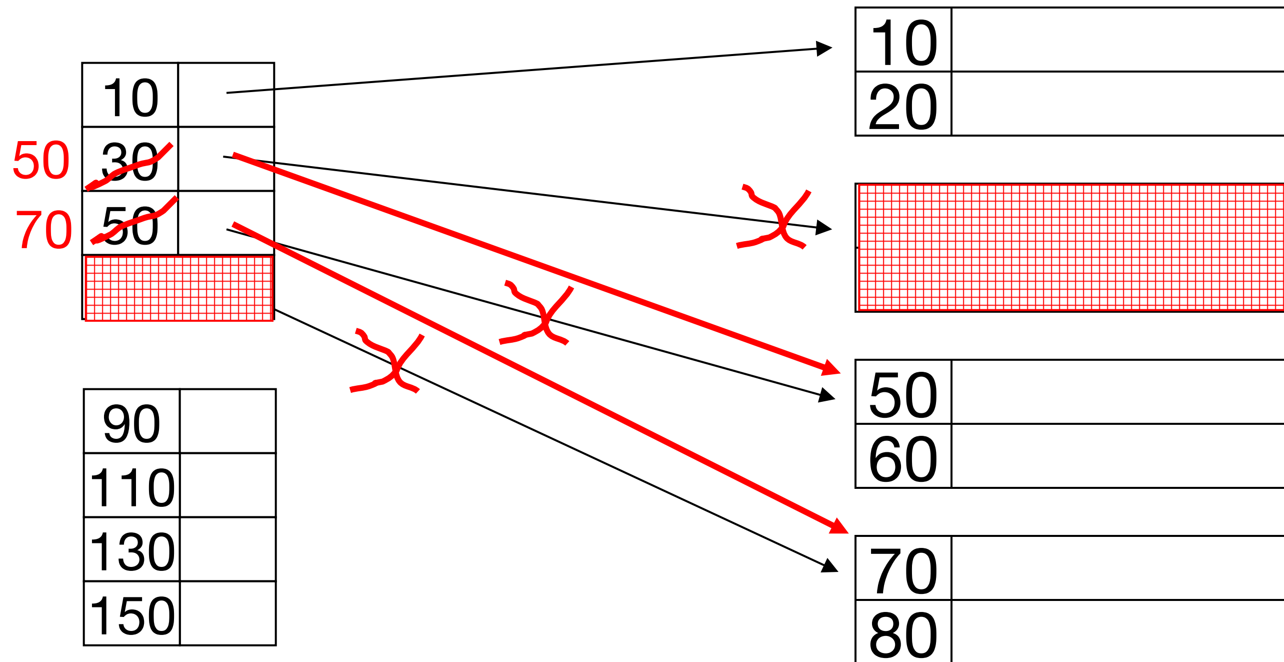
10	
20	

50	
60	

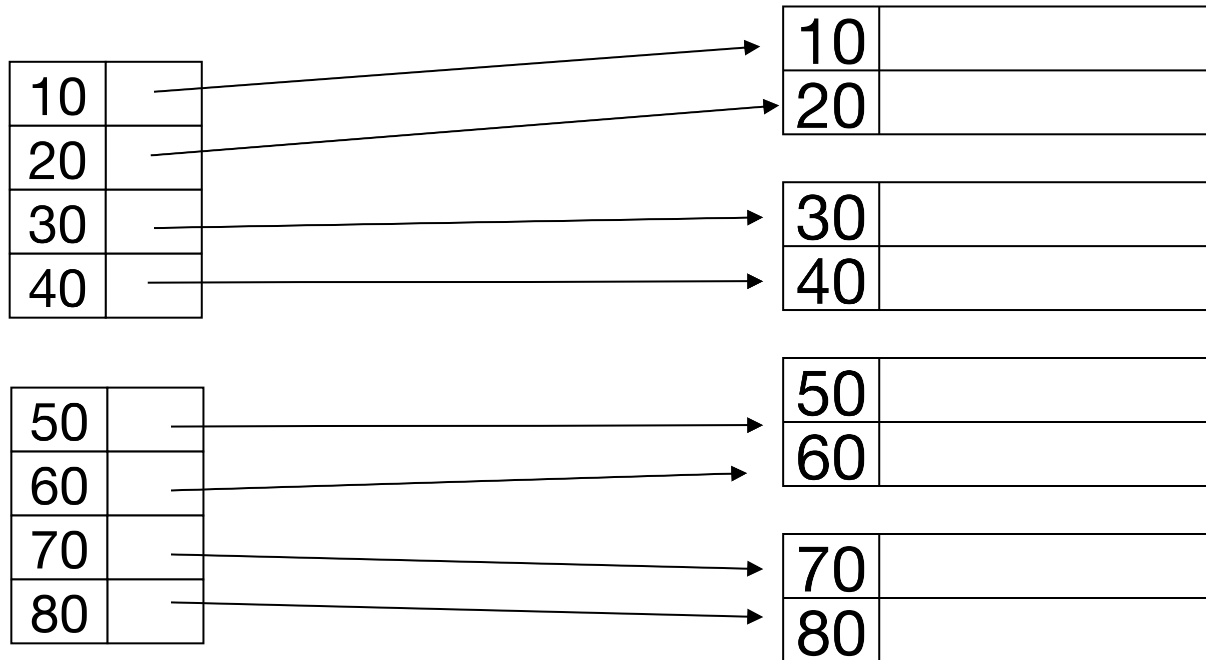
70	
80	

Deletion: Sparse Index

– delete records 30 & 40

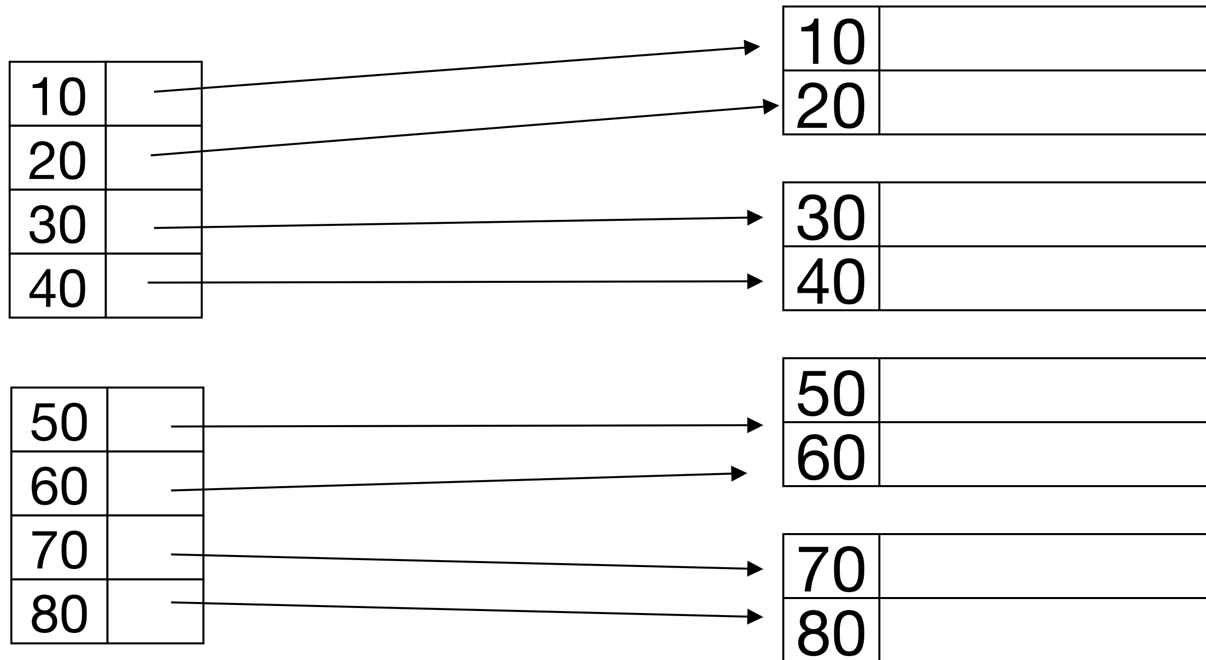


Deletion: Dense Index



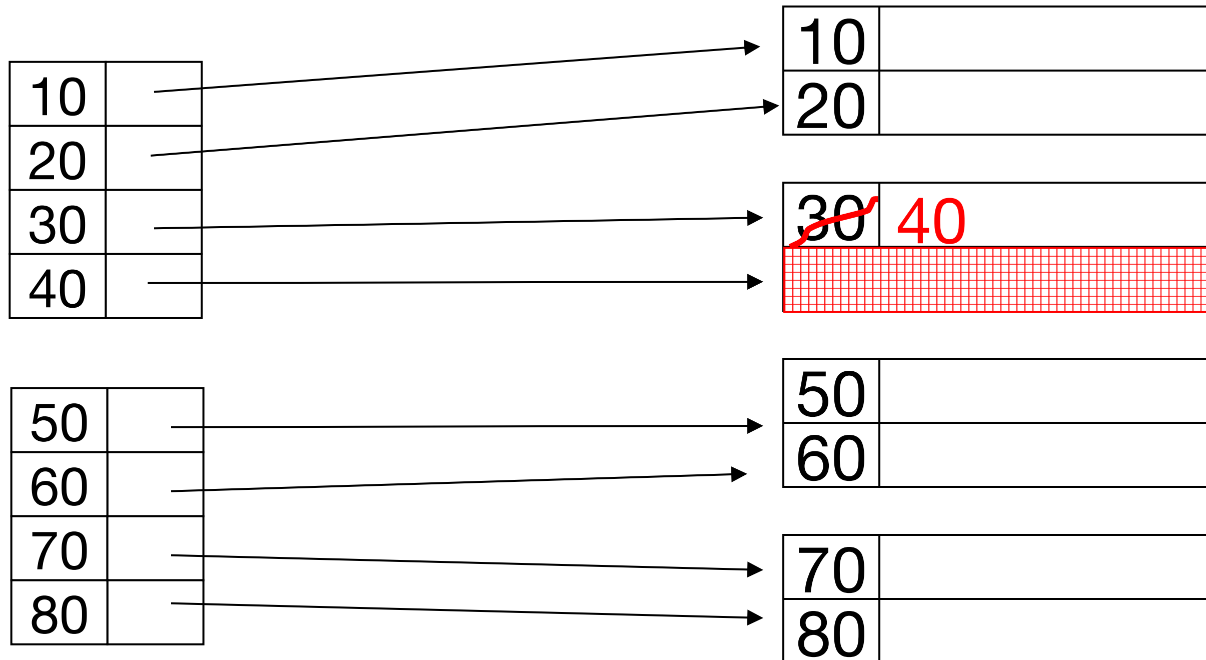
Deletion: Dense Index

– delete record 30



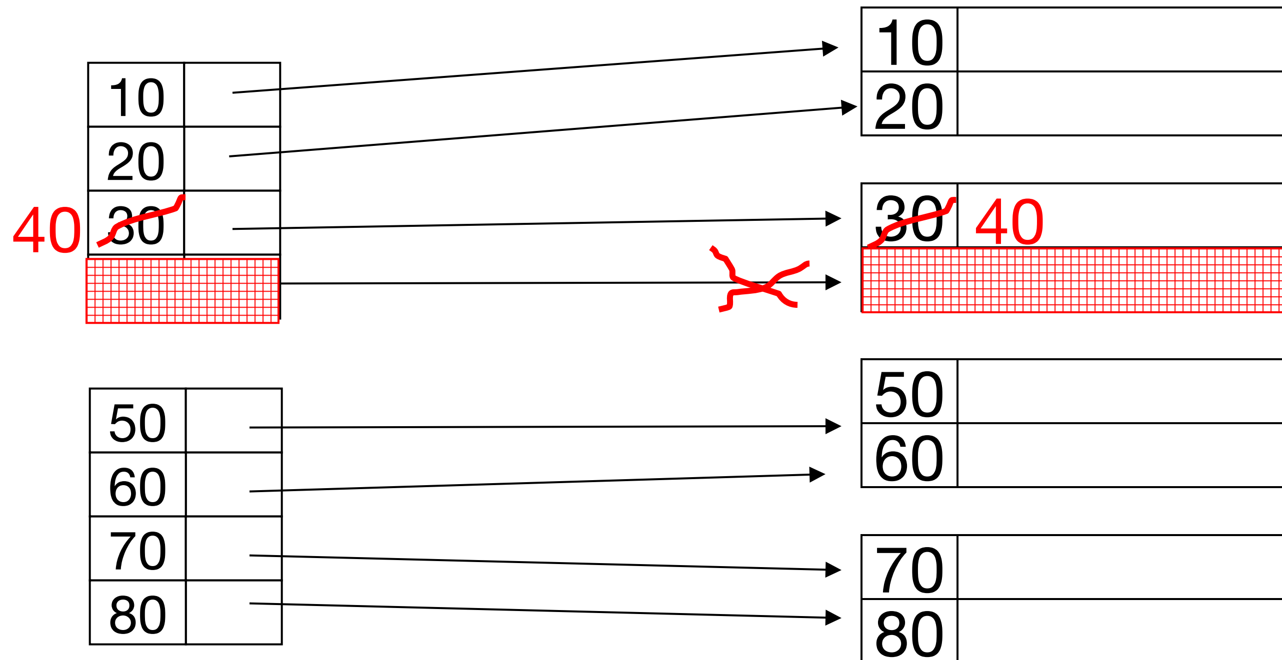
Deletion: Dense Index

– delete record 30



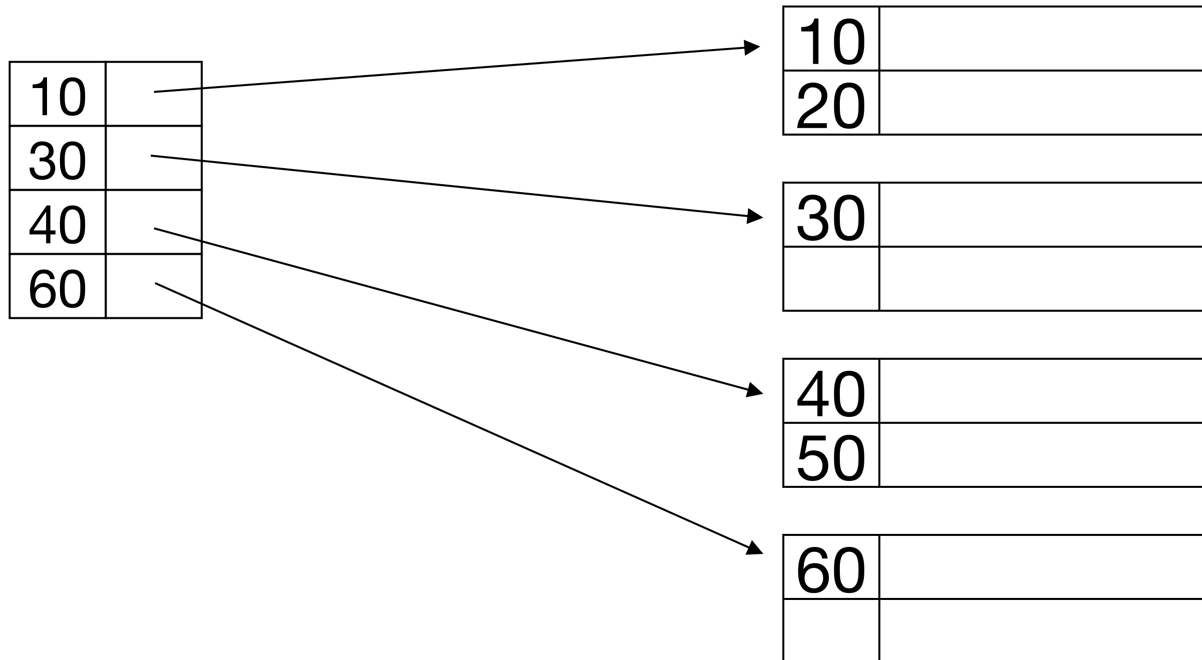
Deletion: Dense Index

– delete record 30



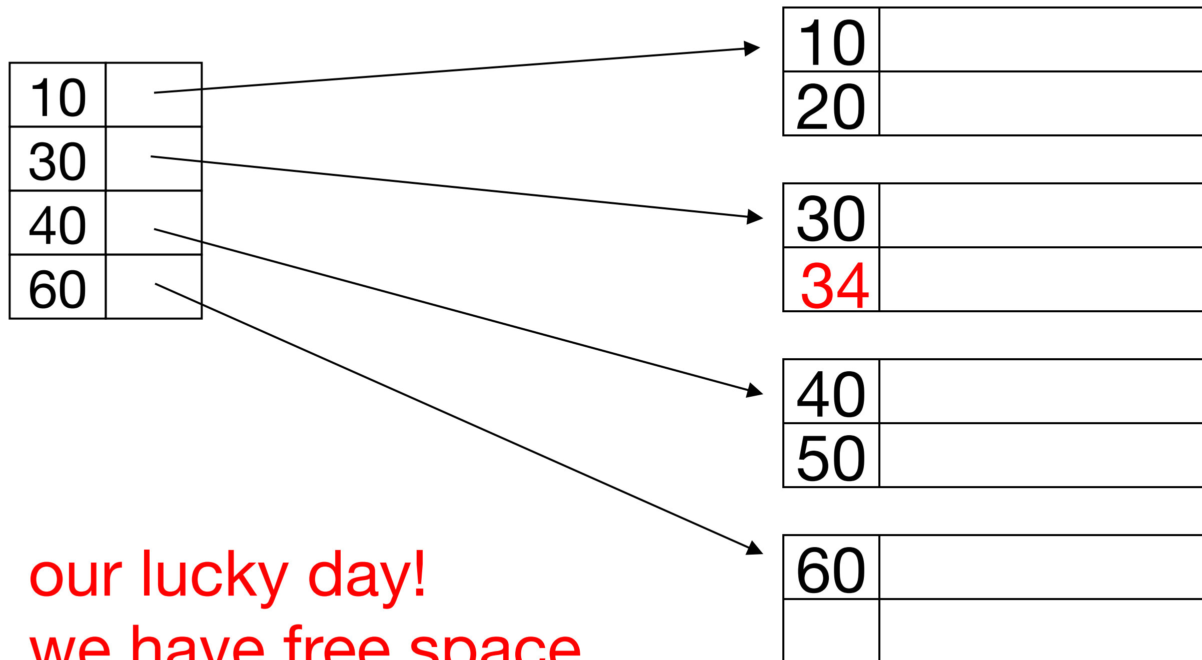
Insertion: Sparse Index

– insert record 34



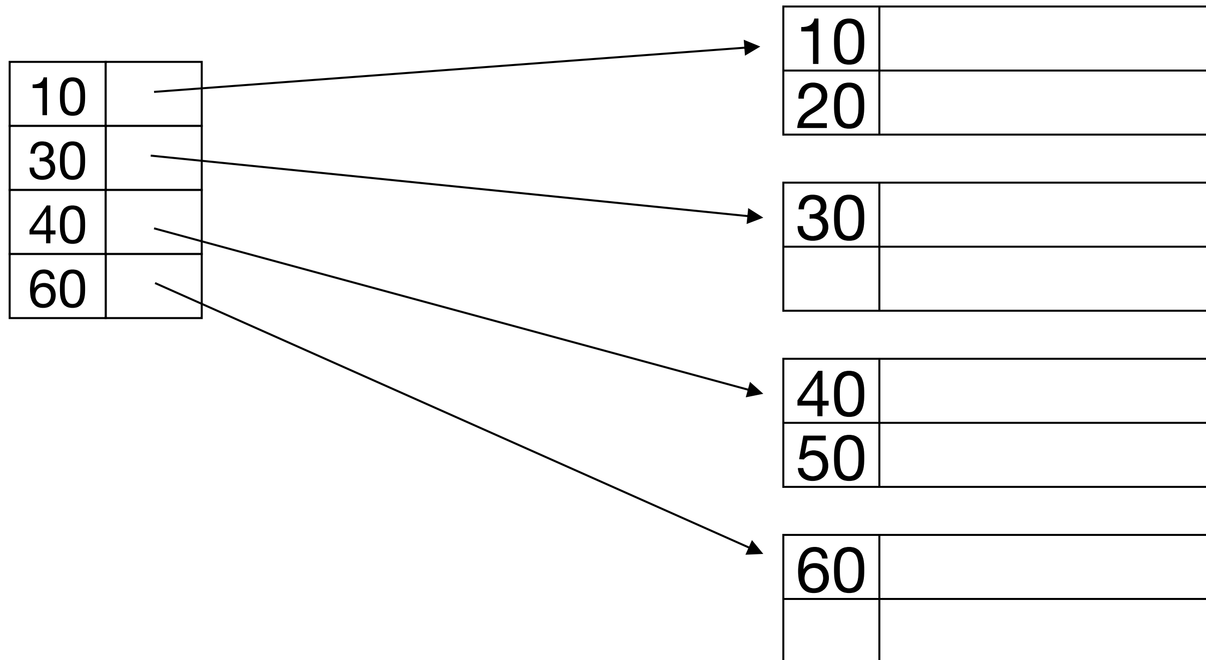
Insertion: Sparse Index

– insert record 34



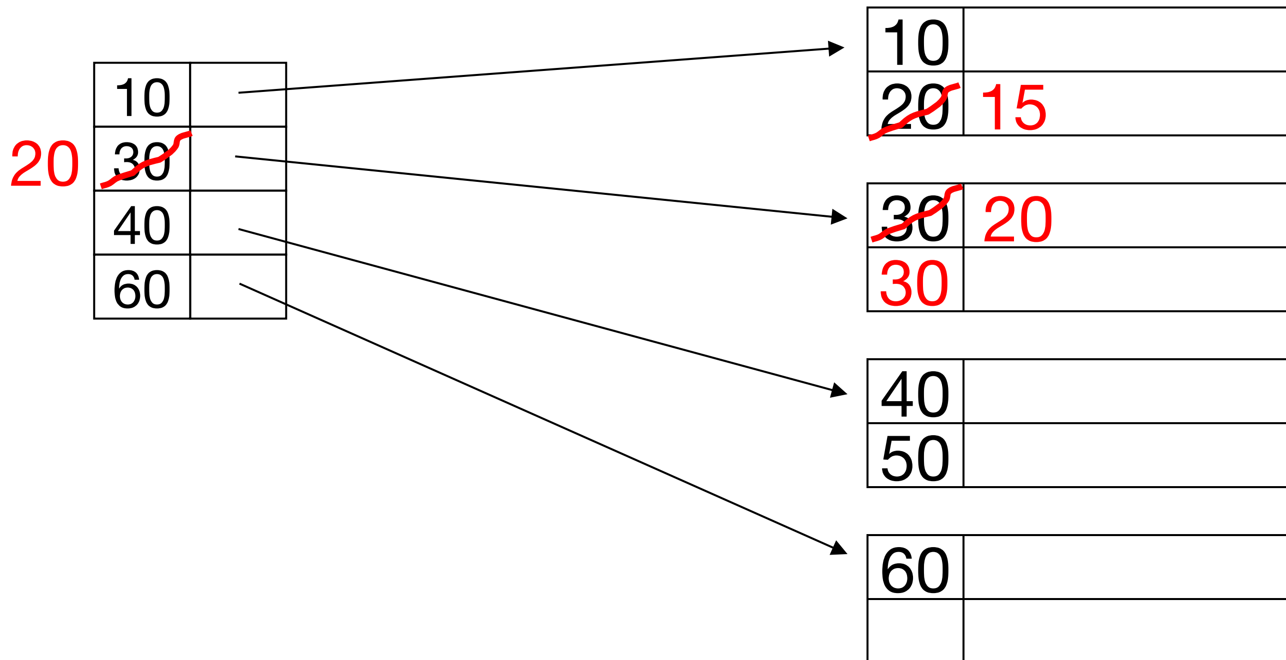
Insertion: Sparse Index

– insert record 15



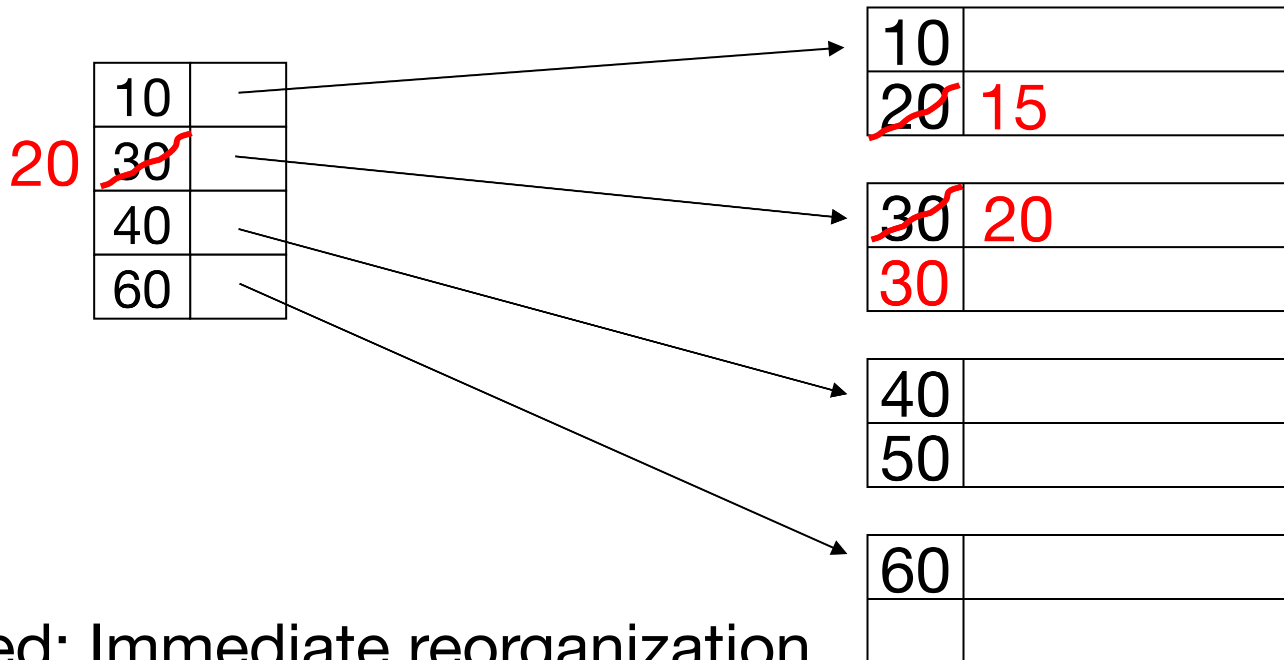
Insertion: Sparse Index

– insert record 15



Insertion: Sparse Index

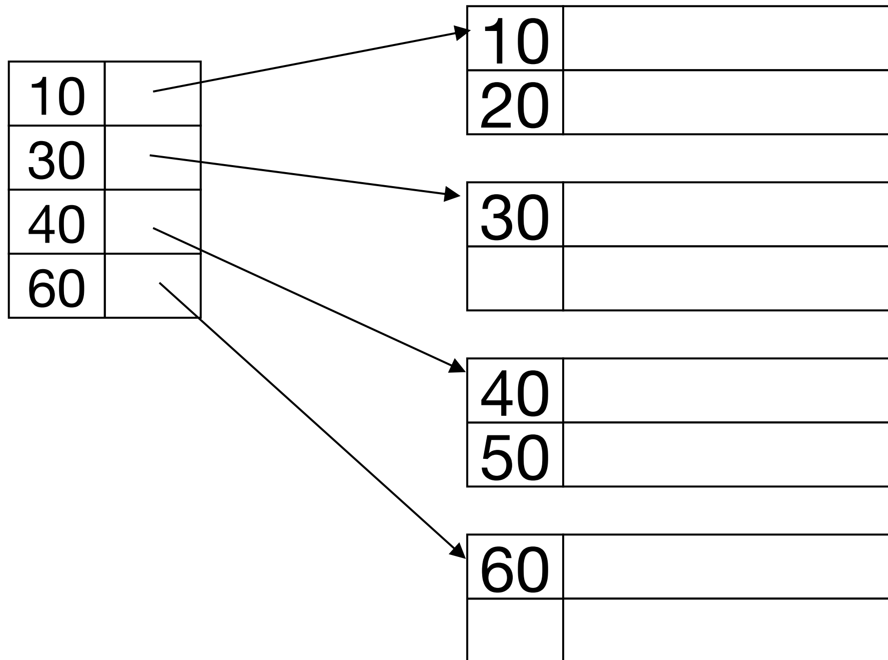
– insert record 15



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

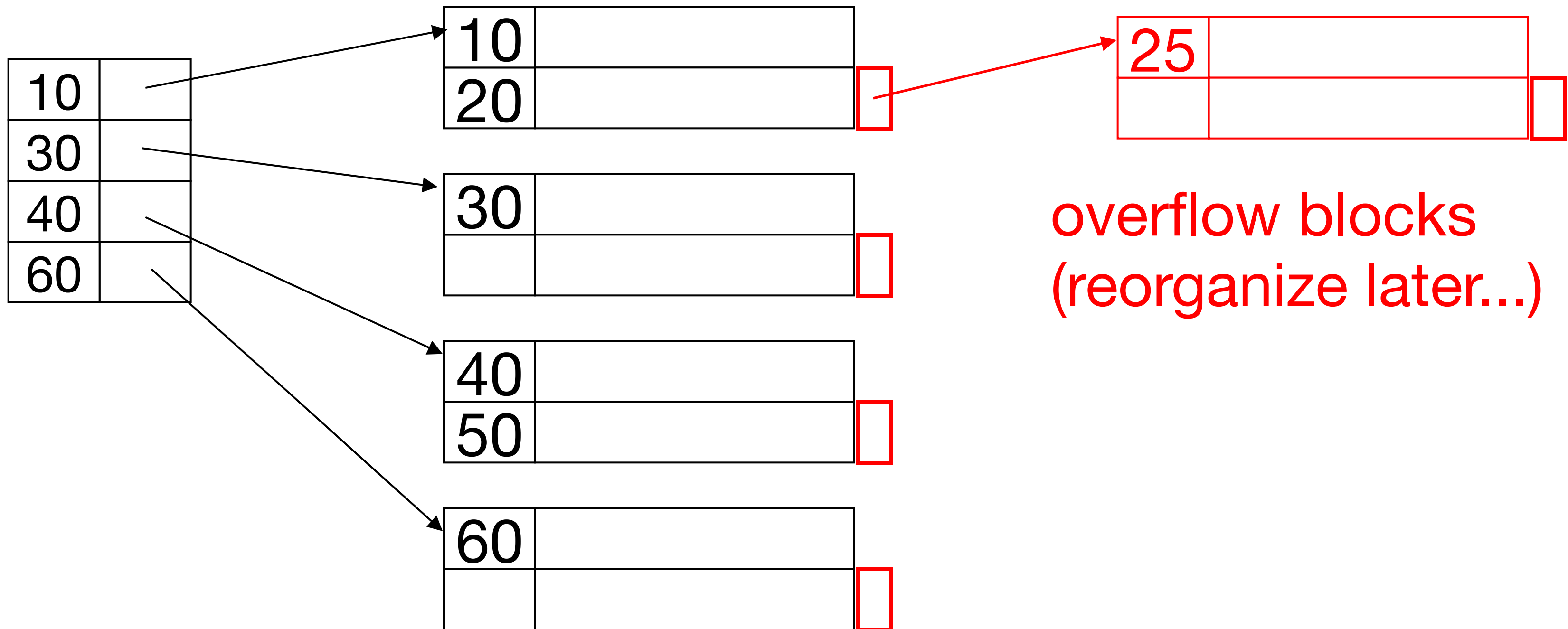
Insertion: Sparse Index

– insert record 25



Insertion: Sparse Index

– insert record 25



Secondary Indexes

Ordering field

30	
50	

20	
70	

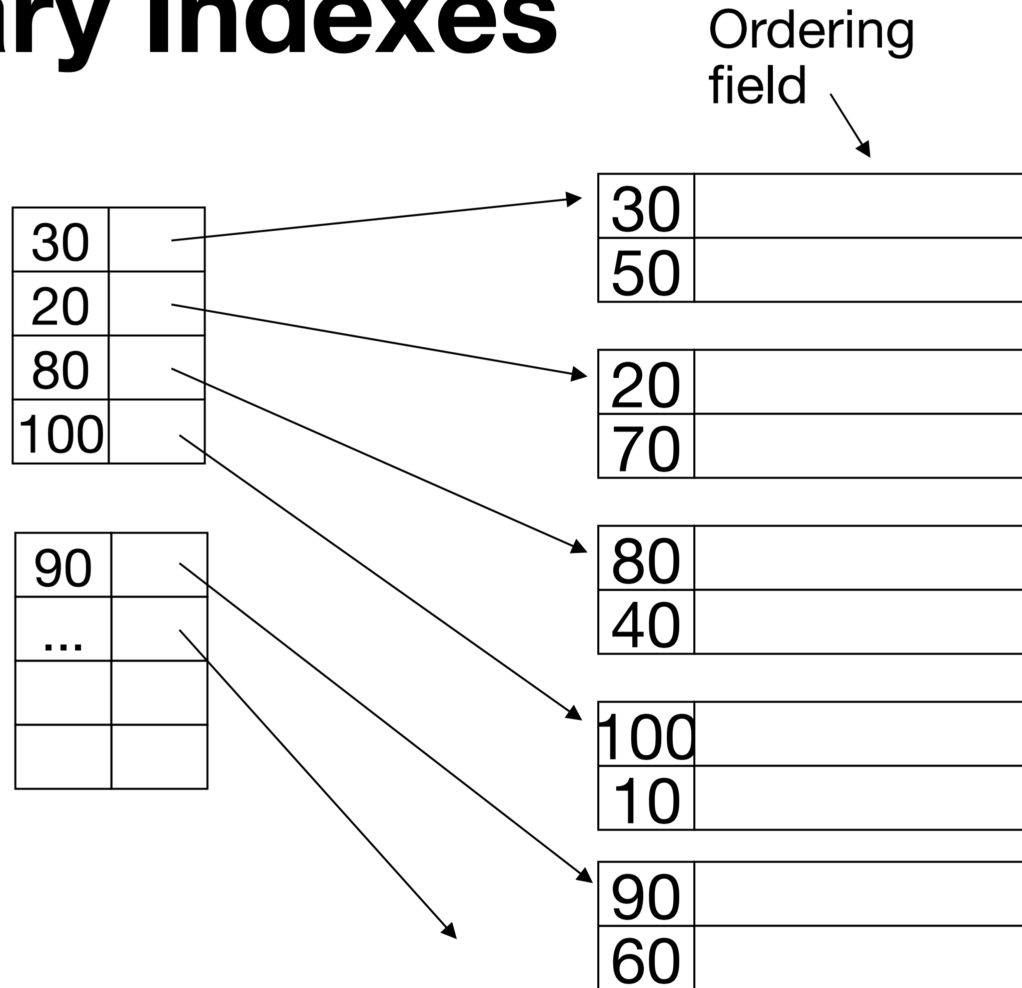
80	
40	

100	
10	

90	
60	

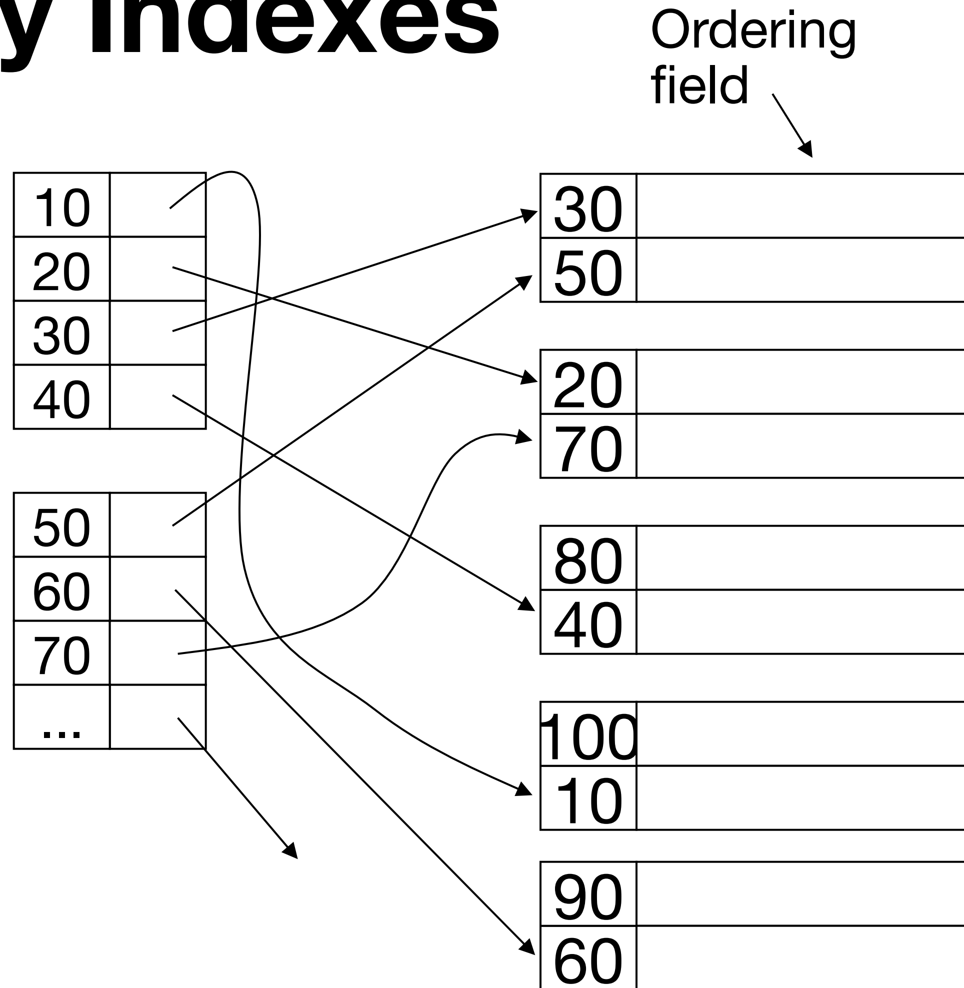
Secondary Indexes

Sparse index:



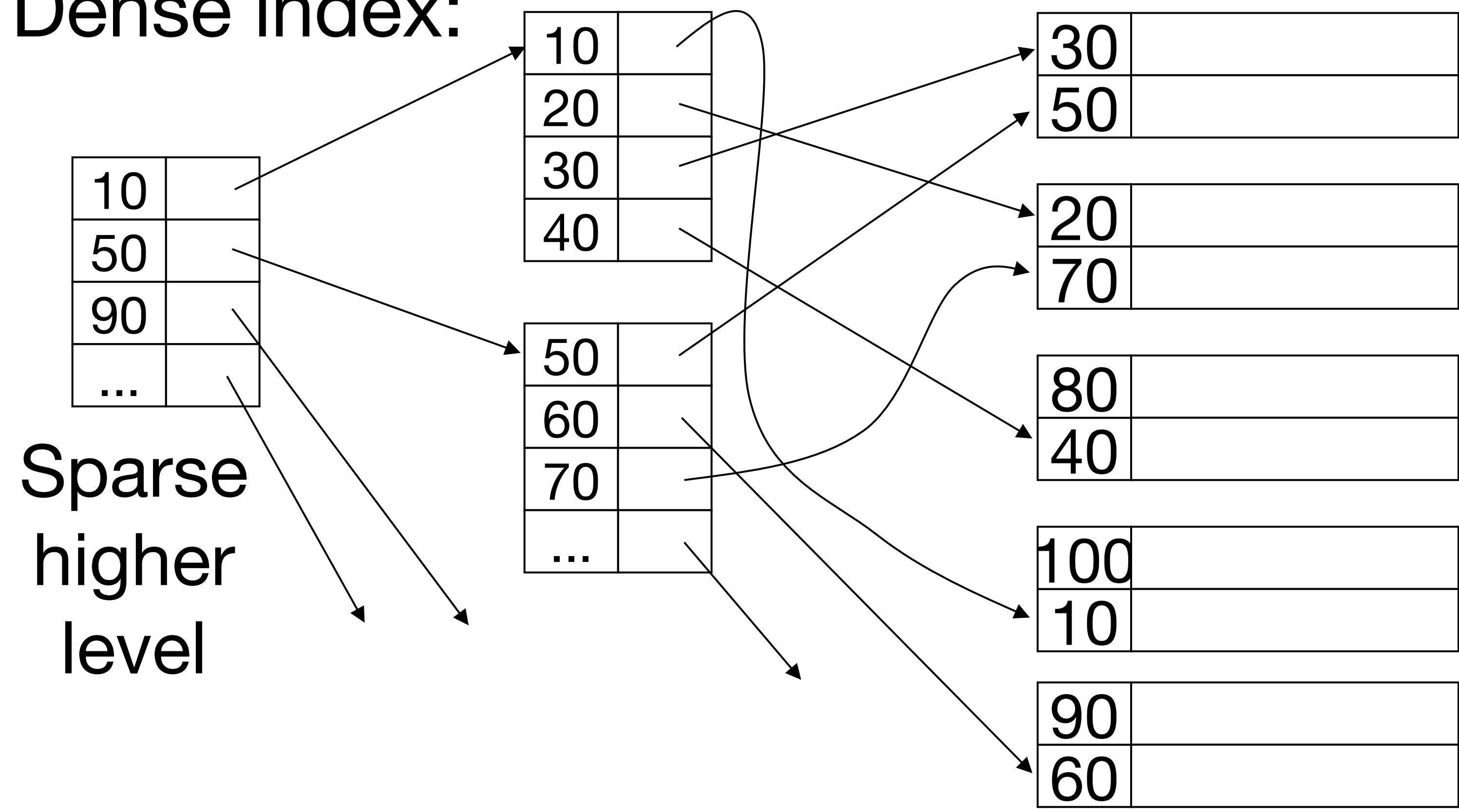
Secondary Indexes

Dense index:

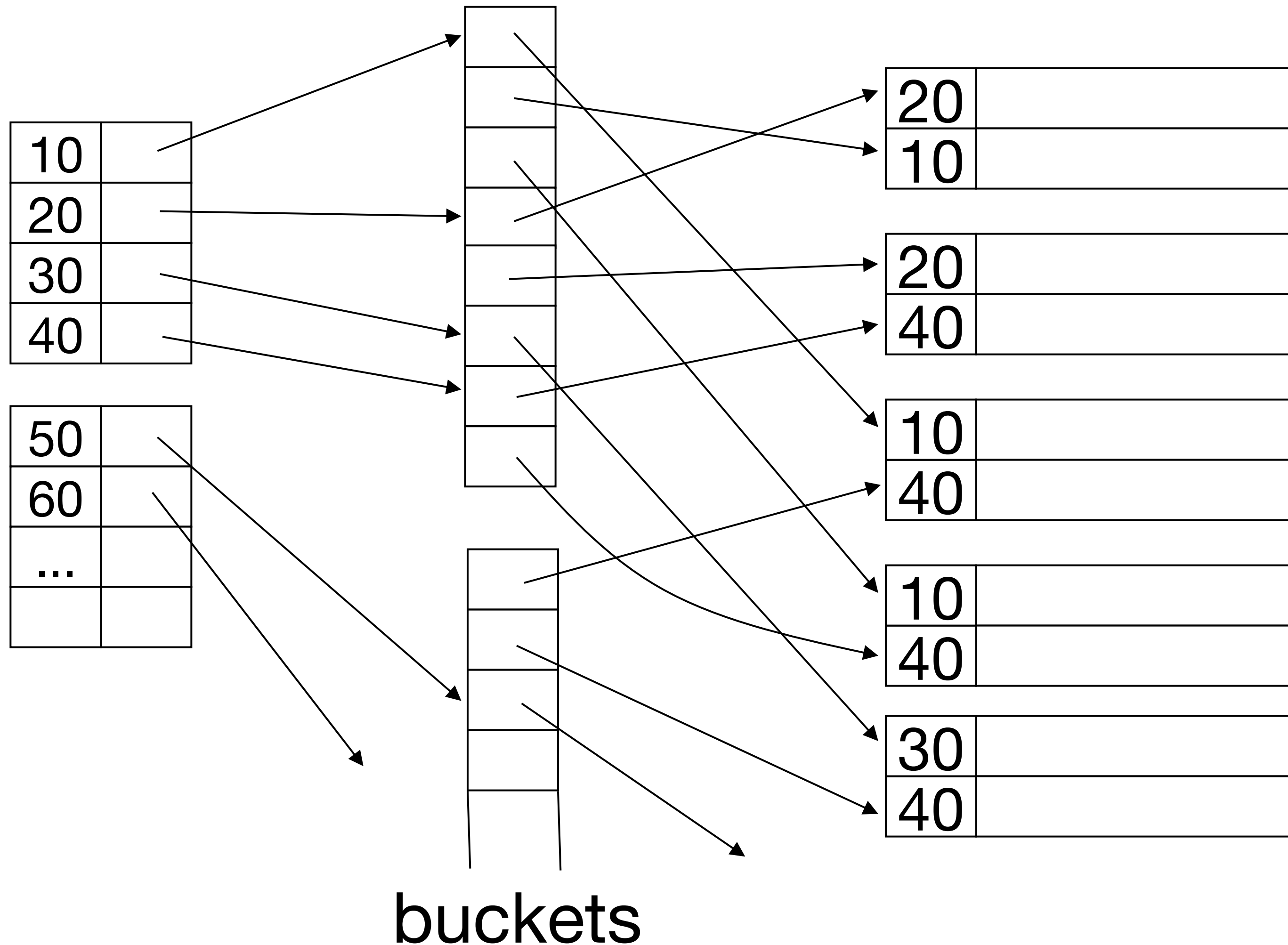


Secondary Indexes

Dense index:



Duplicate Values in Secondary Indexes



Conventional Indexes

Pros:

- Simple
- Index is sequential file (good for scans)

Cons:

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

Some Types of Indexes

Conventional indexes

B-trees

Hash indexes

Multi-key indexing

B-Trees

Another type of index

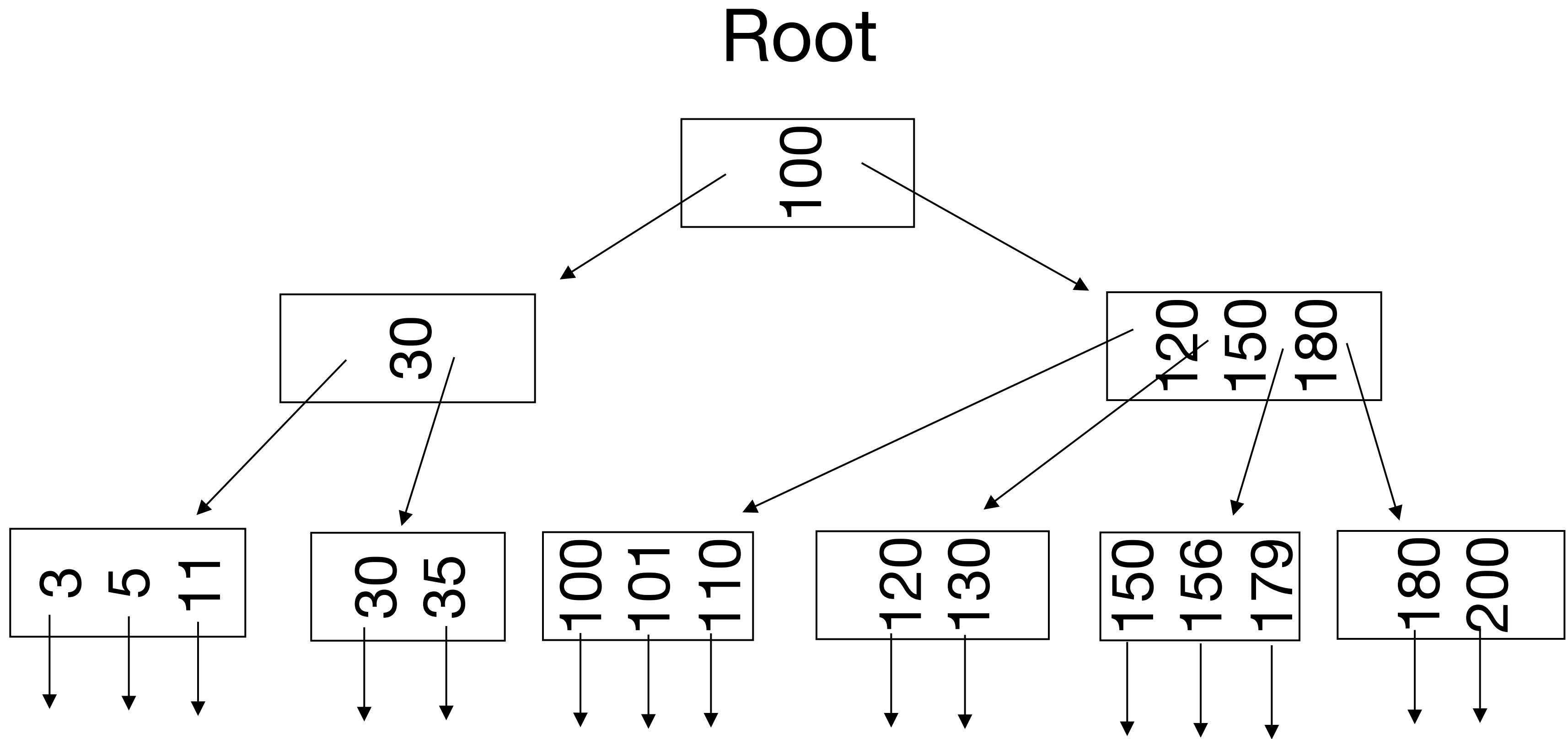
» Give up on sequentiality of index

» Try to get “balance”

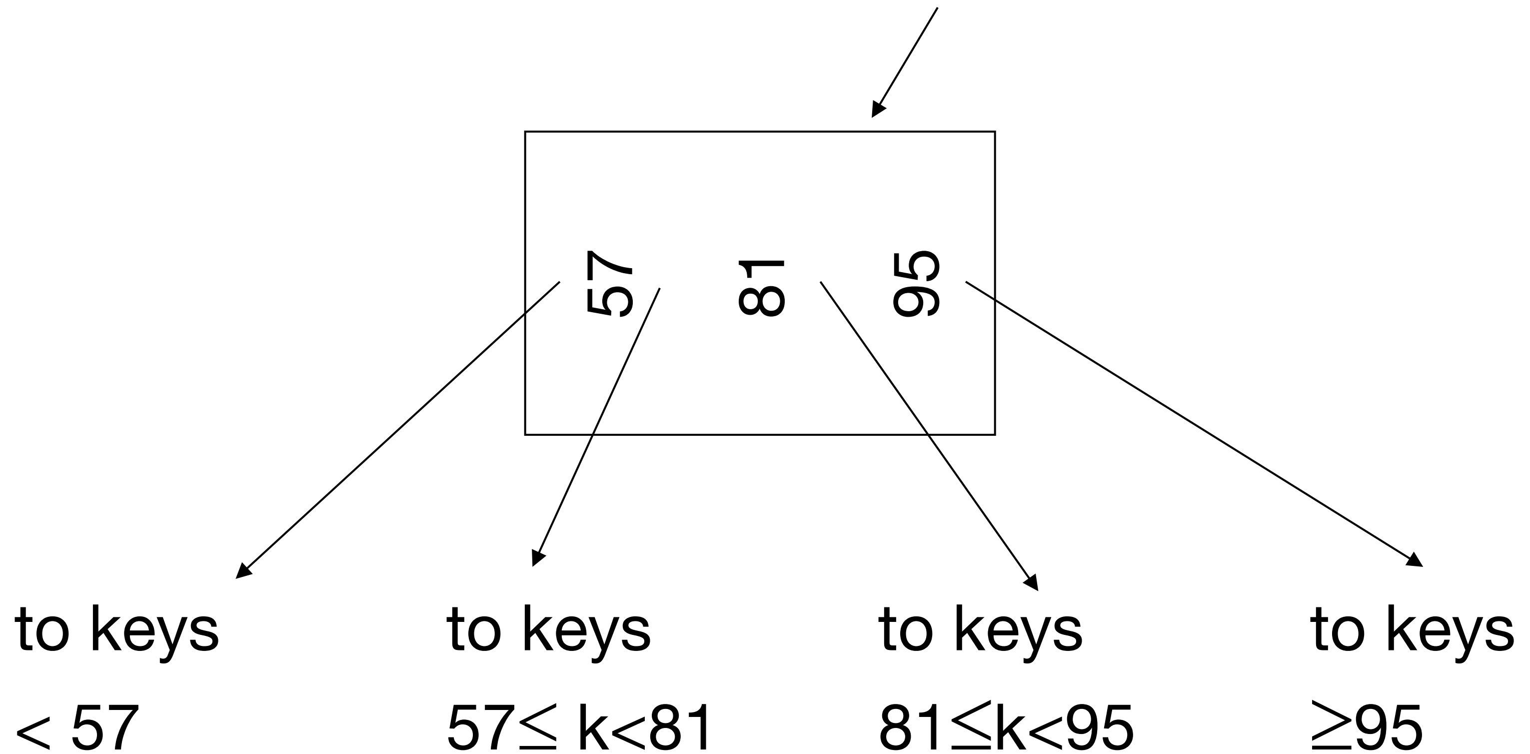
Note: the exact data structure we’ll look at is a **B+ tree**, but plain old “B-trees” are similar

B+ Tree Example

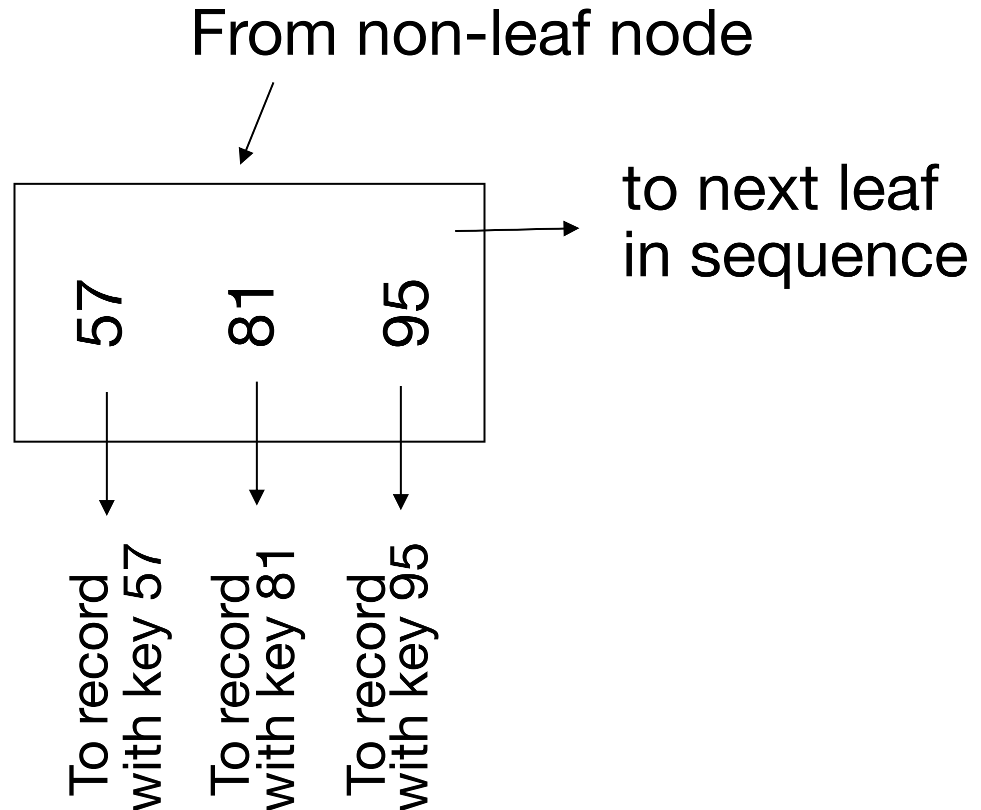
(n = 3)



Sample Non-Leaf



Sample Leaf Node



Size of Nodes on Disk

$$\begin{cases} n + 1 \text{ pointers} \\ n \text{ keys} \end{cases}$$

(Fixed size nodes)

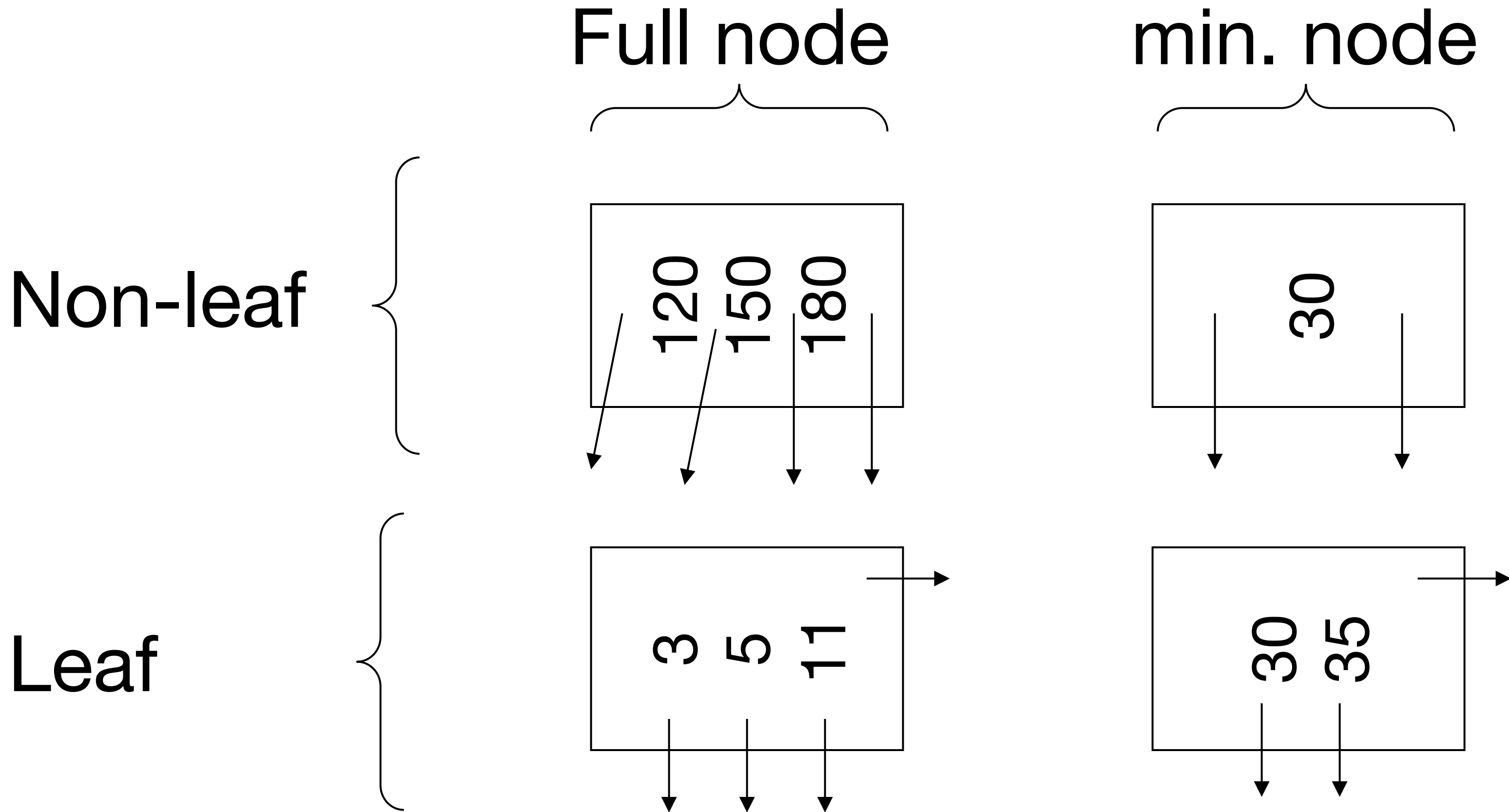
Don't Want Nodes to be Too Empty

Use at least

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

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

Example: n = 3



B+ Tree Rules (tree of order n)

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

2. Pointers in leaves point to records, except for "sequence pointer"

B+ Tree Rules (tree of order n)

(3) Number of pointers/keys for B+ tree:

	Max ptrs	Max keys	Min ptrs→data	Min keys
Non-leaf (non-root)	n+1	n	$\lceil(n+1)/2\rceil$	$\lceil(n+1)/2\rceil$-1
Leaf (non-root)	n+1	n	$\lfloor(n+1)/2\rfloor$	$\lfloor(n+1)/2\rfloor$
Root	n+1	n	2*	1

* When there is only one record in the B+ tree, min pointers in the root is 1 (the other pointers are null)

Insert Into B+ Tree

(a) simple case
- » space available in leaf

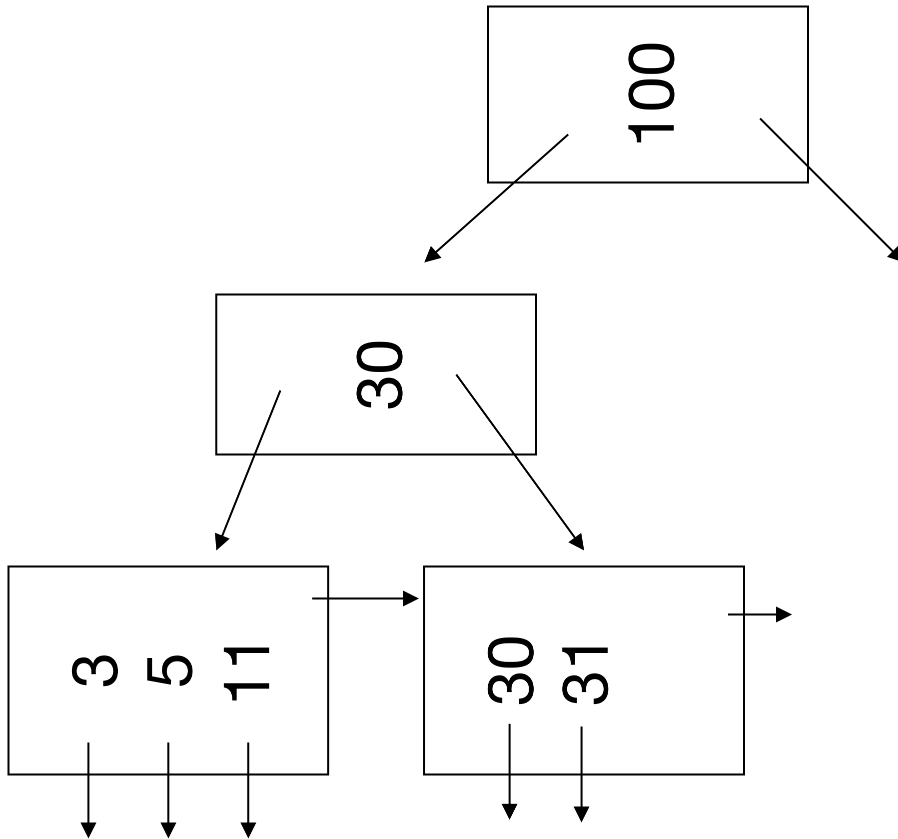
(b) leaf overflow

(c) non-leaf overflow

(d) new root

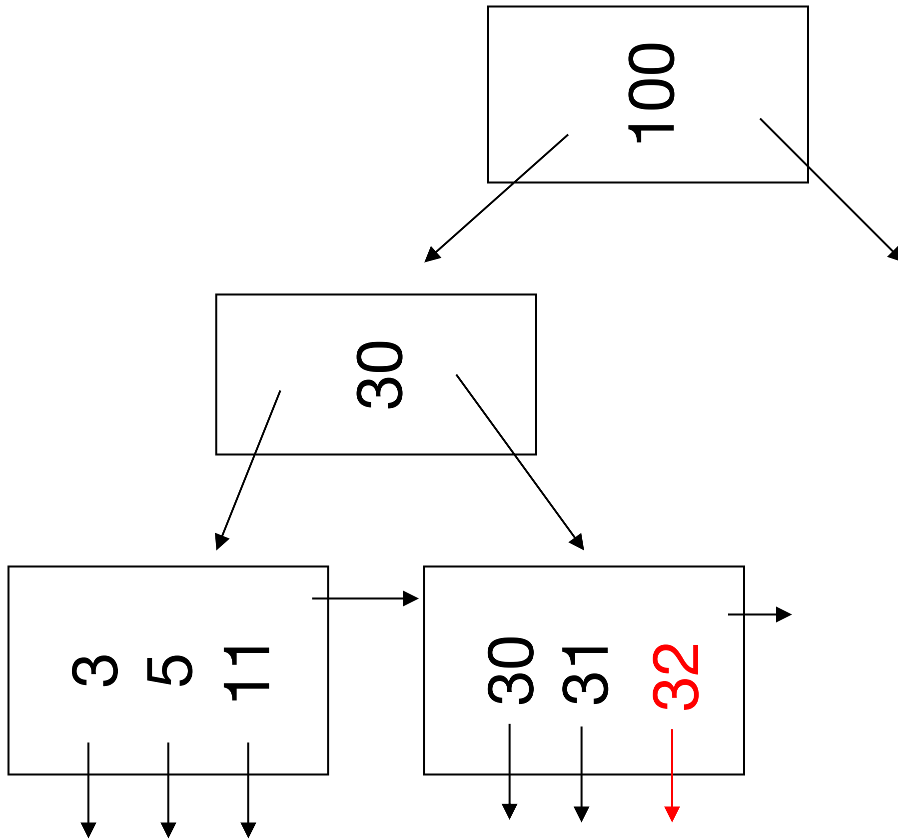
(a) Insert key = 32

n=3



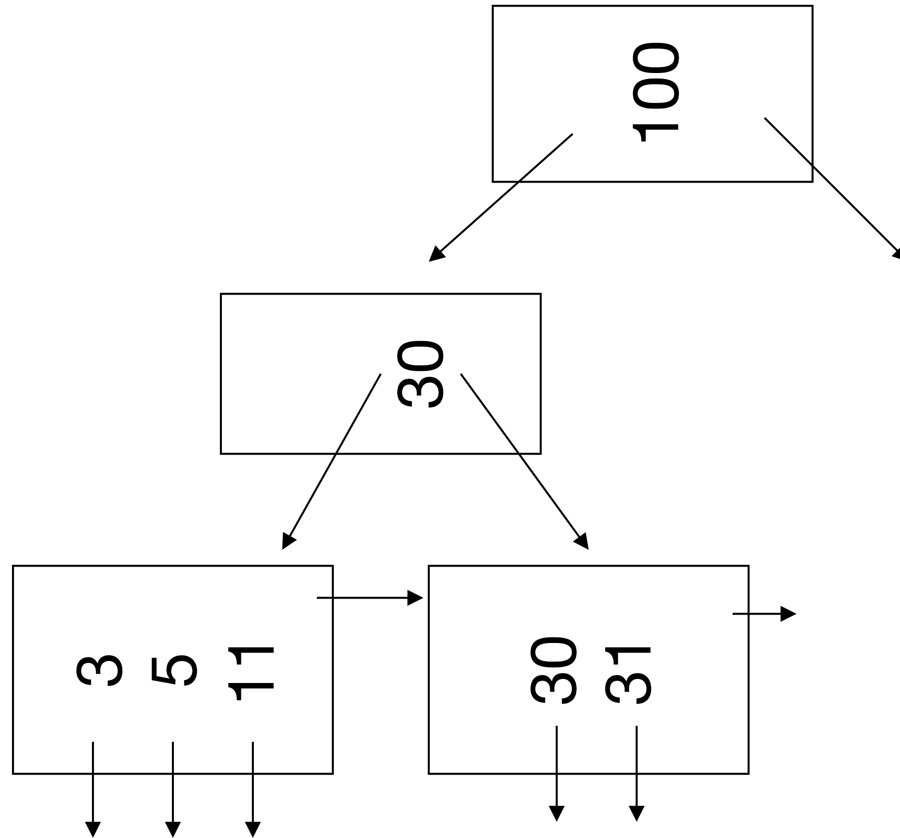
(a) Insert key = 32

n=3



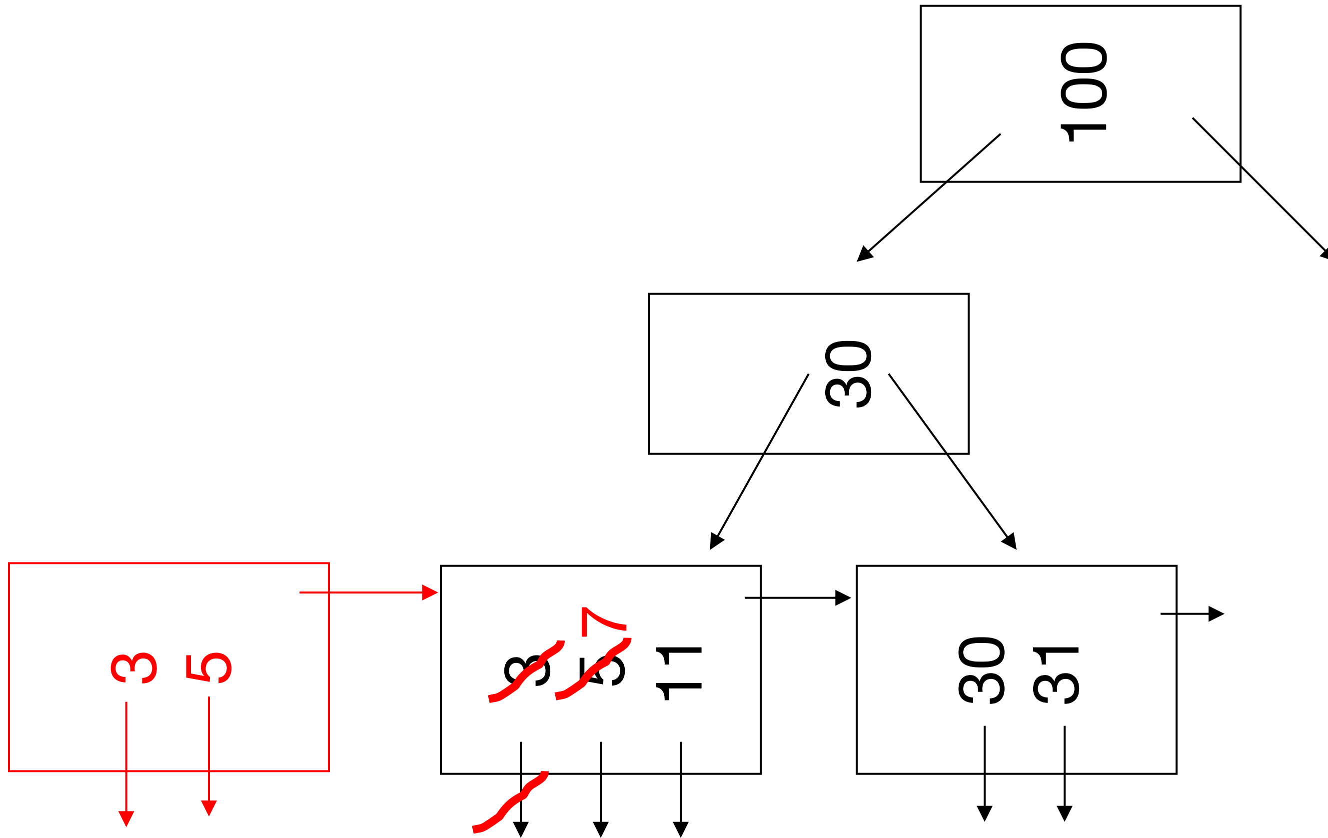
(a) Insert key = 7

n=3



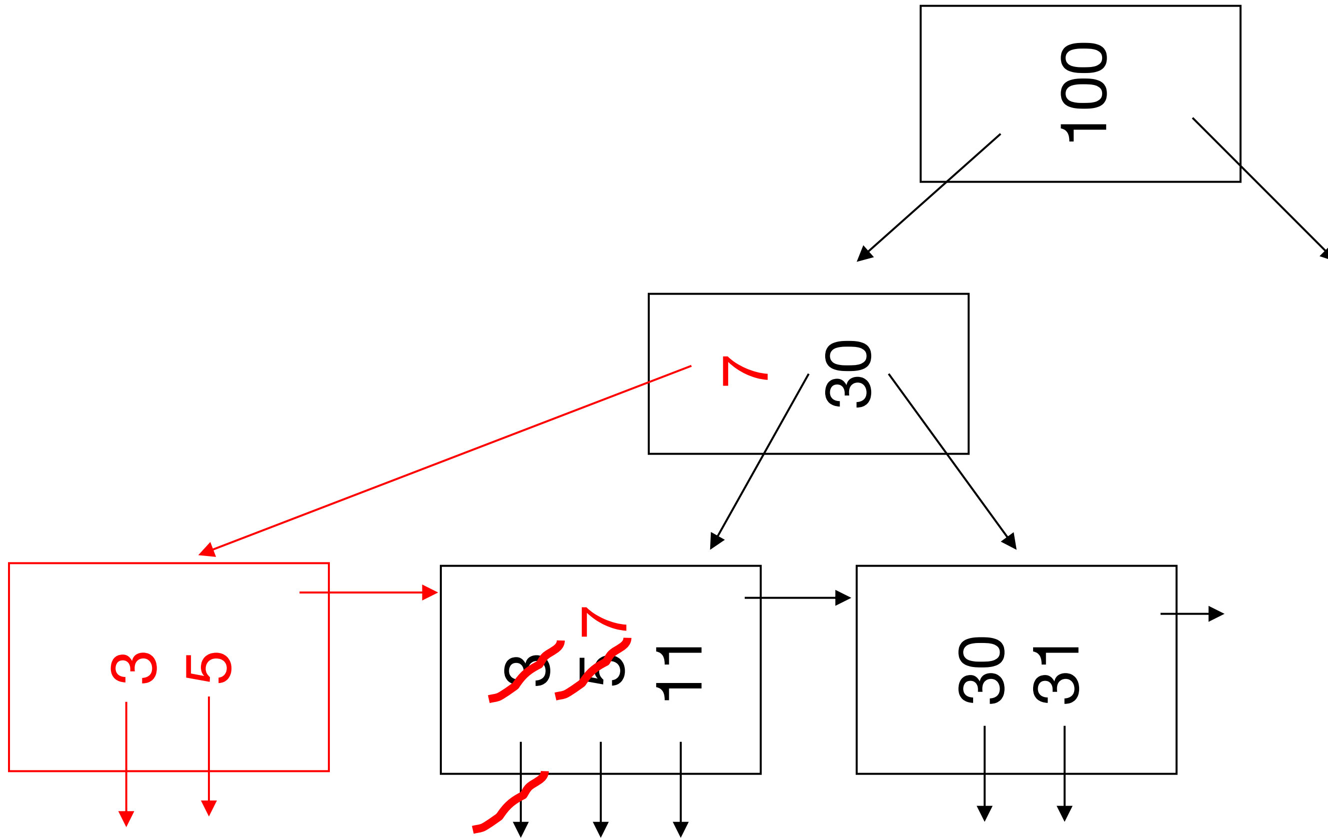
(a) Insert key = 7

n=3



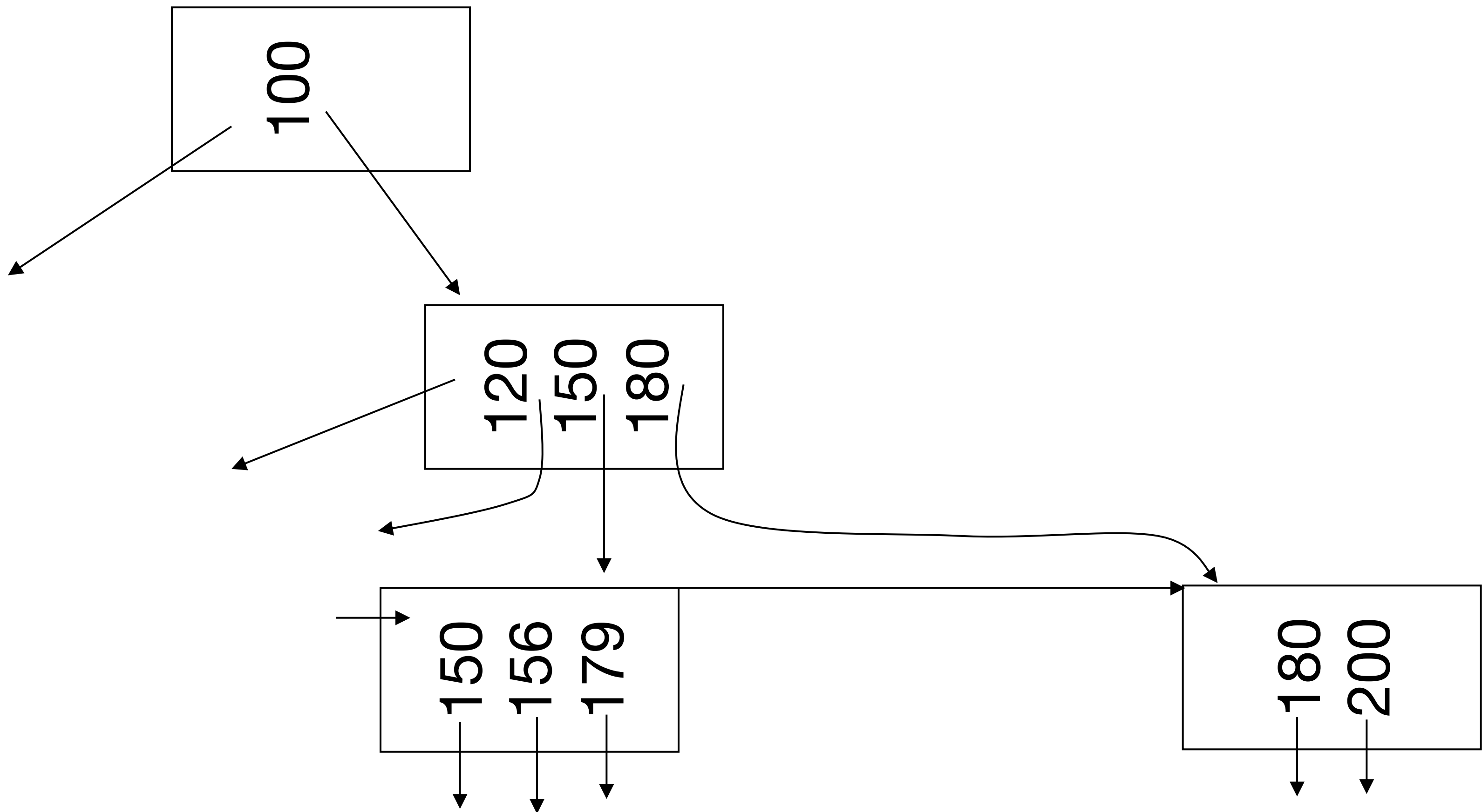
(a) Insert key = 7

n=3



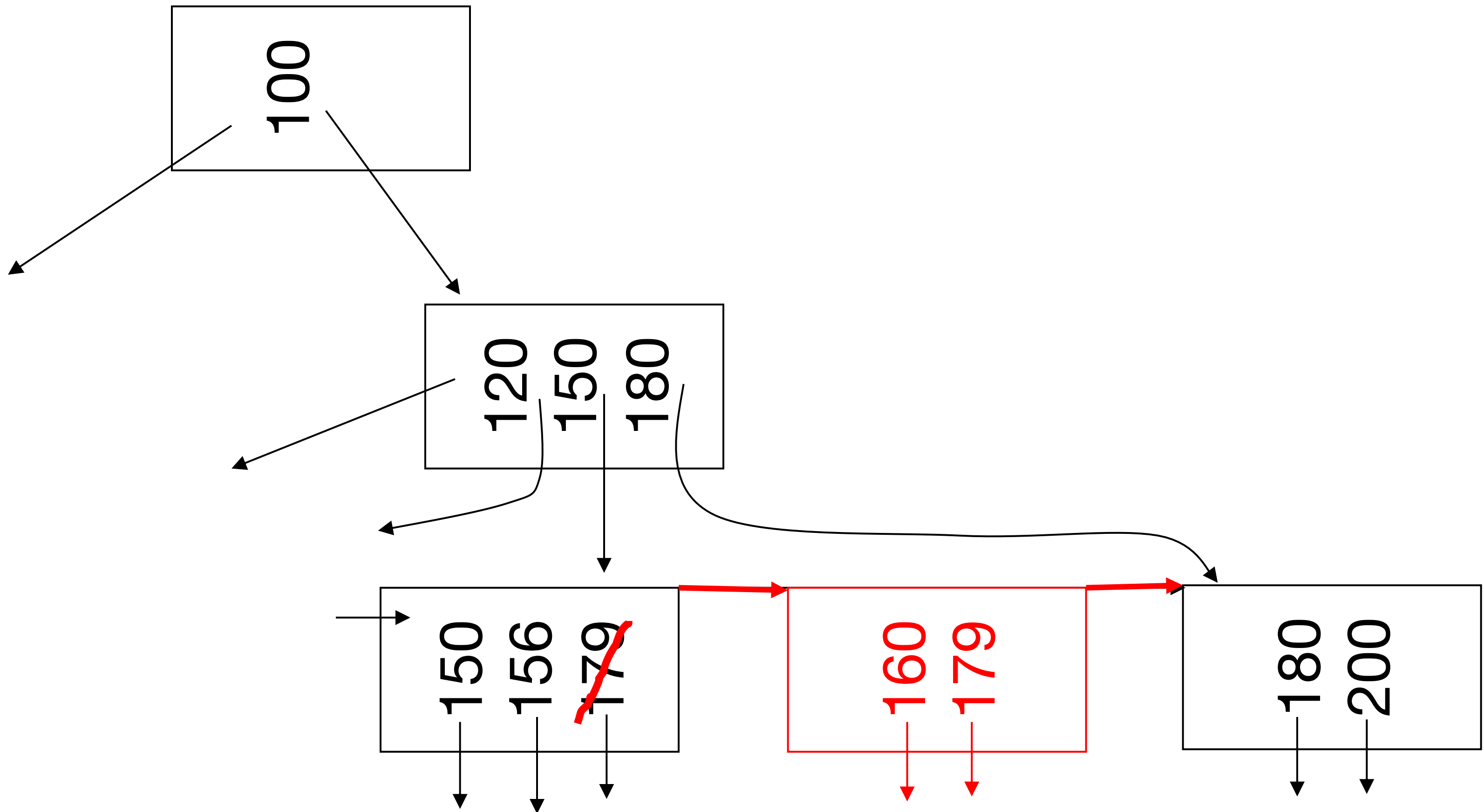
(c) Insert key = 160

n=3



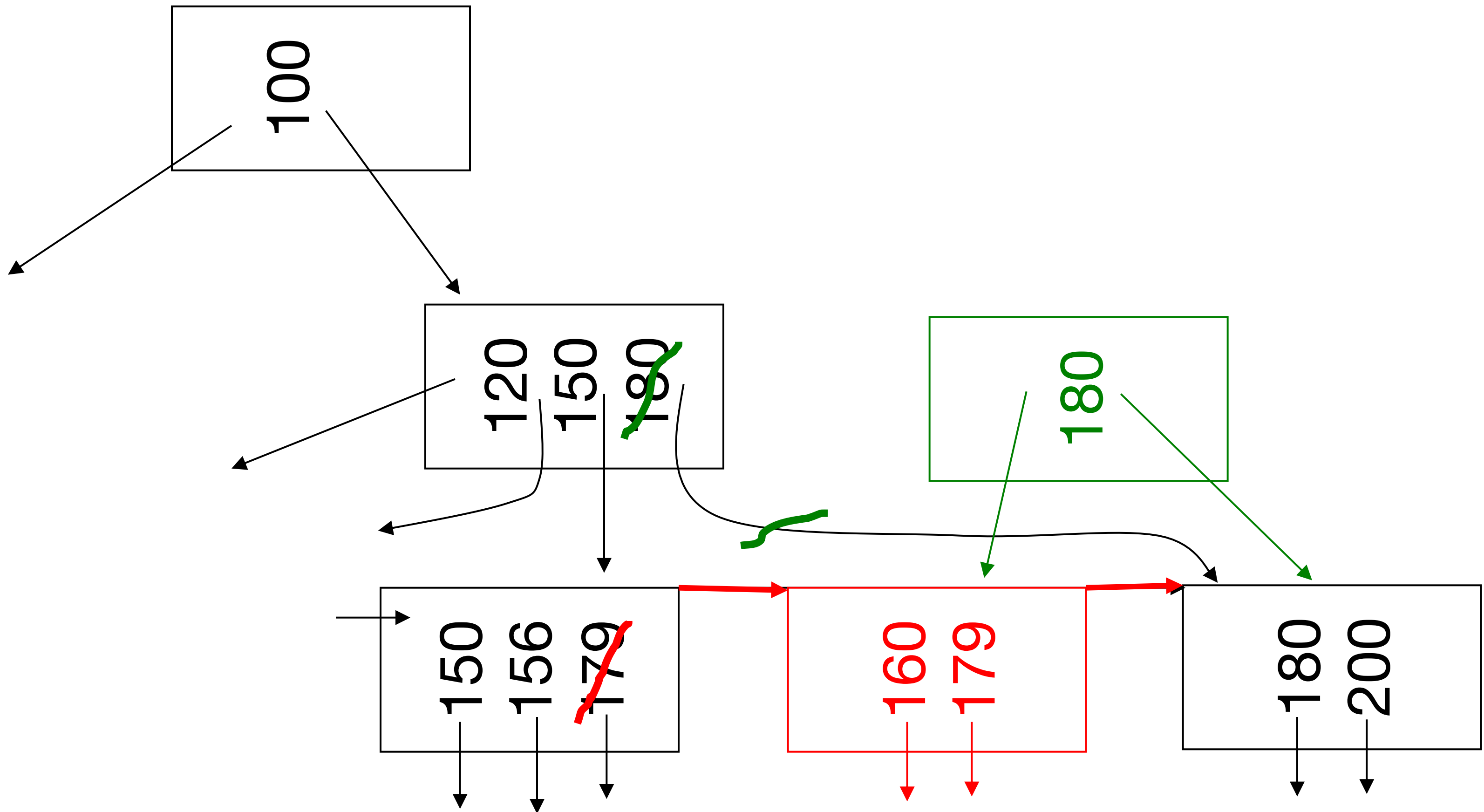
(c) Insert key = 160

n=3



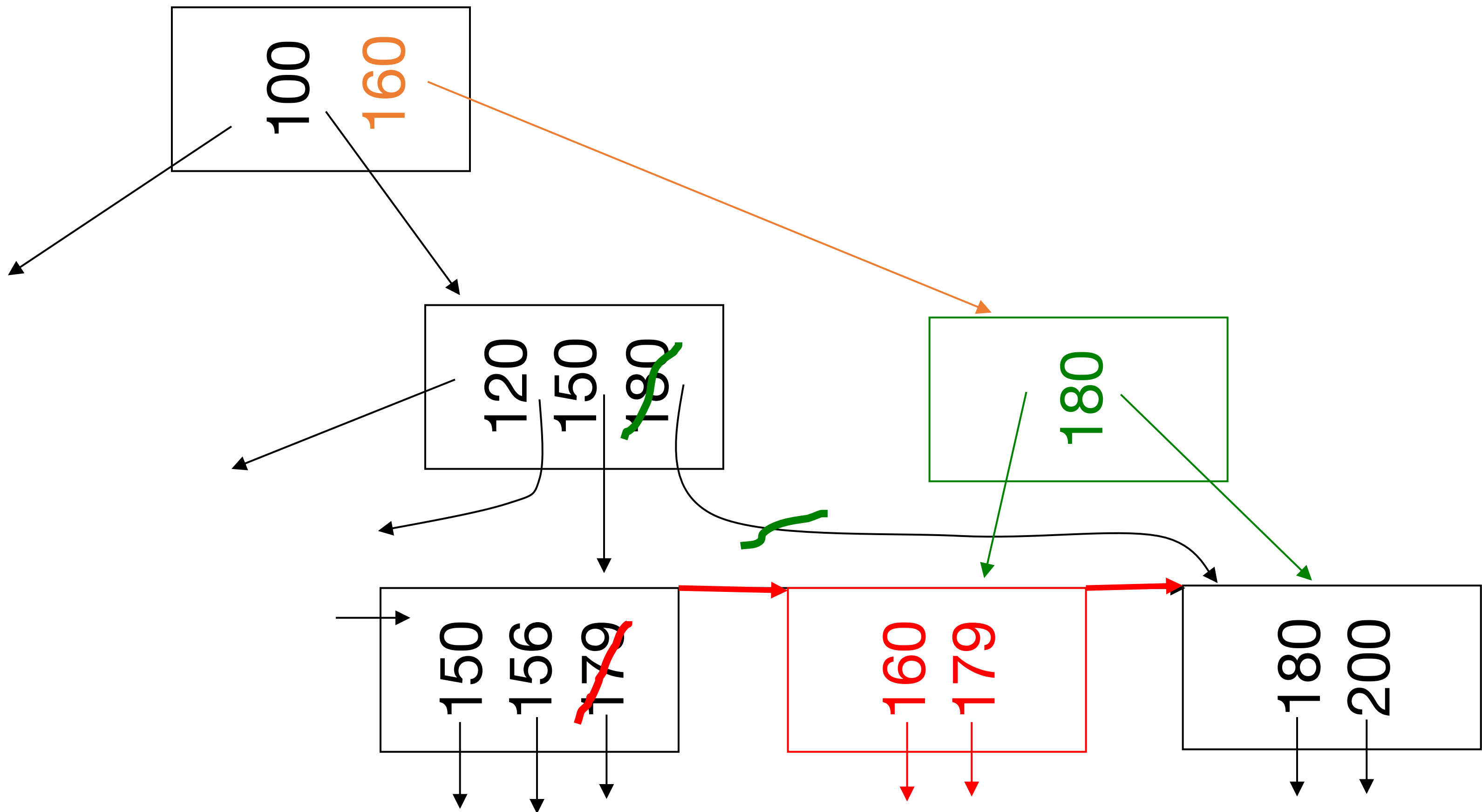
(c) Insert key = 160

n=3



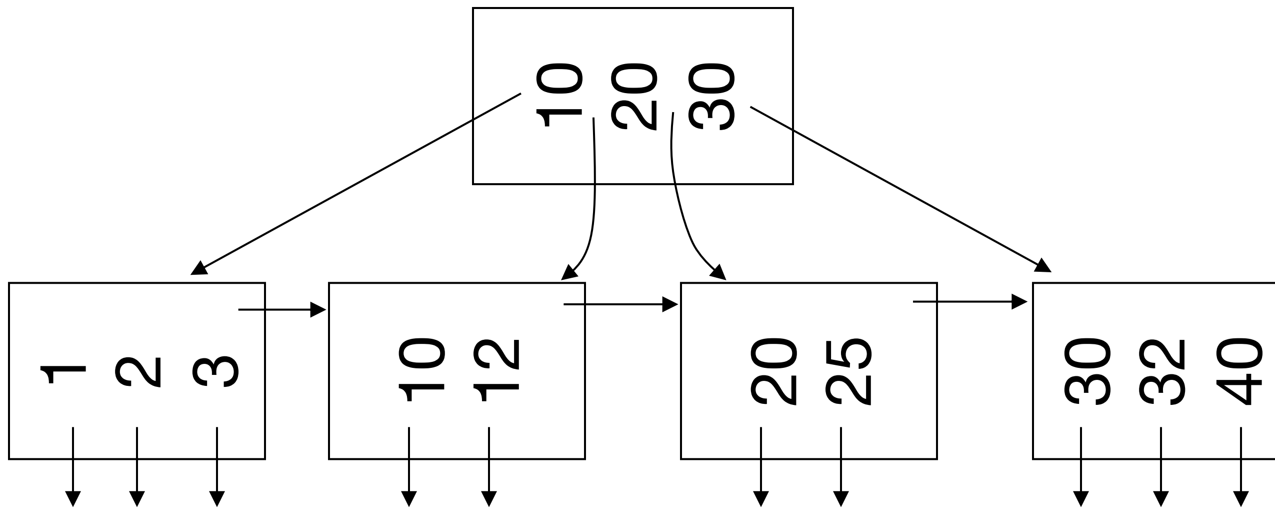
(c) Insert key = 160

n=3



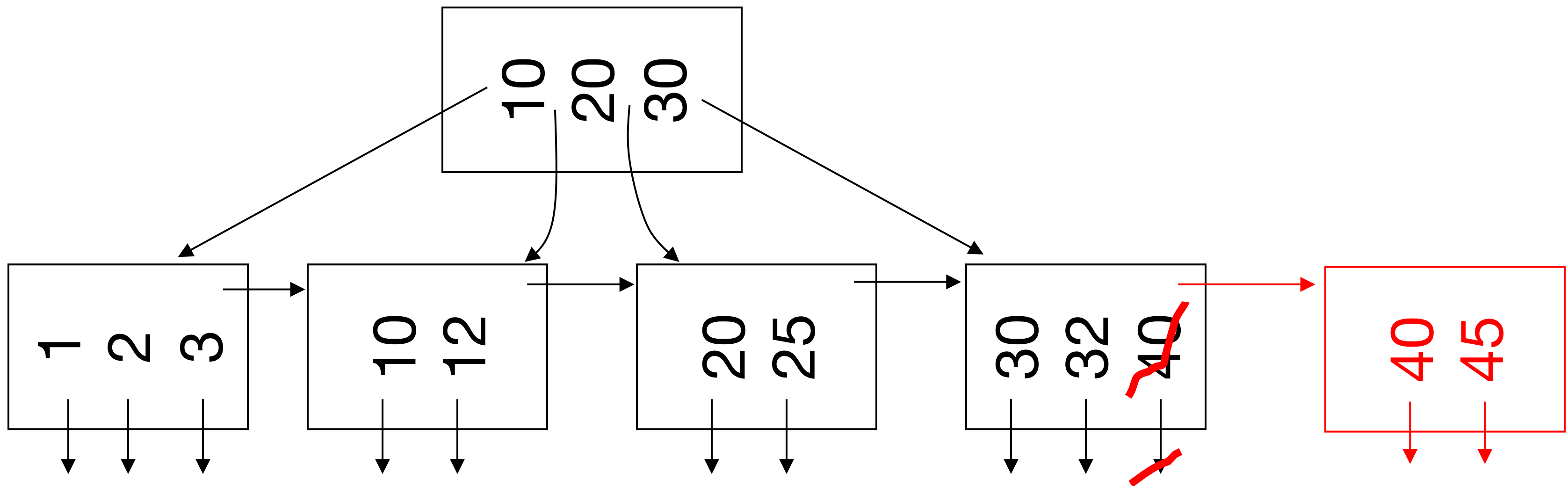
(d) New root, insert 45

n=3



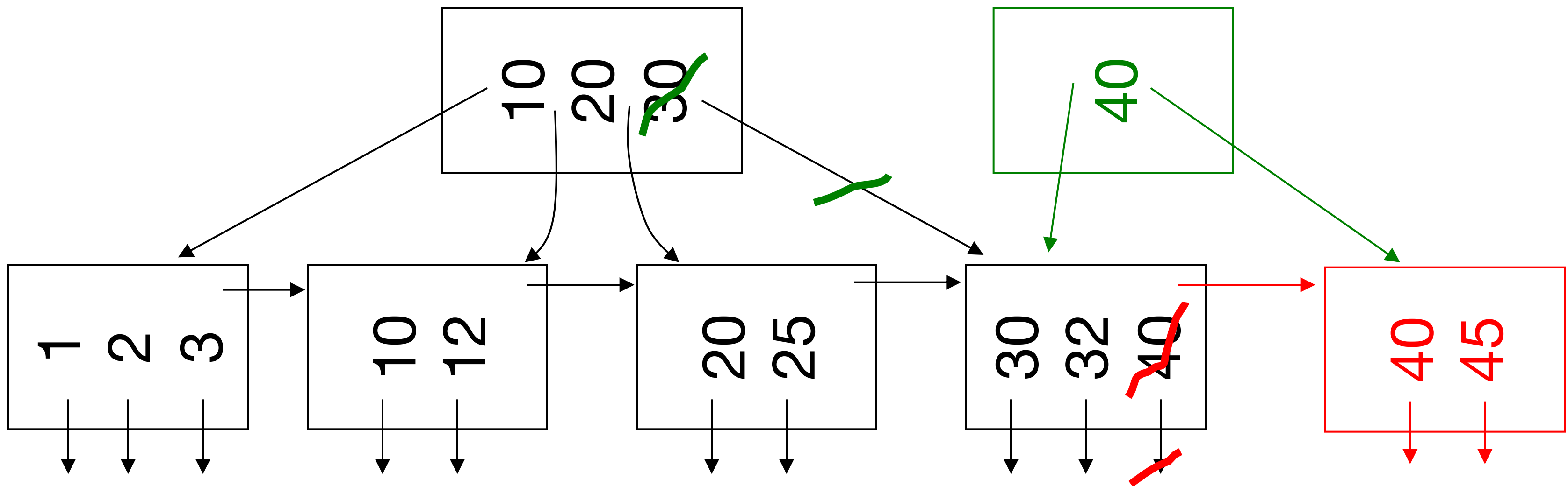
(d) New root, insert 45

n=3



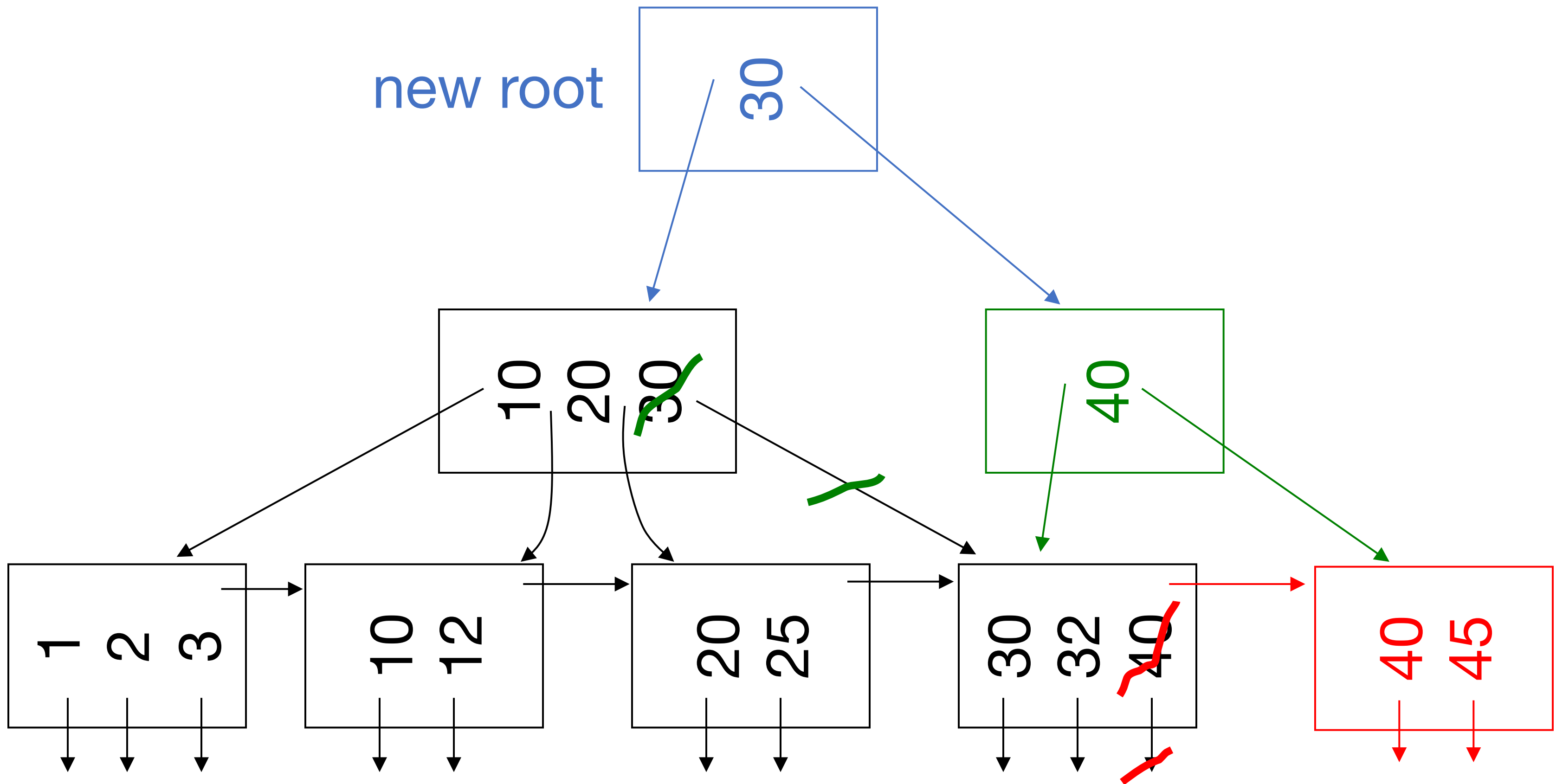
(d) New root, insert 45

n=3



(d) New root, insert 45

n=3



Deletion from B+tree

(a) Simple case: no example

(b) Coalesce with neighbor (sibling)

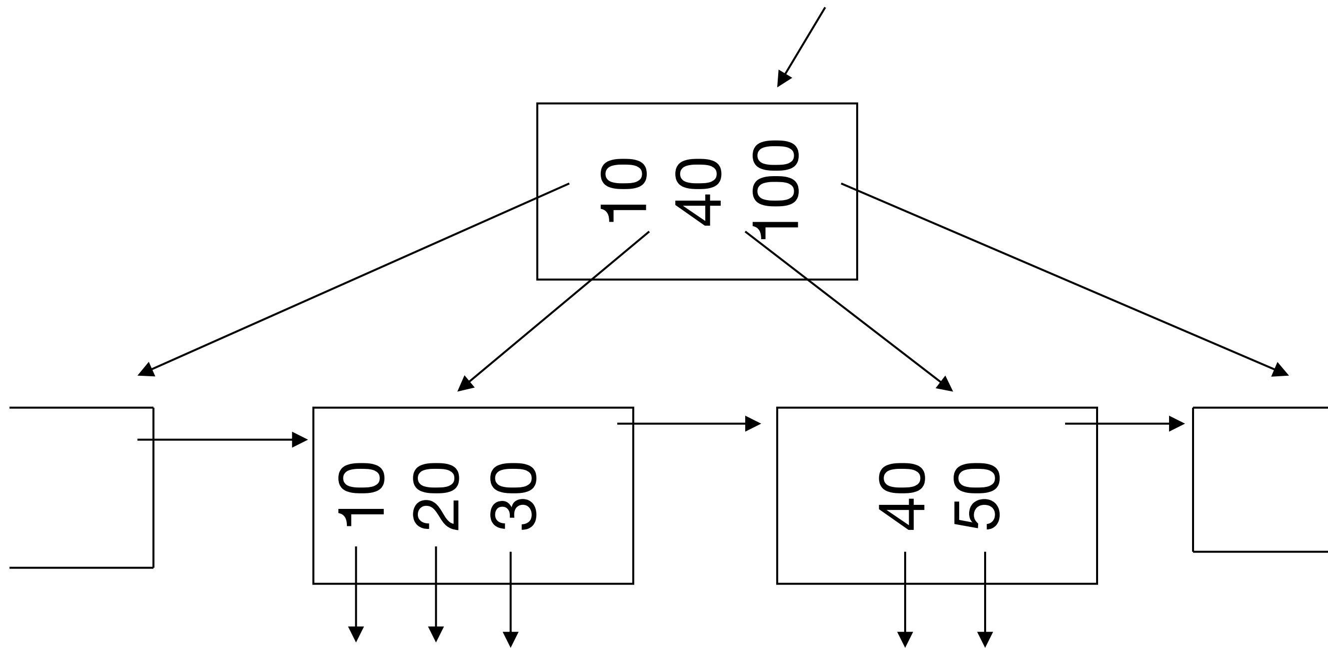
(c) Re-distribute keys

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

(b) Coalesce with sibling

» Delete 50

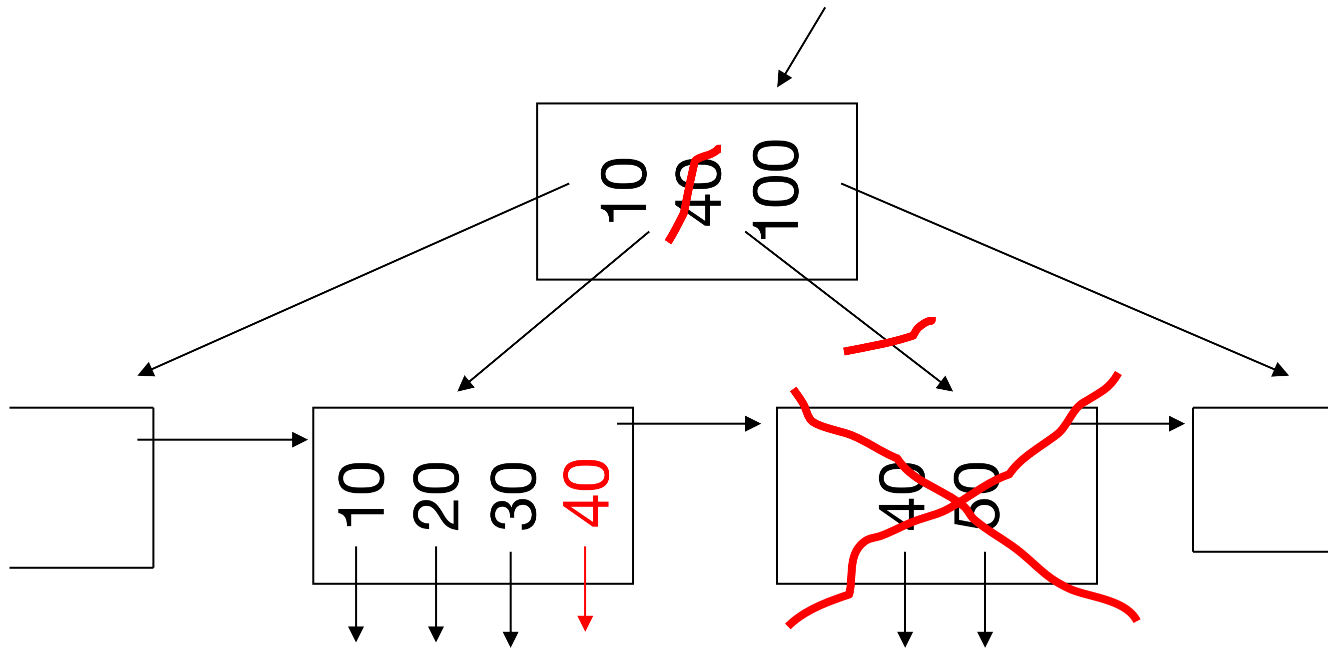
n=4



(b) Coalesce with sibling

» Delete 50

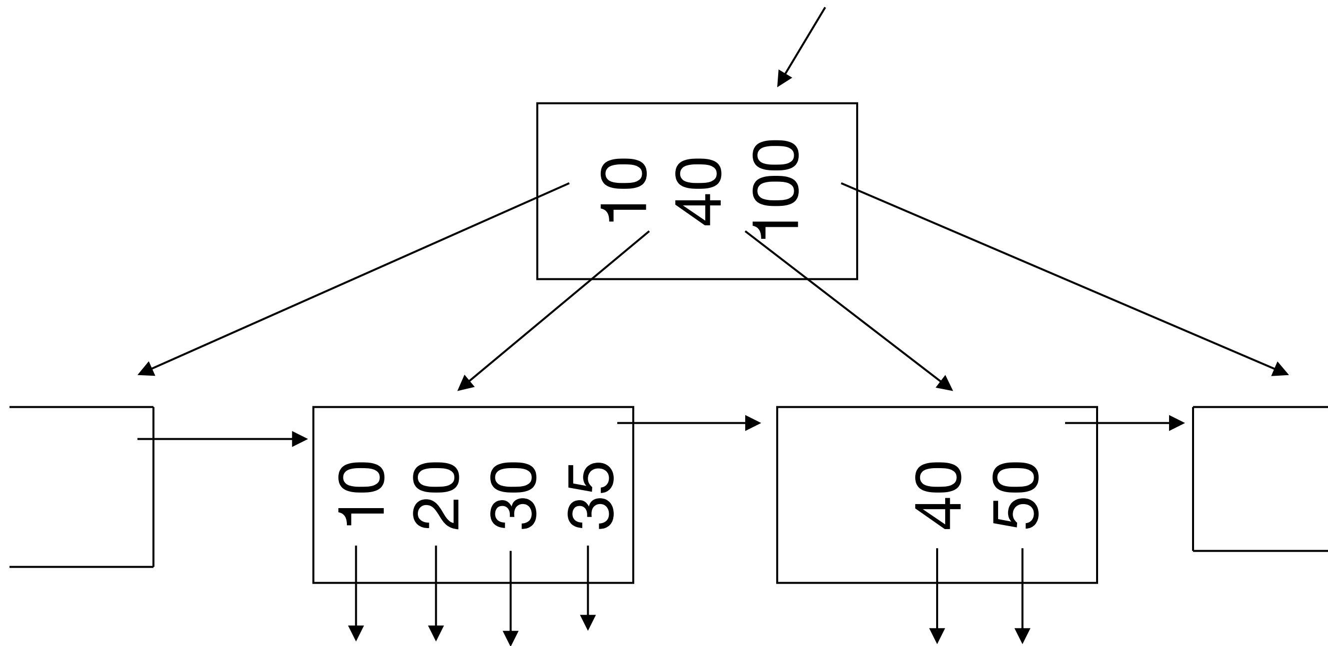
n=4



(c) Redistribute keys

» Delete 50

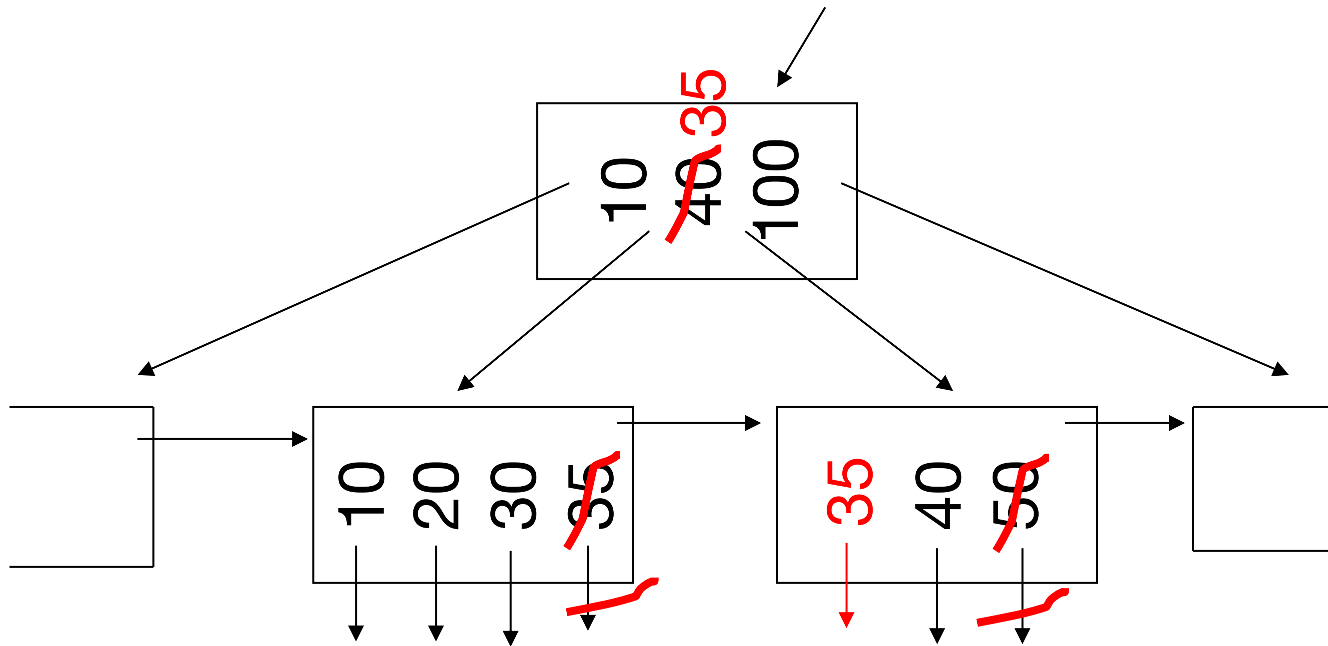
n=4



(c) Redistribute keys

» Delete 50

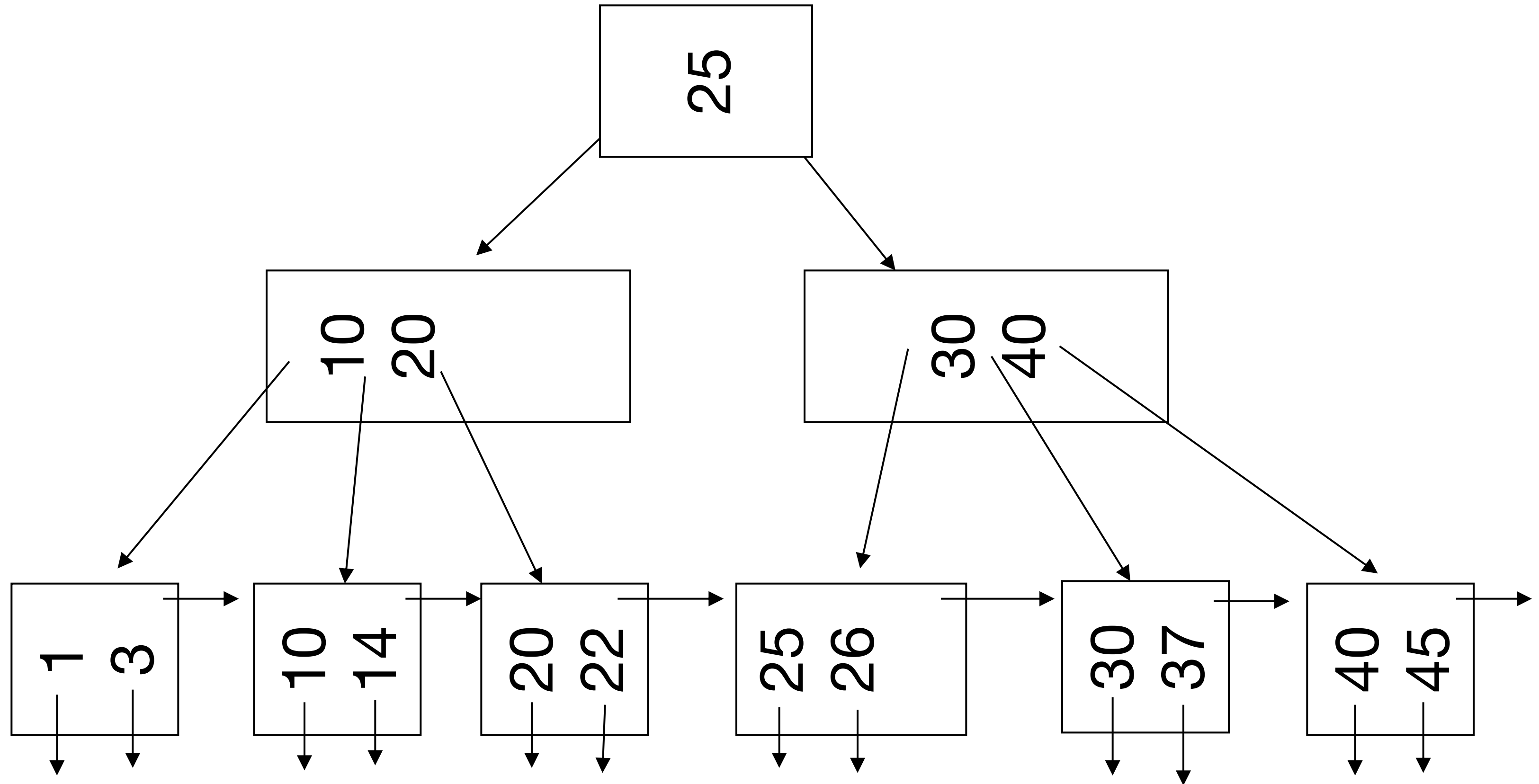
n=4



(d) Non-leaf coalesce

– Delete 37

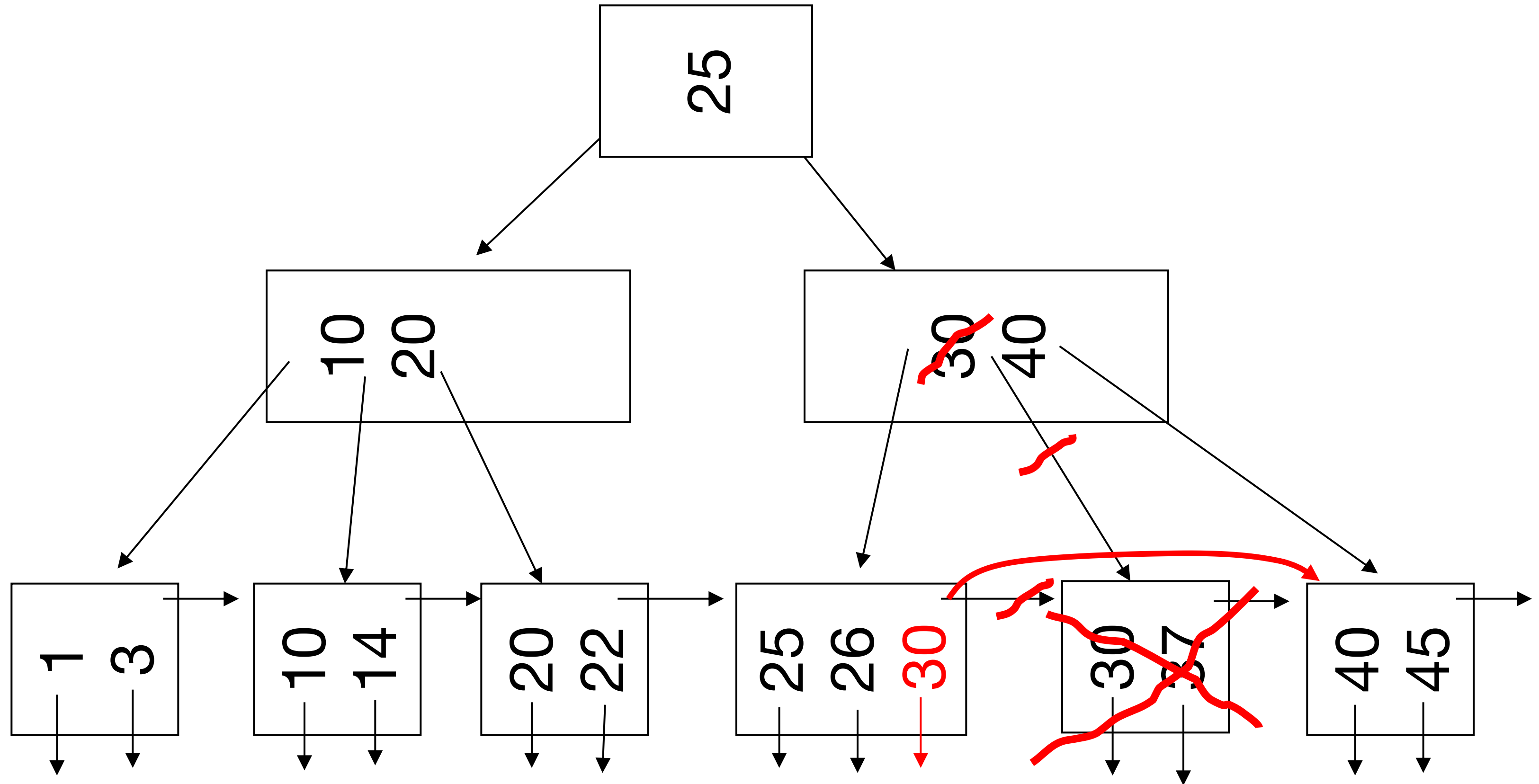
n=4



(d) Non-leaf coalesce

– Delete 37

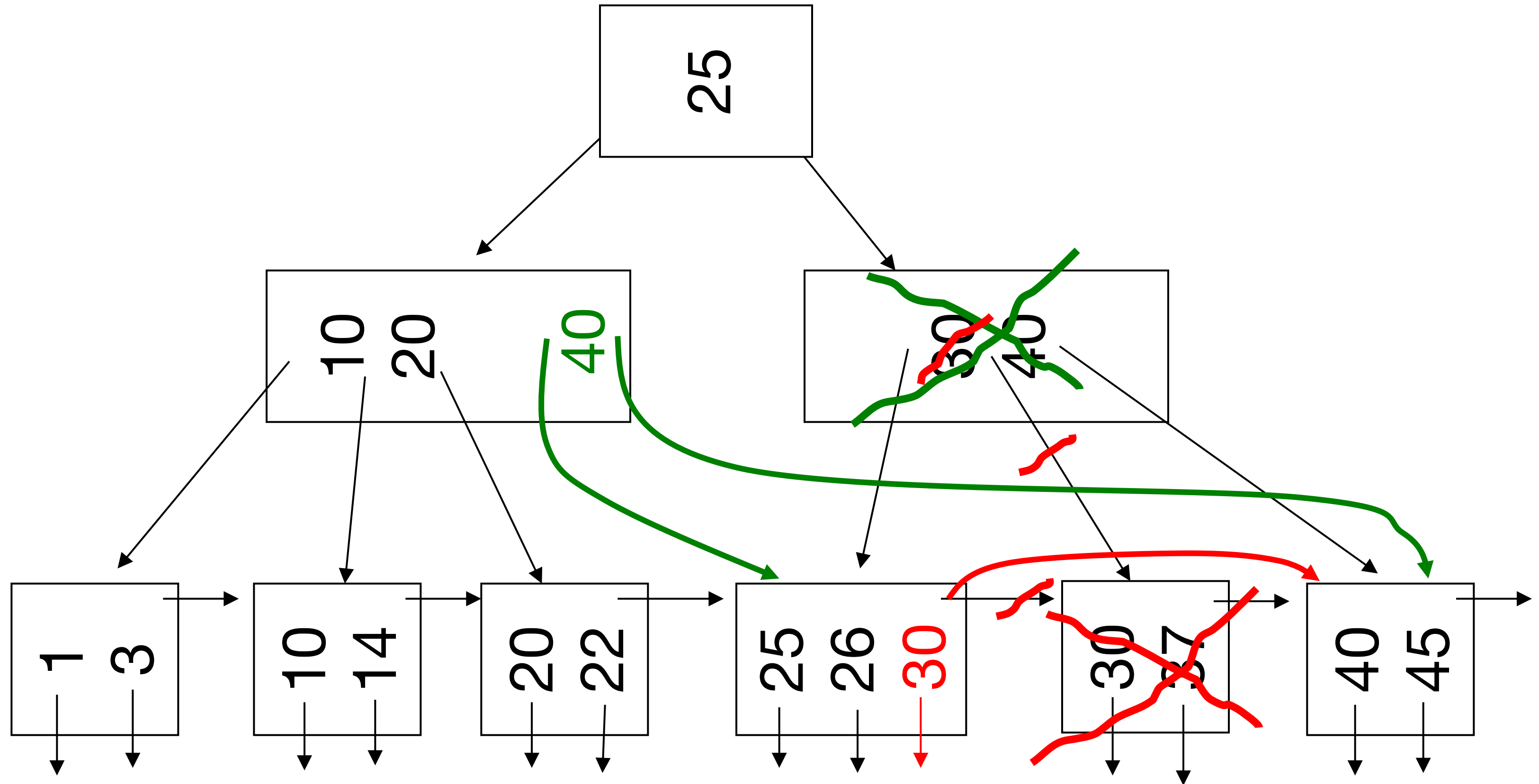
n=4



(d) Non-leaf coalesce

– Delete 37

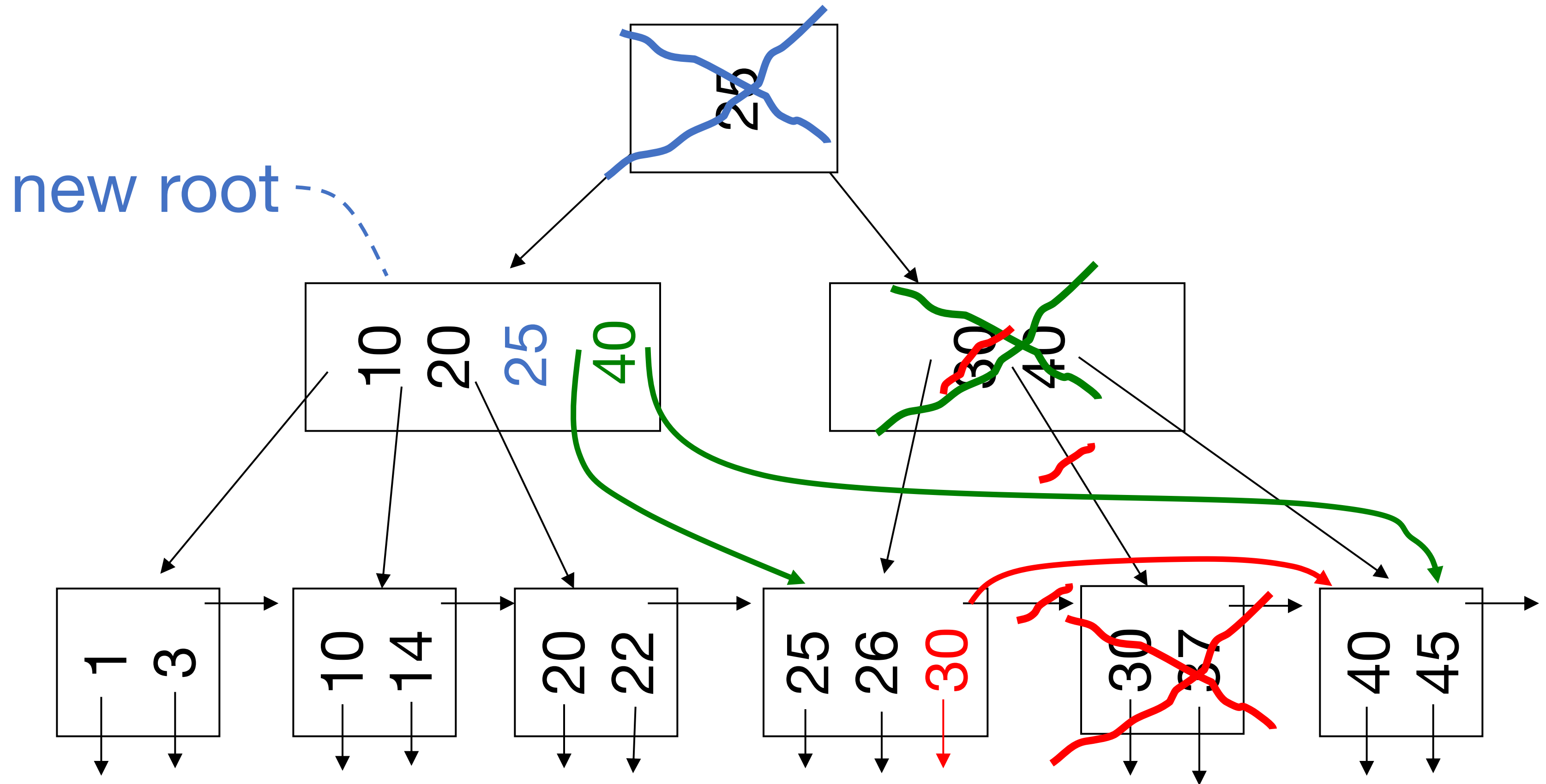
n=4



(d) Non-leaf coalesce

– Delete 37

n=4



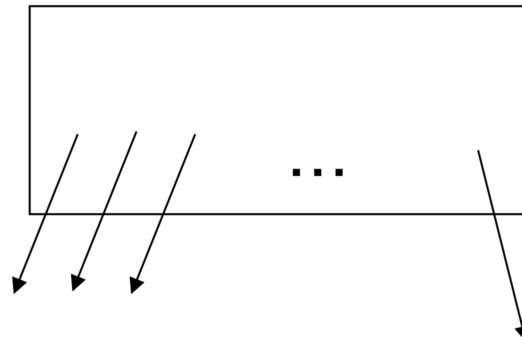
B+ Tree Deletion in Practice

Often, coalescing is not implemented

» Too hard and not worth it! Most datasets only tend to grow in size over time.

Interesting Problem:

For B+ tree, how large should n be?



n is number of keys / node

With modern hardware, get n = 1000 or more

Summary

Wide range of indexes for different data types and queries (e.g. range vs exact)

Key concerns: query time, cost to update, and size of index

Next: given all these storage data structures, how do we run our queries?