Data Storage & Indexes

Instructor: Matei Zaharia
cs245.stanford.edu

Outline

Co-designing storage and compute (paper)

Indexes

Outline

Co-designing storage and compute (paper)

Indexes

C-Store Storage

The storage construct was a “projection”; what does that mean?

C-Store Compression

Five types of compression:

- » Null suppression
- » Dictionary encoding
- » Run-length encoding
- » Bit-vector encoding
- » Lempel-Ziv

Tradeoff: size vs ease of computation

API for Compressed Blocks

Properties	Iterator Access	Block Information
isOneValue()	getNext()	getSize()
isValueSorted()	asArray()	getStartValue()
isPosContig()		getEndPosition()

Table 1: Compressed Block API

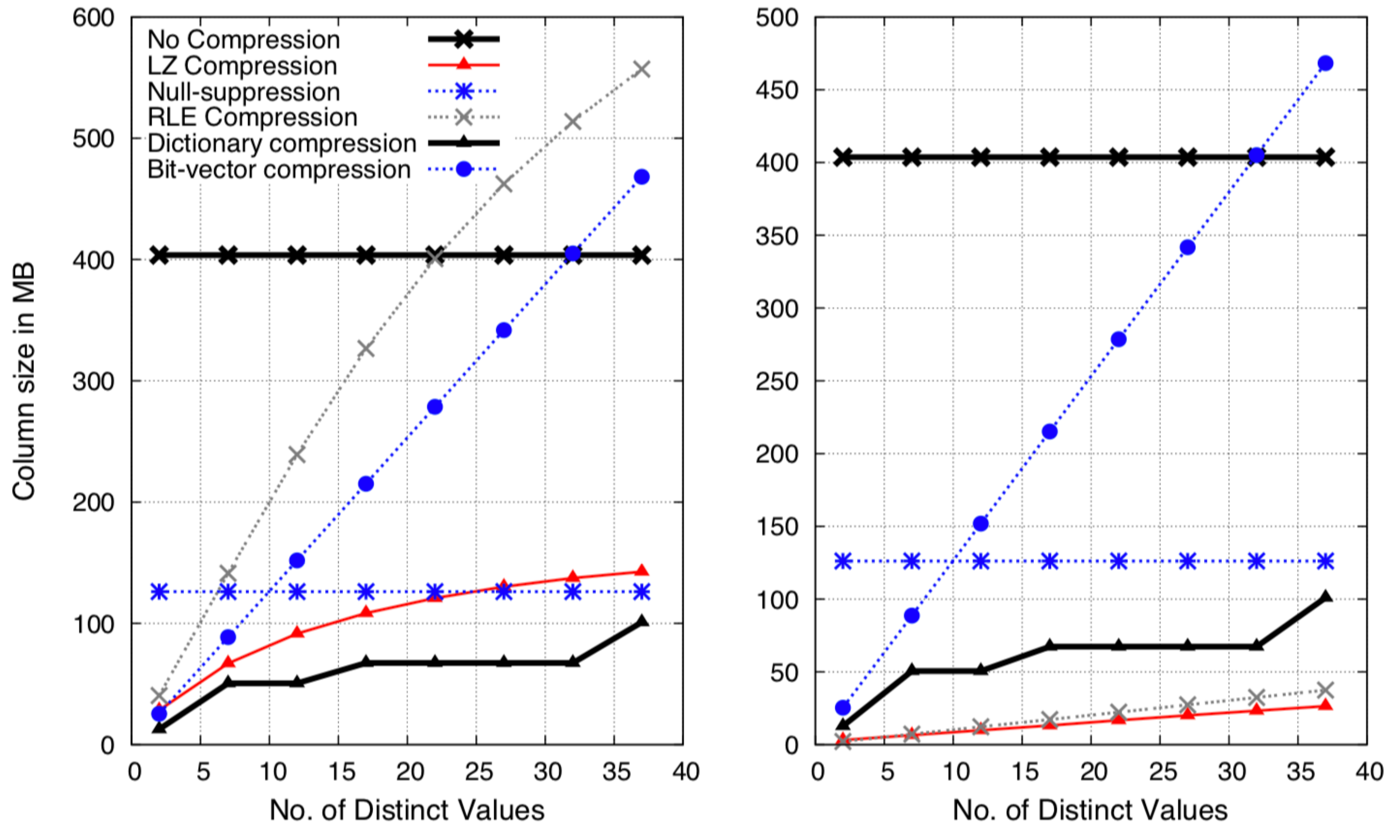
Encoding Type	Sorted?	1 value?	Pos. contig.?
RLE	yes	yes	yes
Bit-string	yes	yes	no
Null Supp.	no/yes	no	yes
Lempel-Ziv	no/yes	no	yes
Dictionary	no/yes	no	yes
Uncompressed	no/yes	no	no/yes

Using the Block API

```
COUNT(COLUMN c1)
 b = GET NEXT COMPRESSED BLOCK FROM c1
 WHILE b IS NOT NULL
   IF b.ISONEVALUE()
     x = FETCH CURRENT COUNT FOR b.GETSTARTVAL()
     x = x + b.GETSIZE()
   ELSE
     a = b.ASARRAY()
     FOR EACH ELEMENT i IN a
       x = FETCH CURRENT COUNT FOR i
       x = x + 1
   b = GET NEXT COMPRESSED BLOCK FROM c1
```

Figure 2: Pseudocode for Simple Count Aggregation

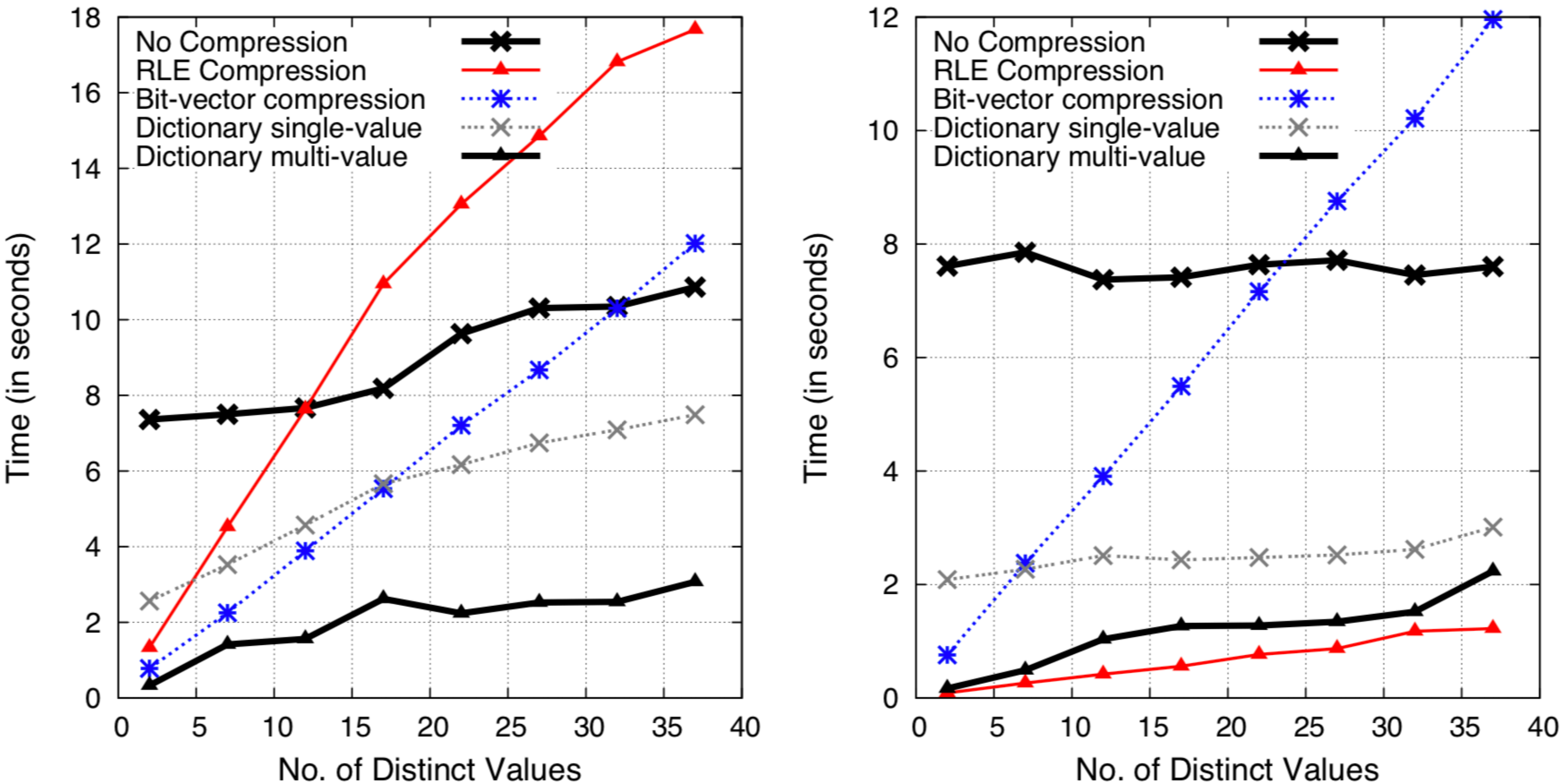
Data Size with Each Scheme



(a) Runs of length 50 (b) Runs of length 1000

Figure 4: Compressed column sizes for varied compression schemes on column with sorted runs of size 50 (a) and 1000 (b)

Performance with Each Scheme



(a) Runs of length 50

(b) Runs of length 1000

Fig … data

How would the results change on SSDs?

Outline

Co-designing storage and compute (paper)

Indexes

Key Operations on an Index

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

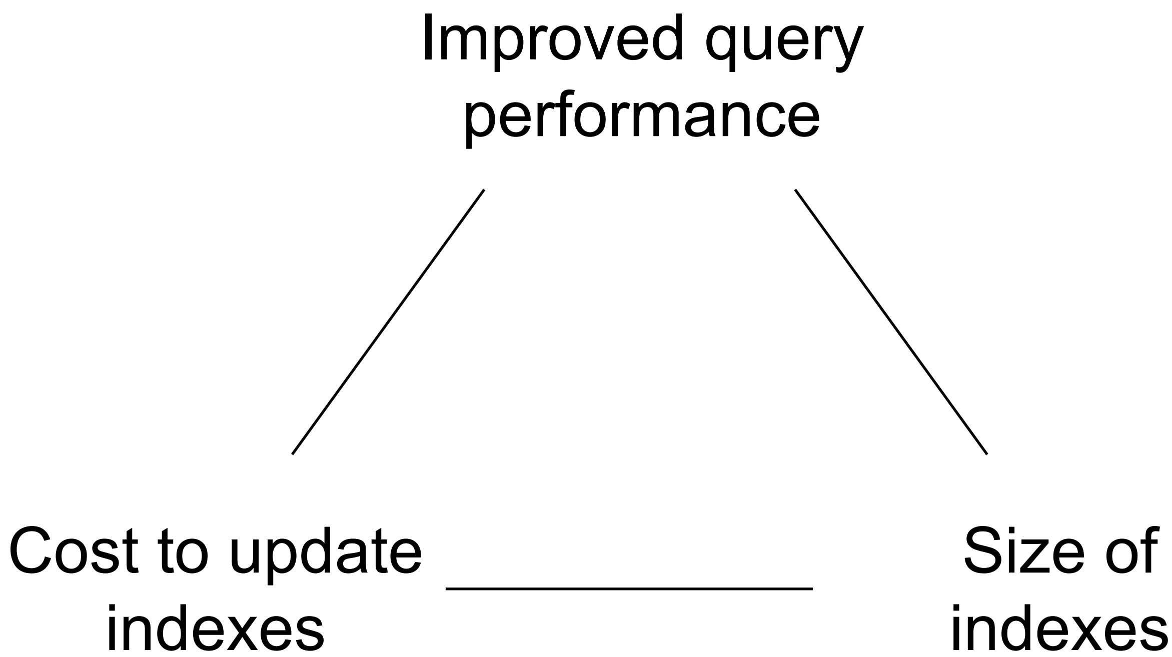
» Key can be one field or a tuple of fields (e.g. `country="US" AND state="CA"`)

» 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

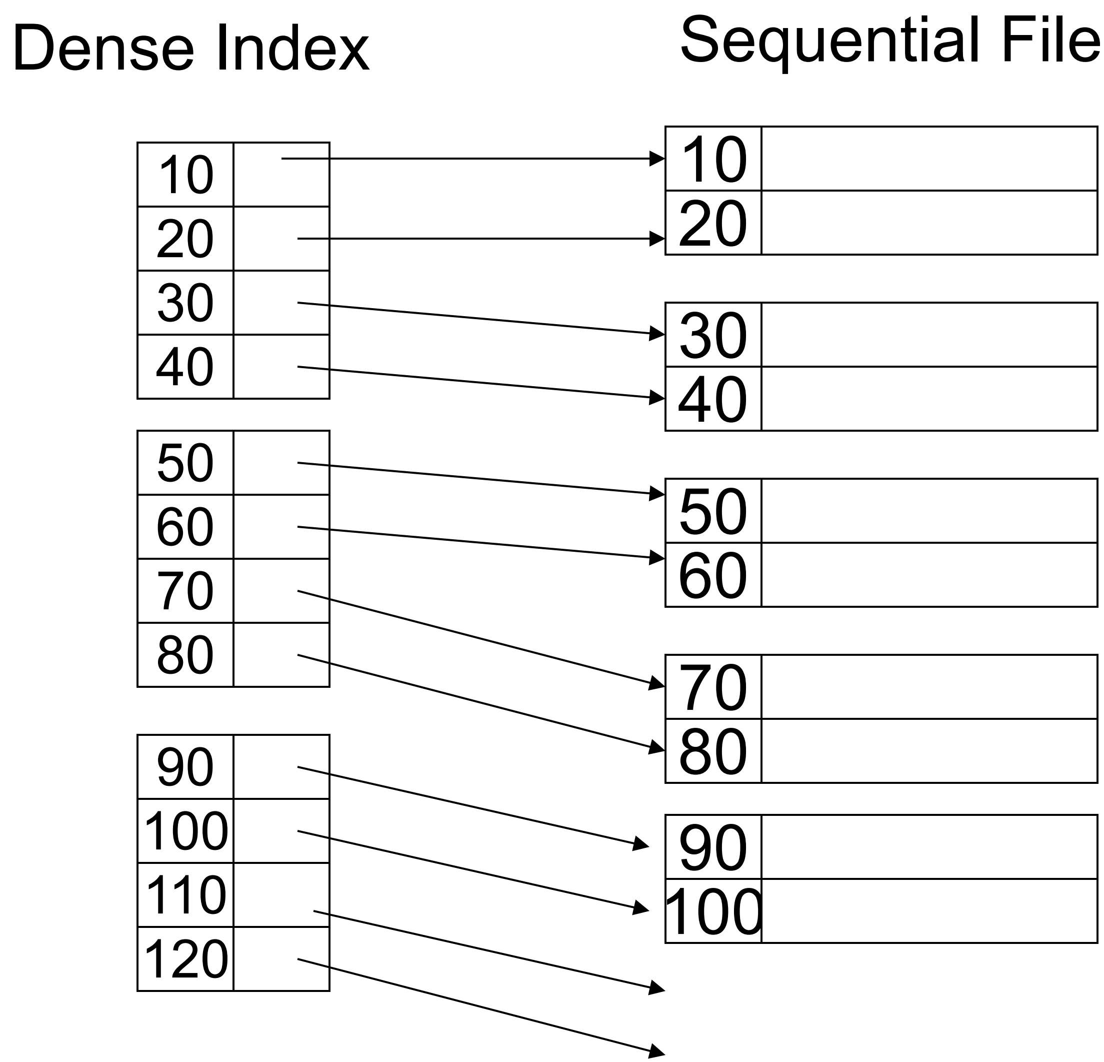
10	
20	

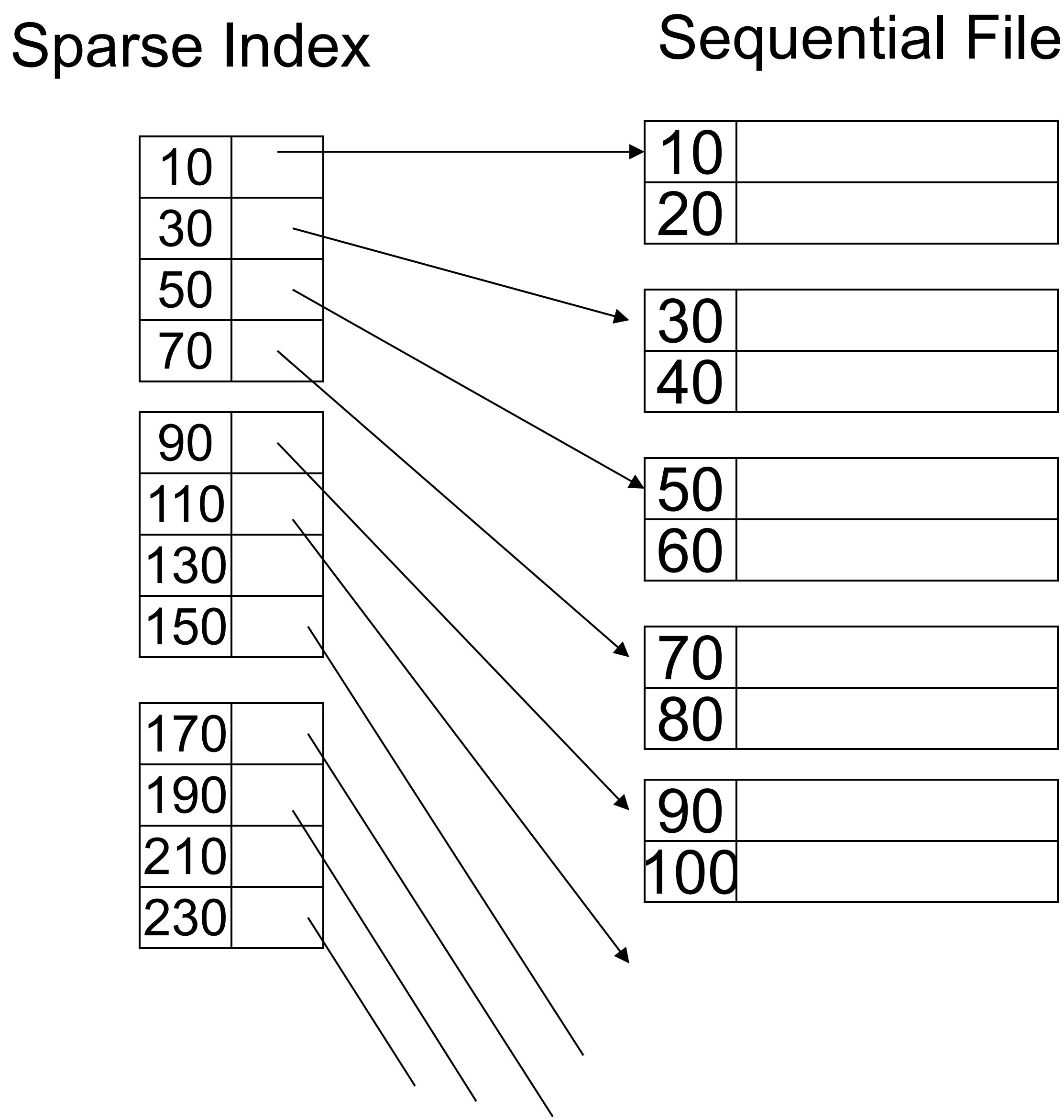
30	
40	

50	
60	

70	
80	

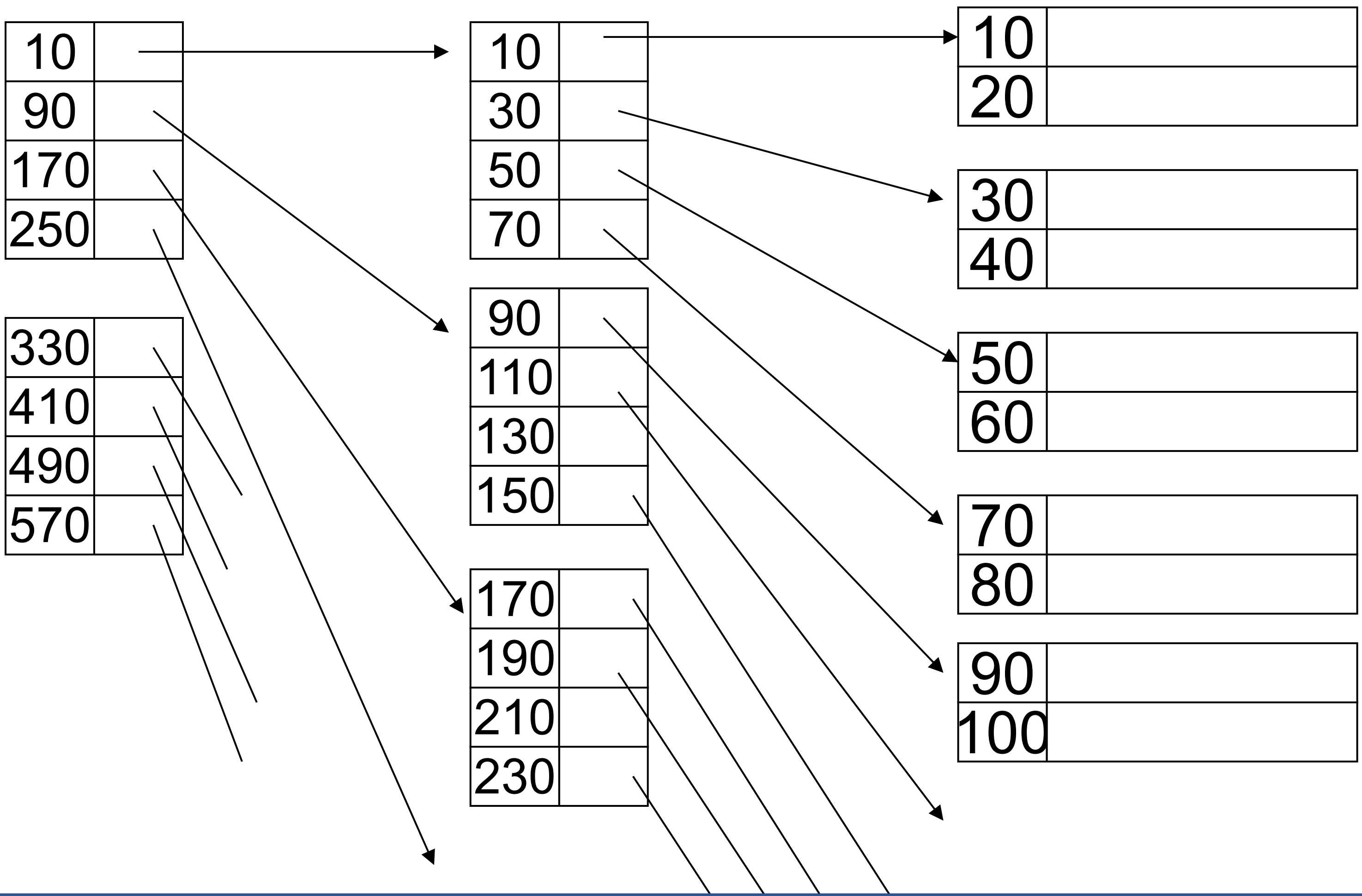
90	
100	


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

2-level sparse index

Sequential File



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 any record exists 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

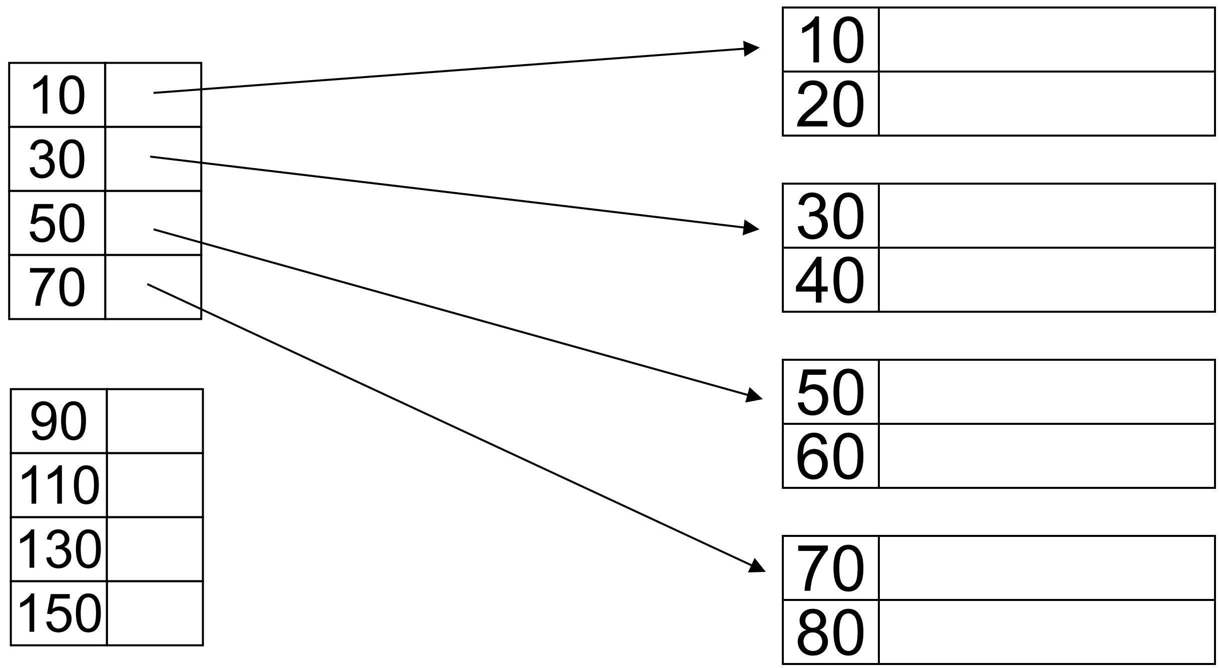
Multi-level index

Handling Duplicate Keys

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

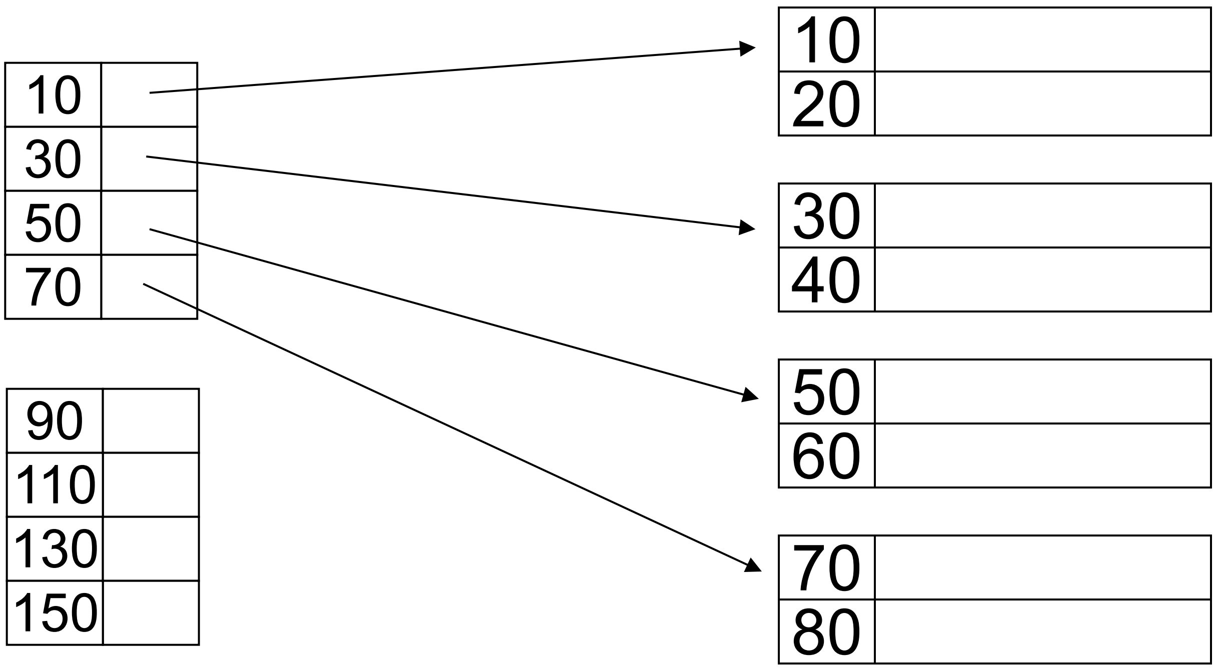
For a secondary index, need to point to a list of records since they can be anywhere

Deletion: Sparse Index



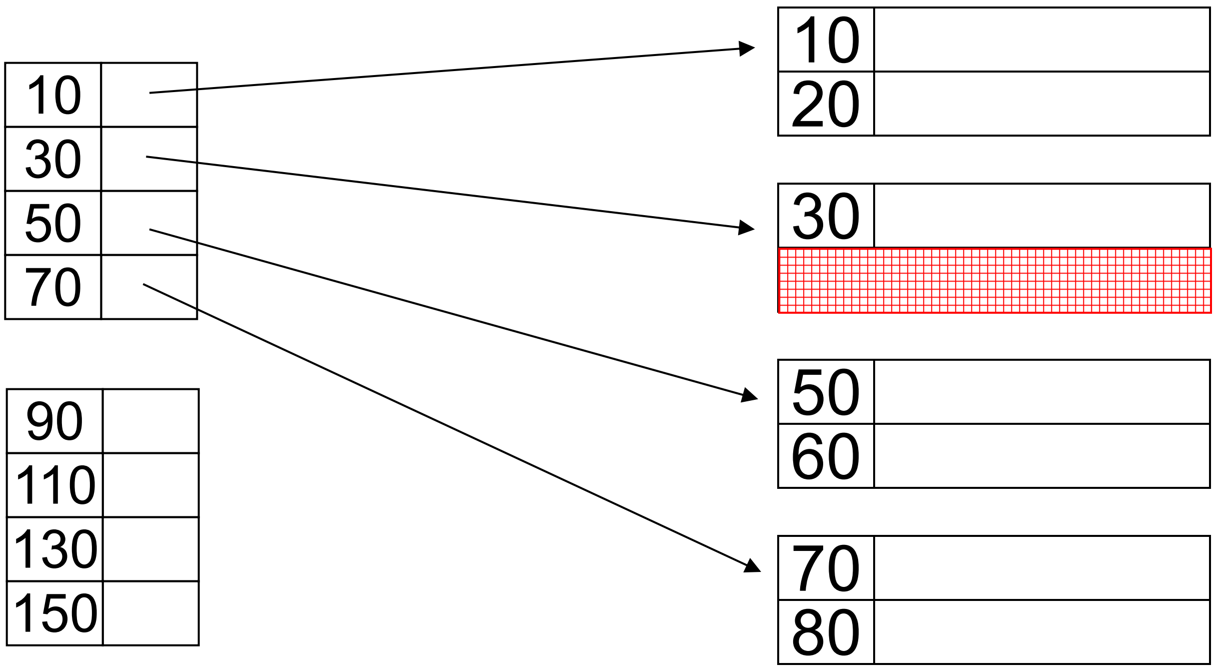
Deletion: Sparse Index

– delete record 40



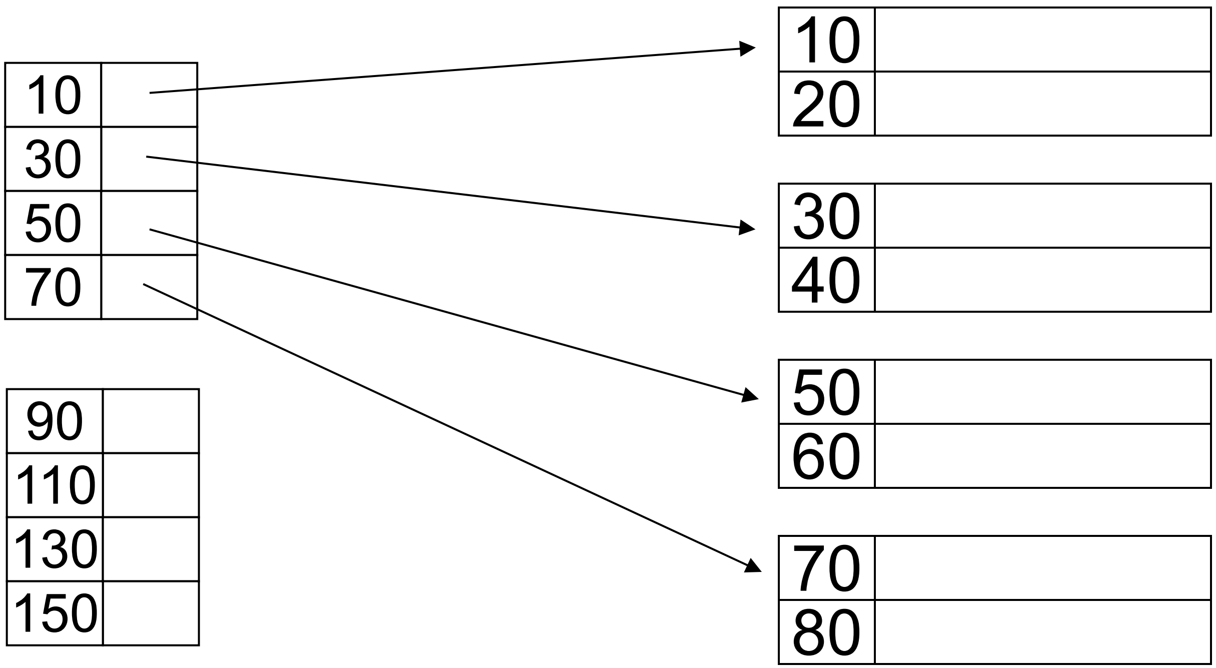
Deletion: Sparse Index

– delete record 40



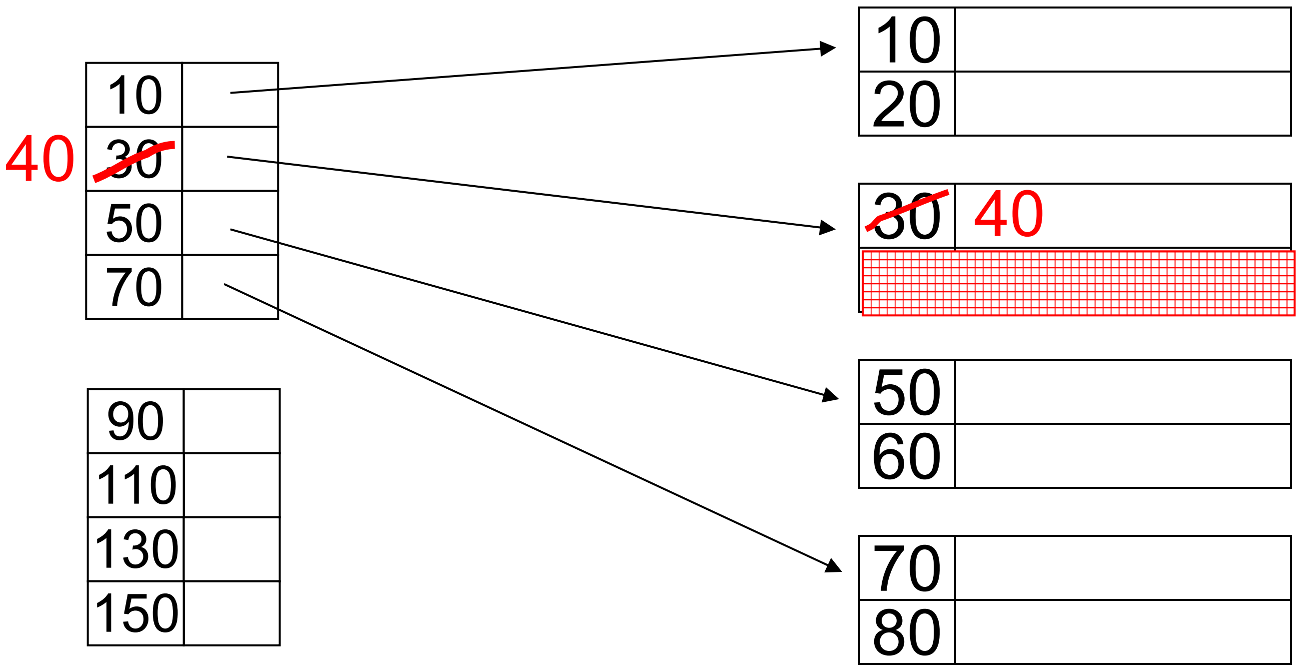
Deletion: Sparse Index

– delete record 30



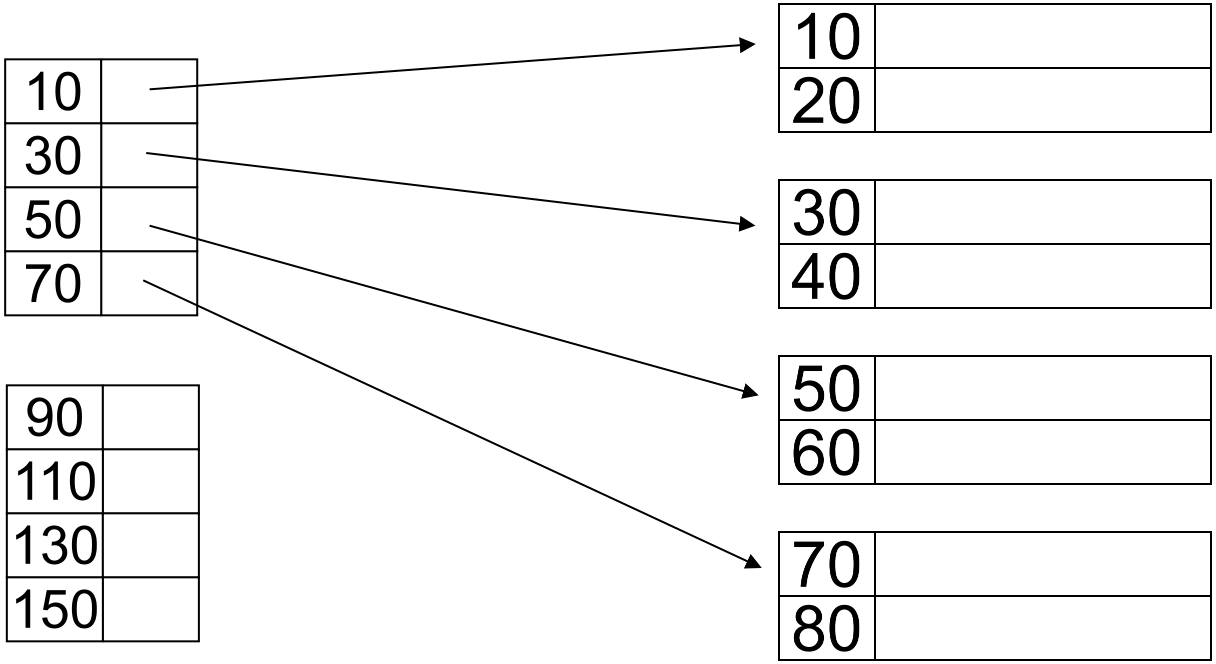
Deletion: Sparse Index

– delete record 30



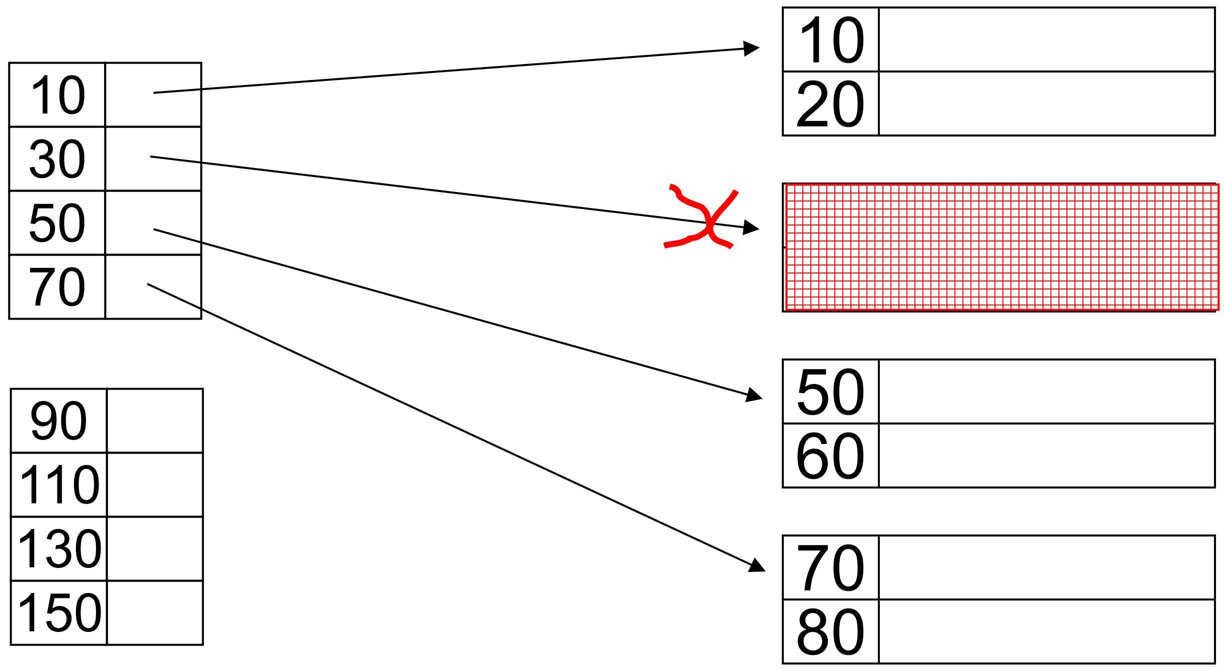
Deletion: Sparse Index

– delete records 30 & 40



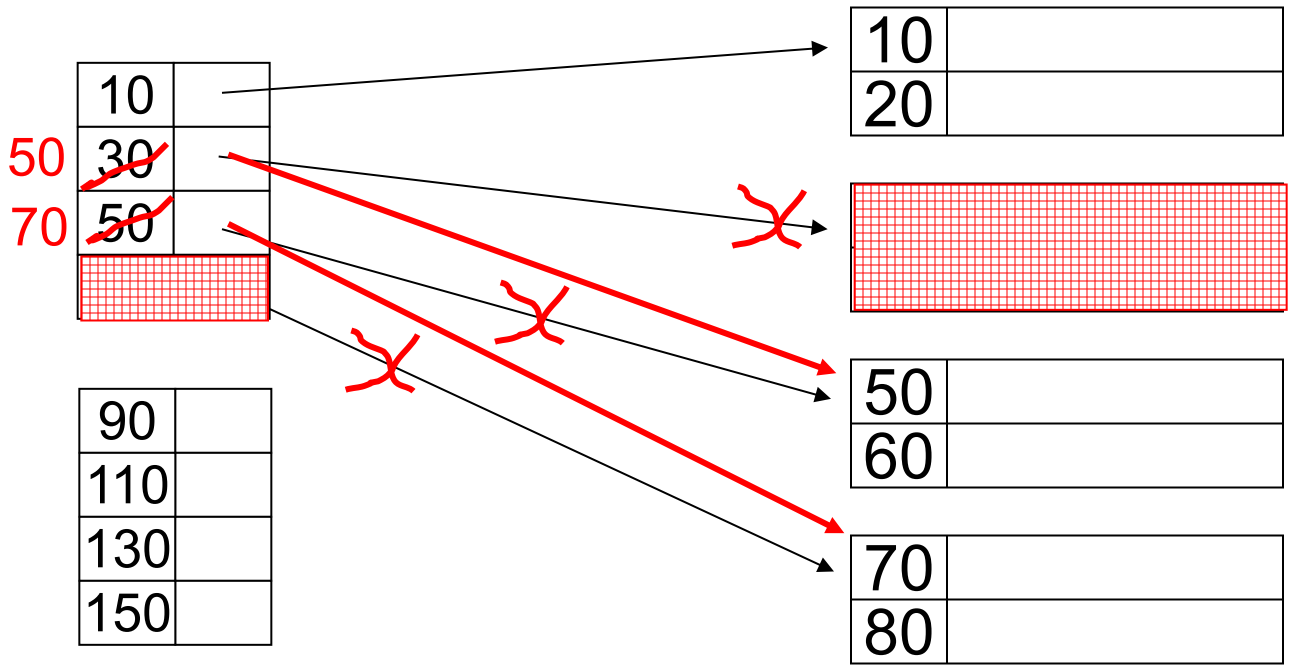
Deletion: Sparse Index

– delete records 30 & 40

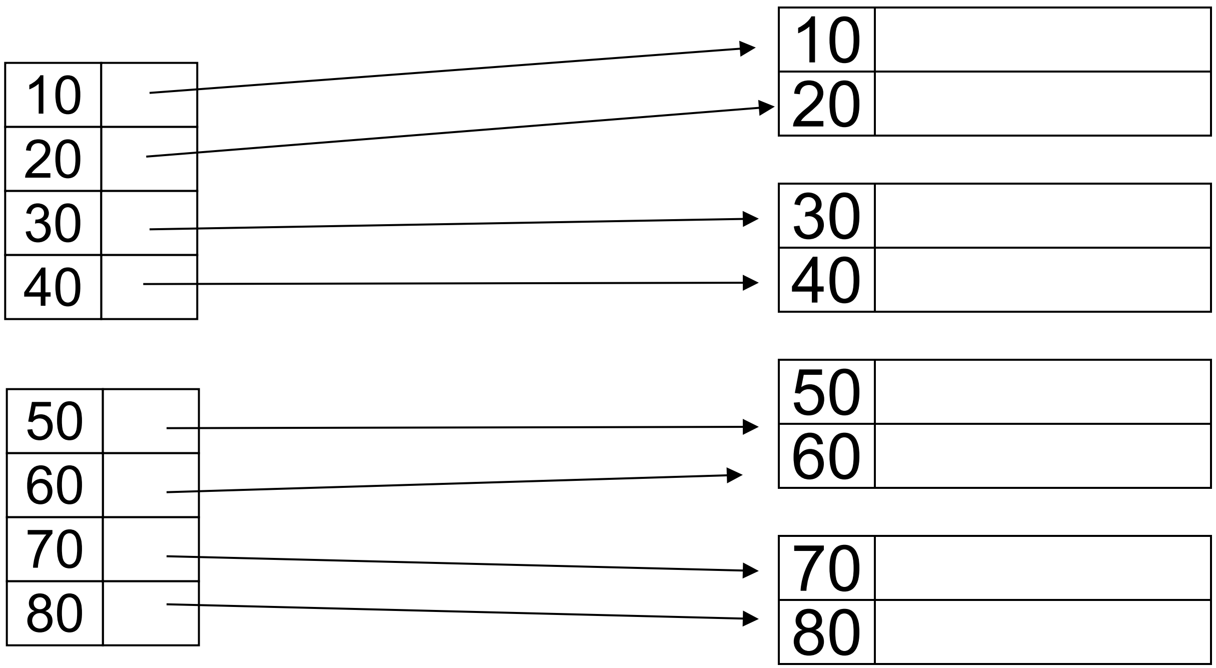


Deletion: Sparse Index

– delete records 30 & 40

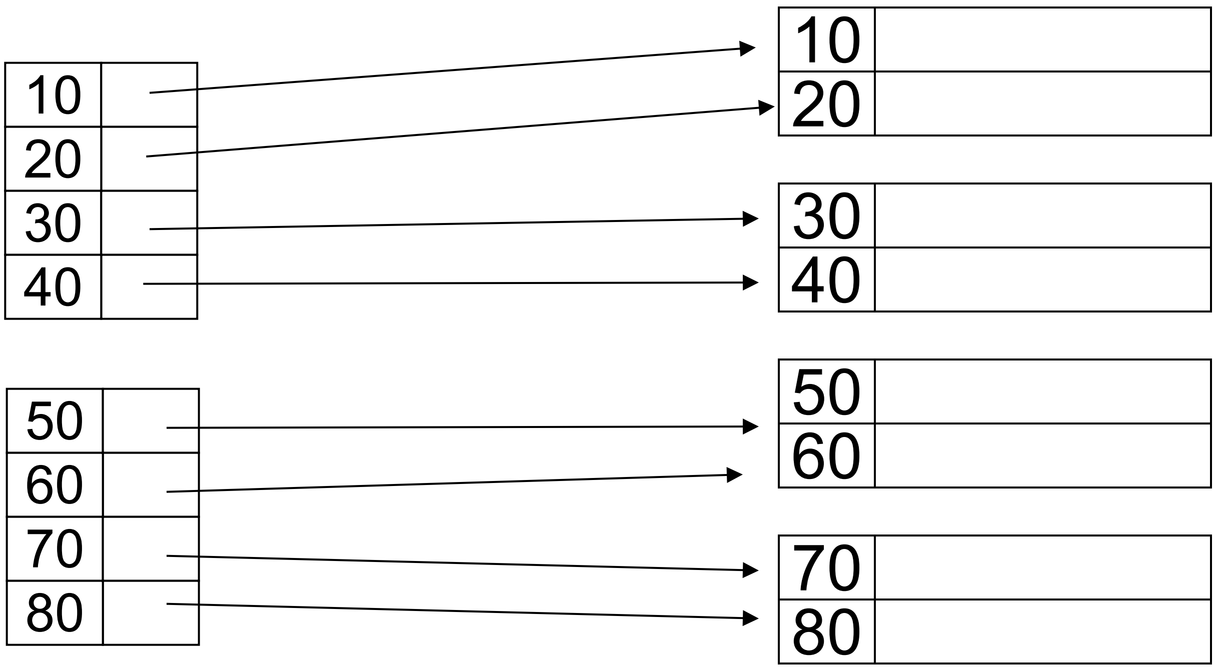


Deletion: Dense Index



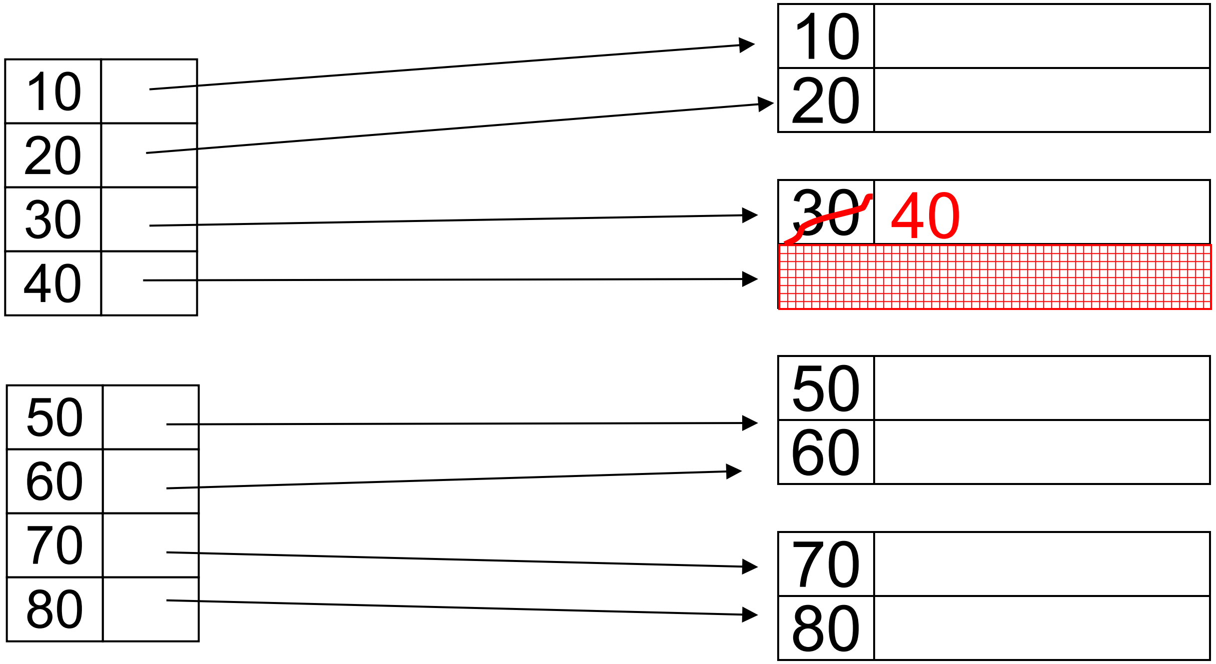
Deletion: Dense Index

– delete record 30



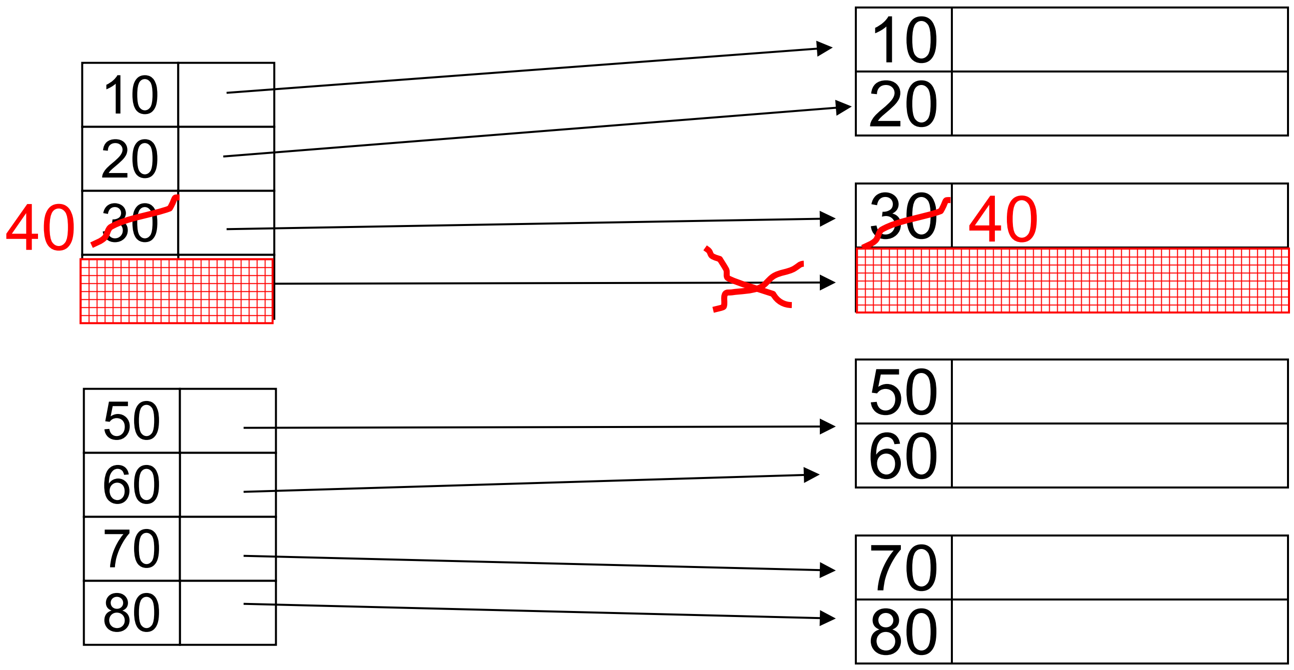
Deletion: Dense Index

– delete record 30



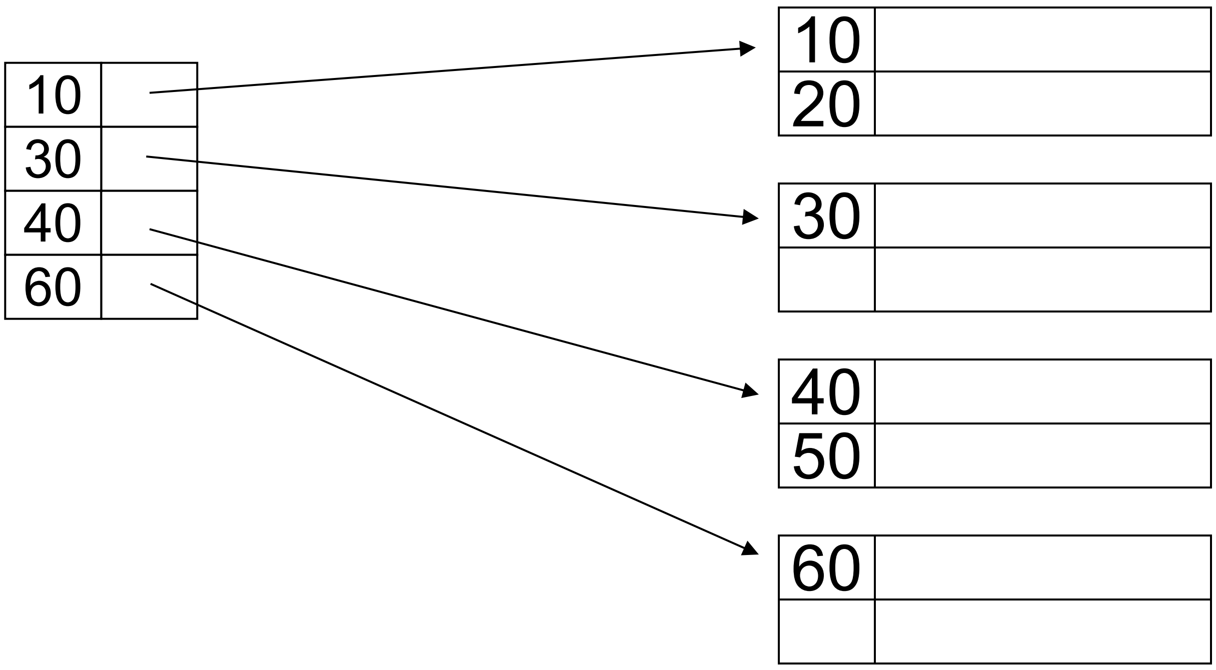
Deletion: Dense Index

– delete record 30



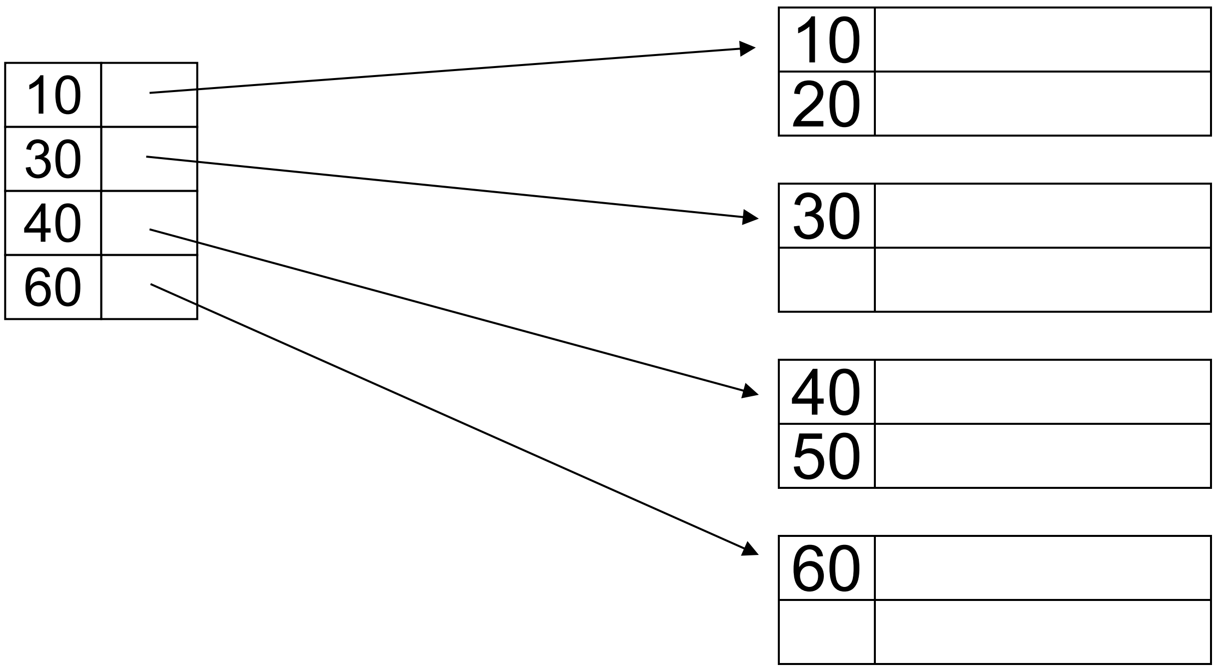
Insertion: Sparse Index

– insert record 34



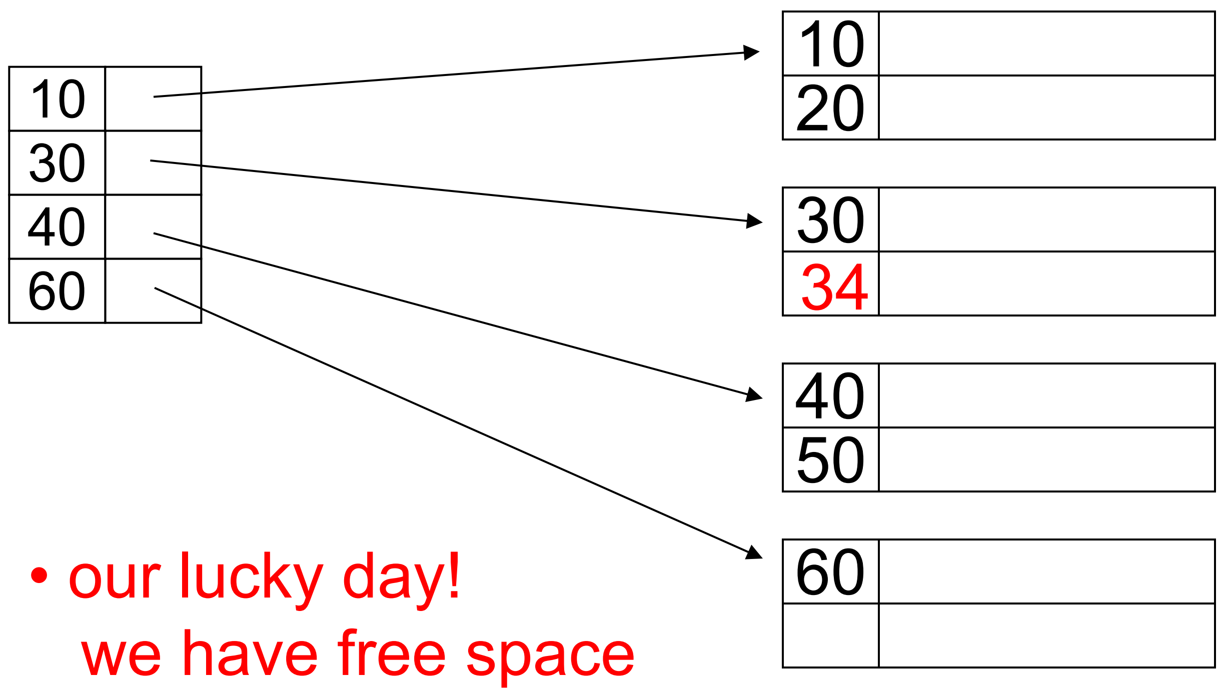
Insertion: Sparse Index

– insert record 34



Insertion: Sparse Index

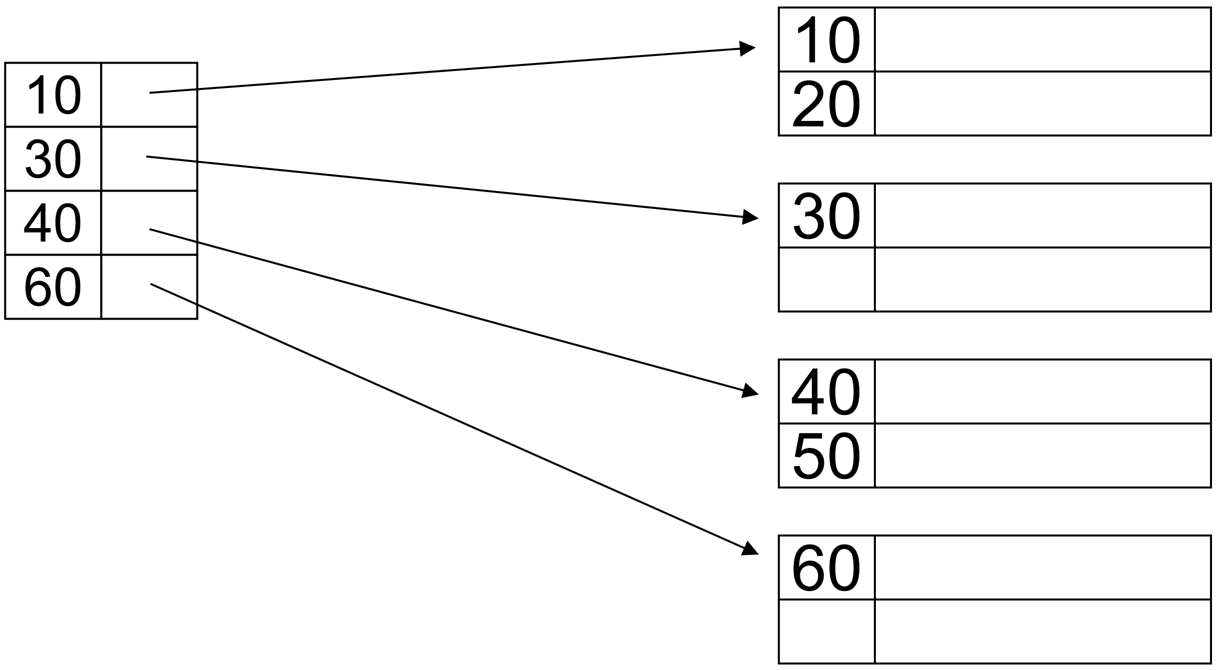
– insert record 34



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

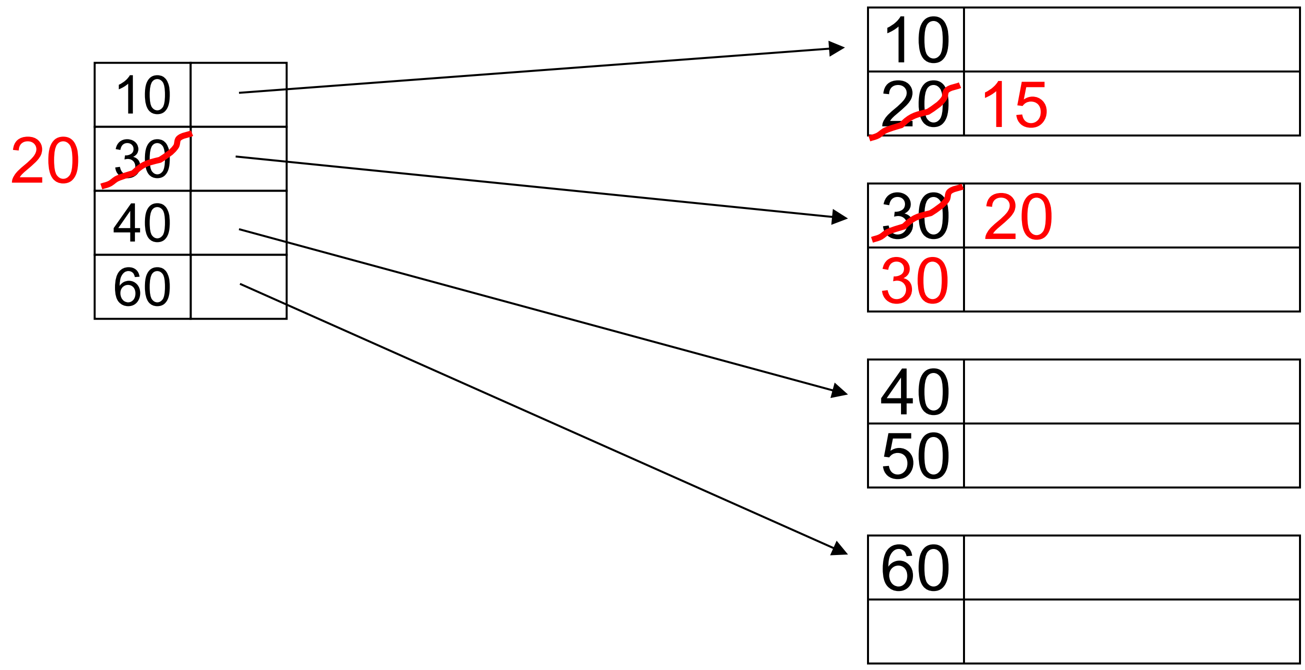
Insertion: Sparse Index

– insert record 15



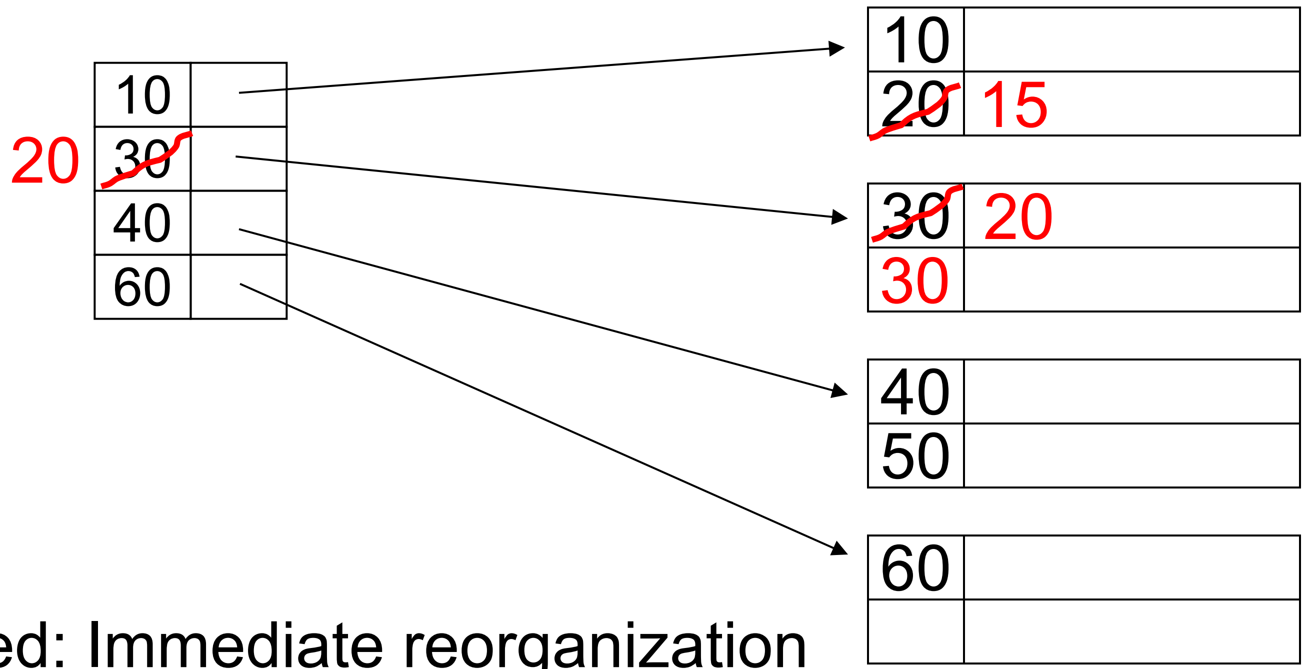
Insertion: Sparse Index

– insert record 15



Insertion: Sparse Index

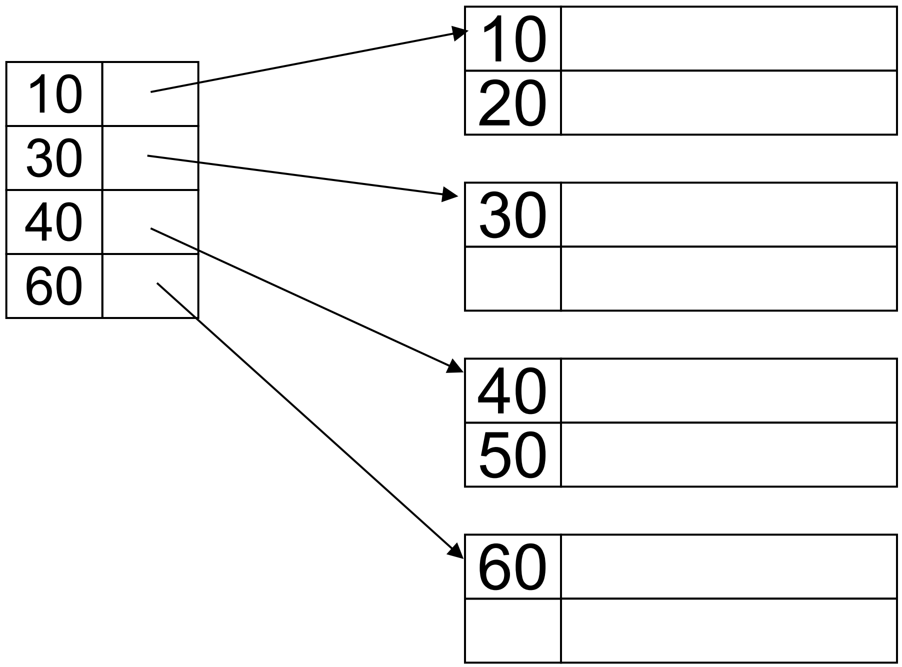
– insert record 15



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

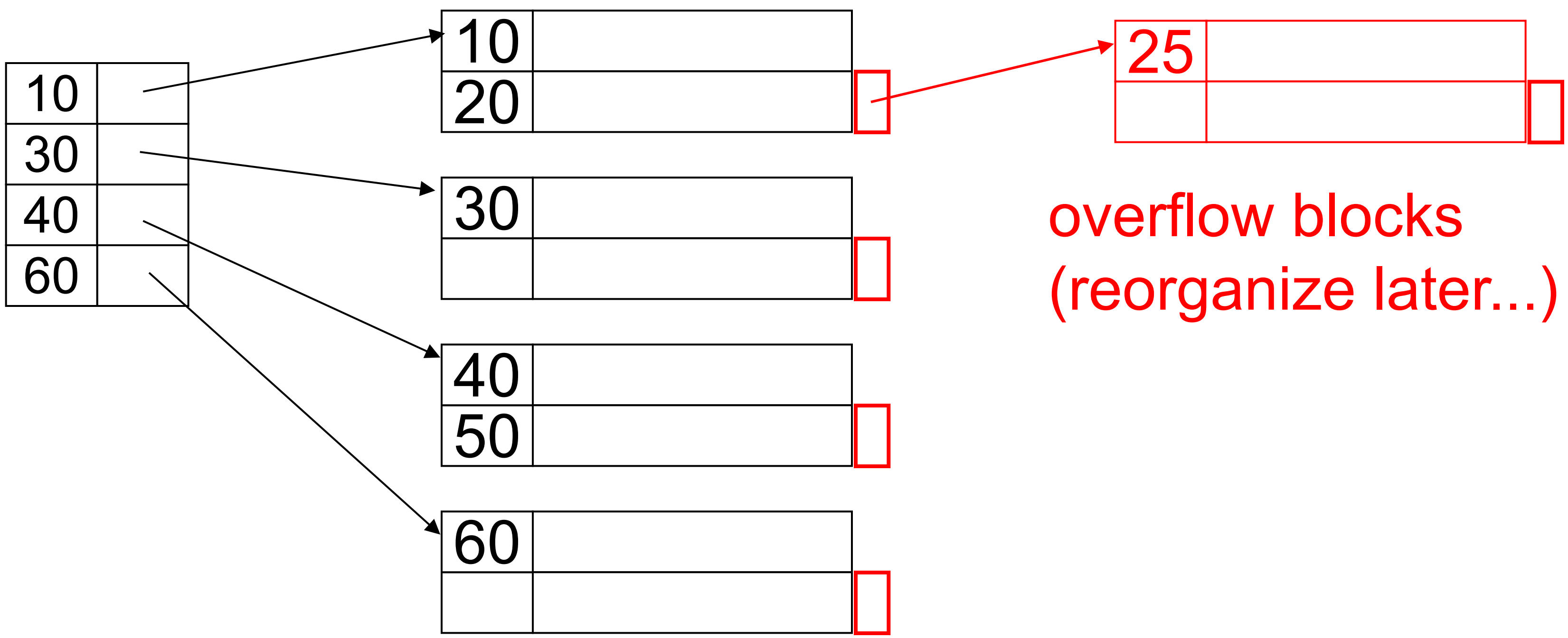
Insertion: Sparse Index

– insert record 25

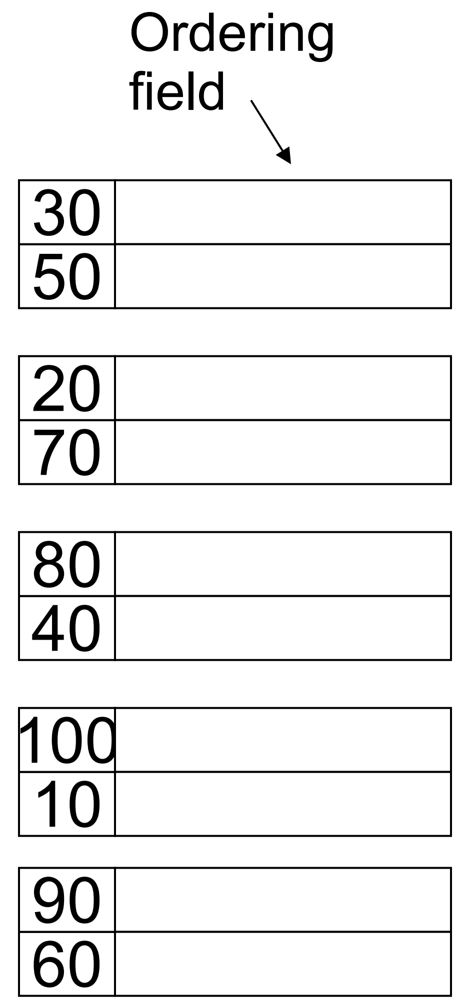


Insertion: Sparse Index

– insert record 25

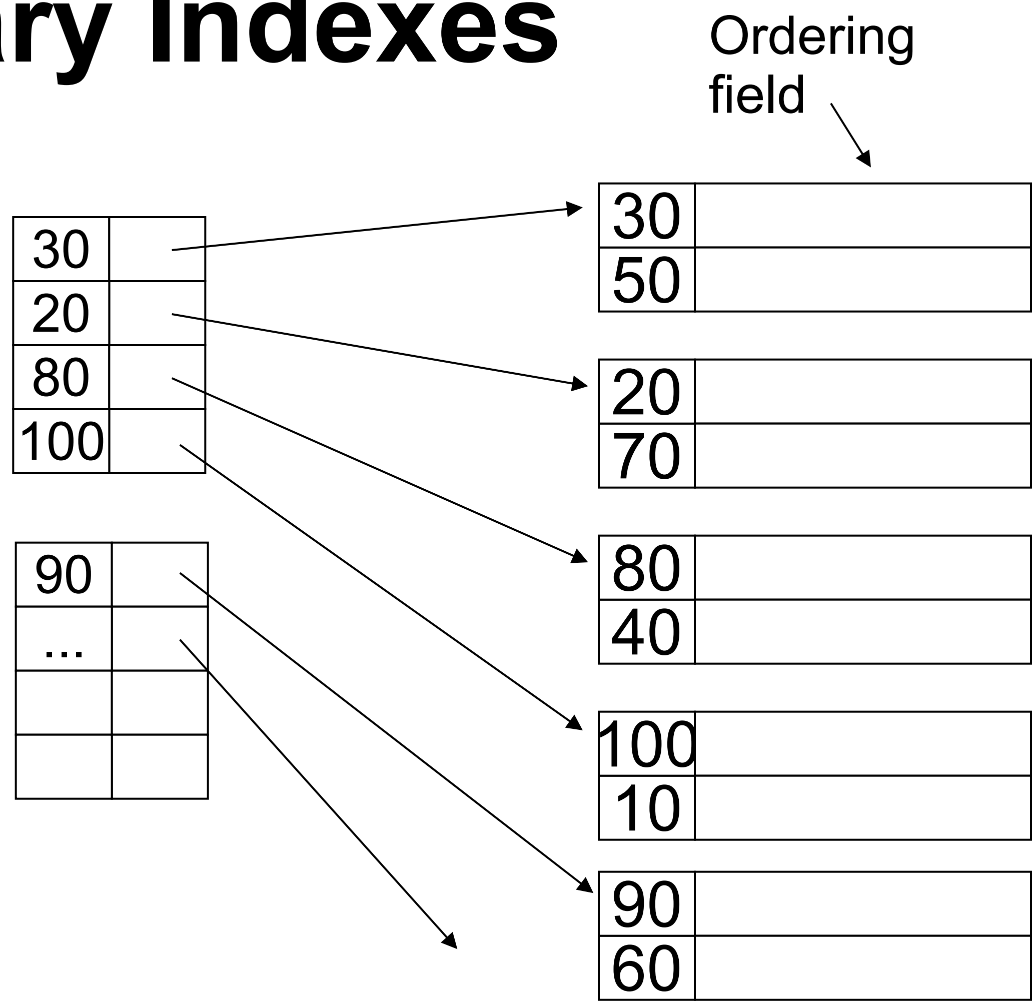


Secondary Indexes

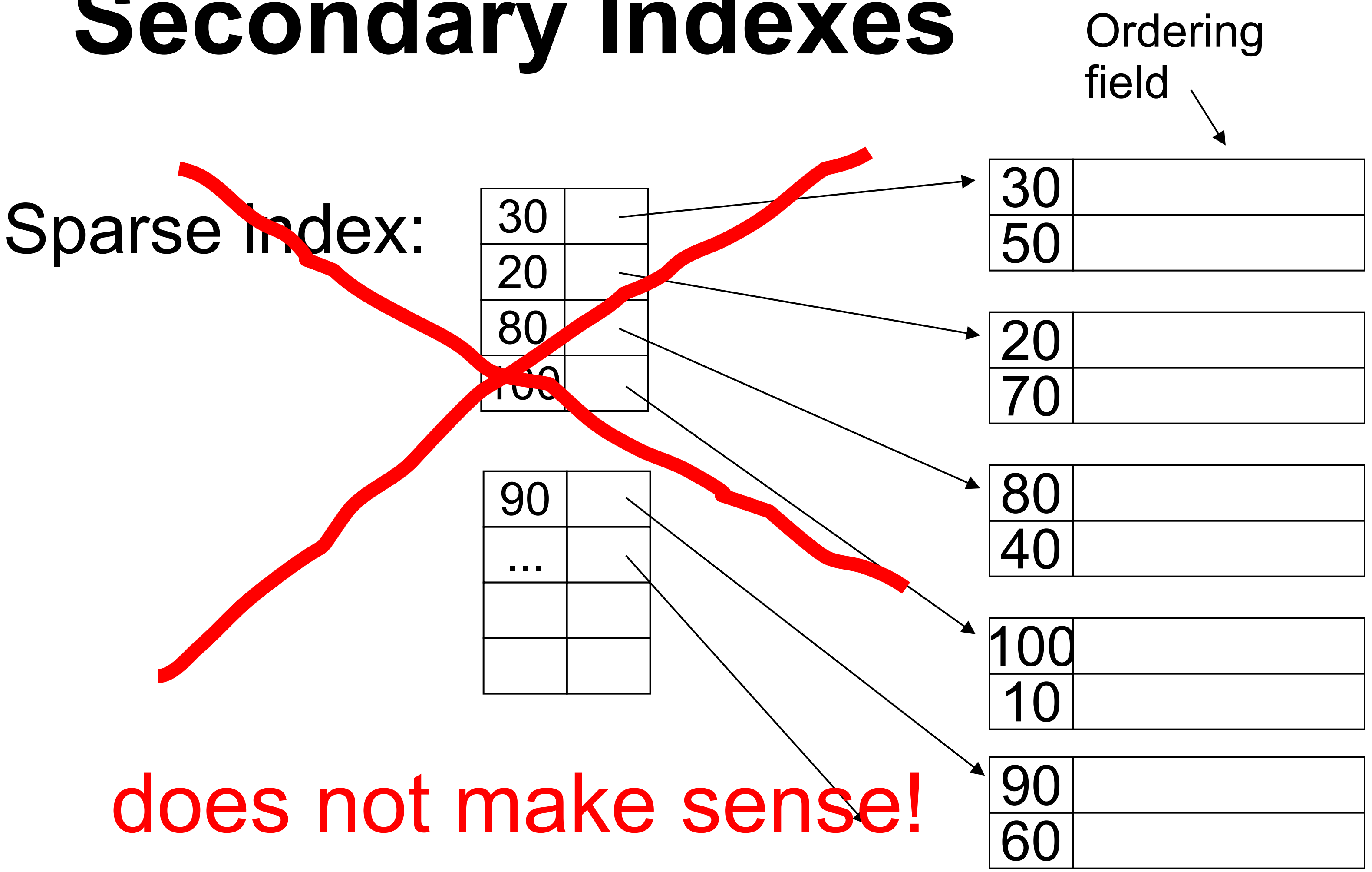


Secondary Indexes

Sparse index:



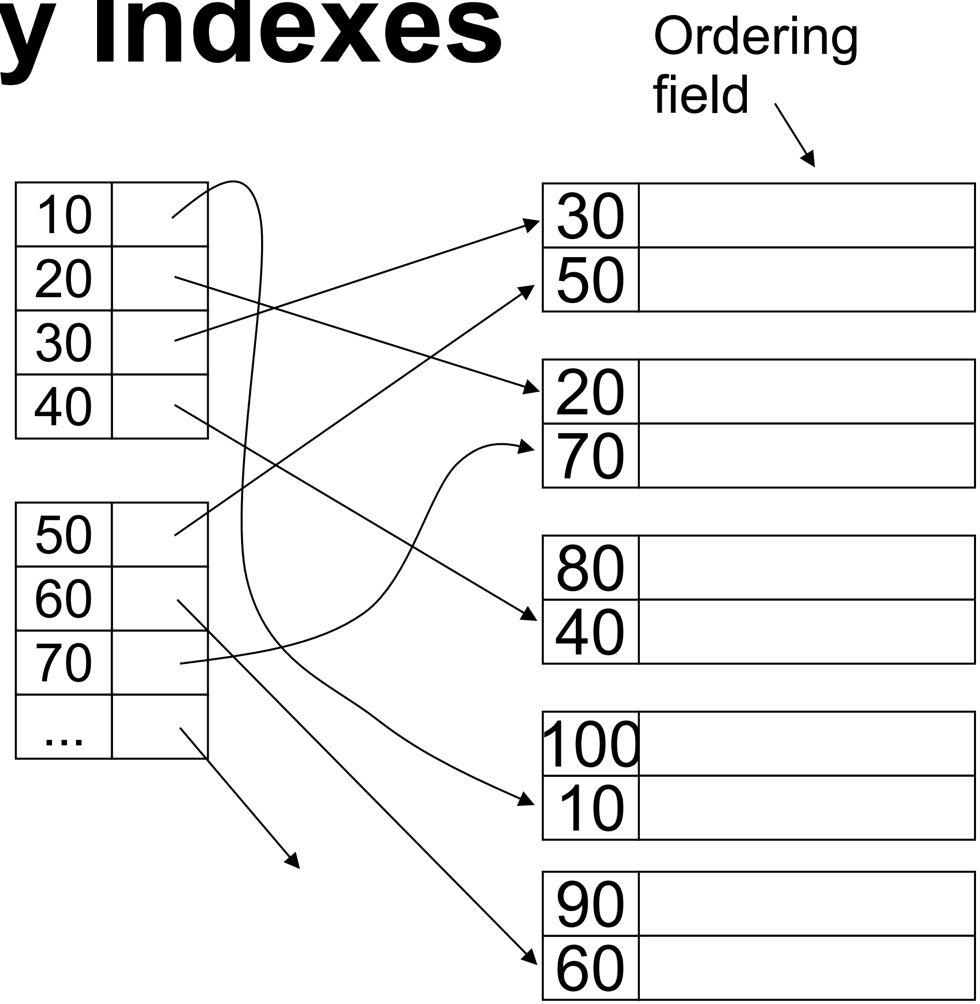
Secondary Indexes



does not make sense!

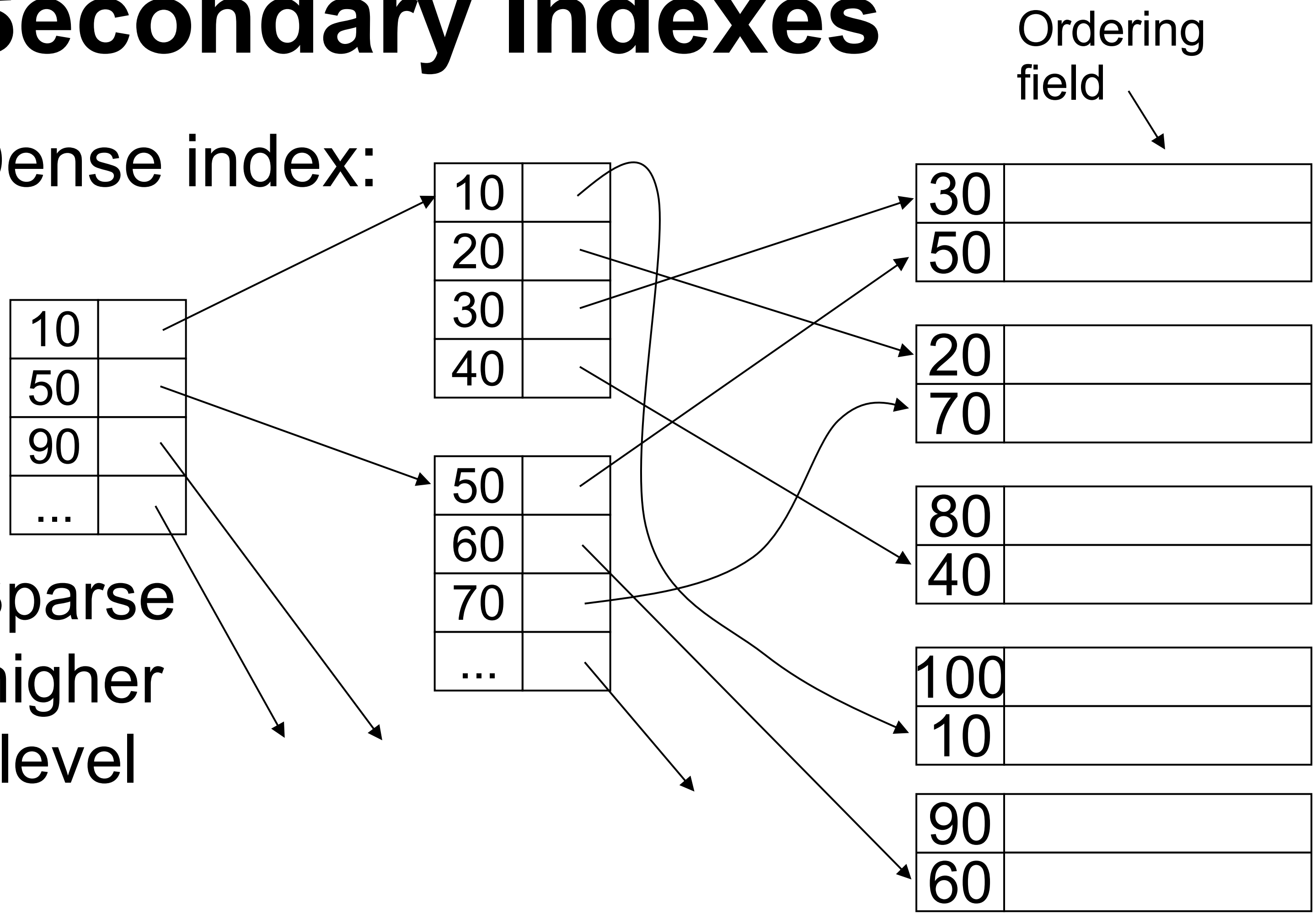
Secondary Indexes

Dense index:



Secondary Indexes

Dense index:



Sparse higher level

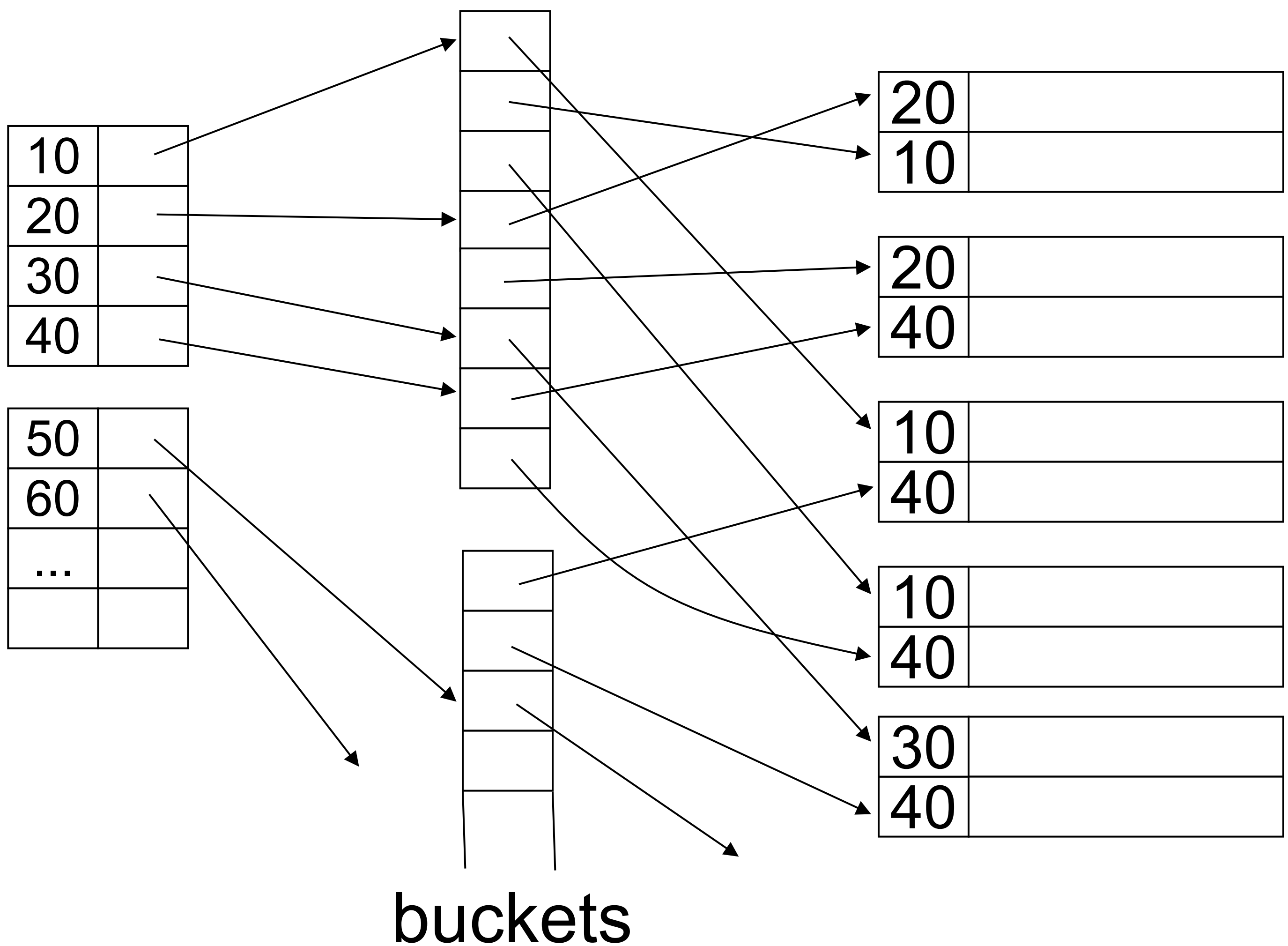
With Secondary Indexes

Lowest level is dense

Other levels are sparse

Pointers are record pointers (not block)

Duplicate Values in Secondary Indexes



Another Benefit of Buckets

Can compute complex queries through Boolean operations on record pointer lists

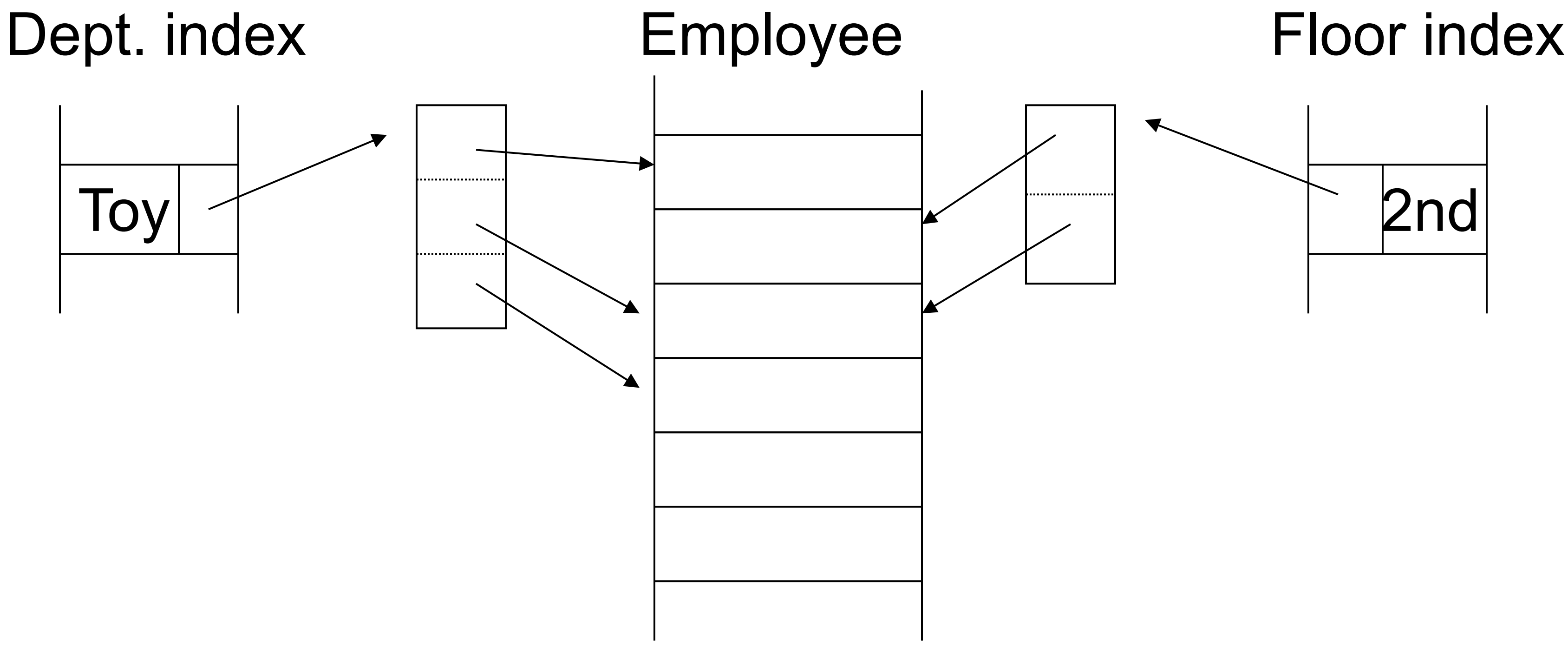
Consider an employee table with foreign keys for department and floor:

DeptID	…
1	
2	

EmpID	Name	DeptID	FloorID
1	Alice	2	1
2	Bob	2	2

FloorID	…
1	
2	

Query: Get Employees in (Toy Dept) AND (2^{nd} floor)



Intersect "Toy" bucket and "2^{nd} floor" buckets to get list of matching employees

This Idea is Used in Text Information Retrieval

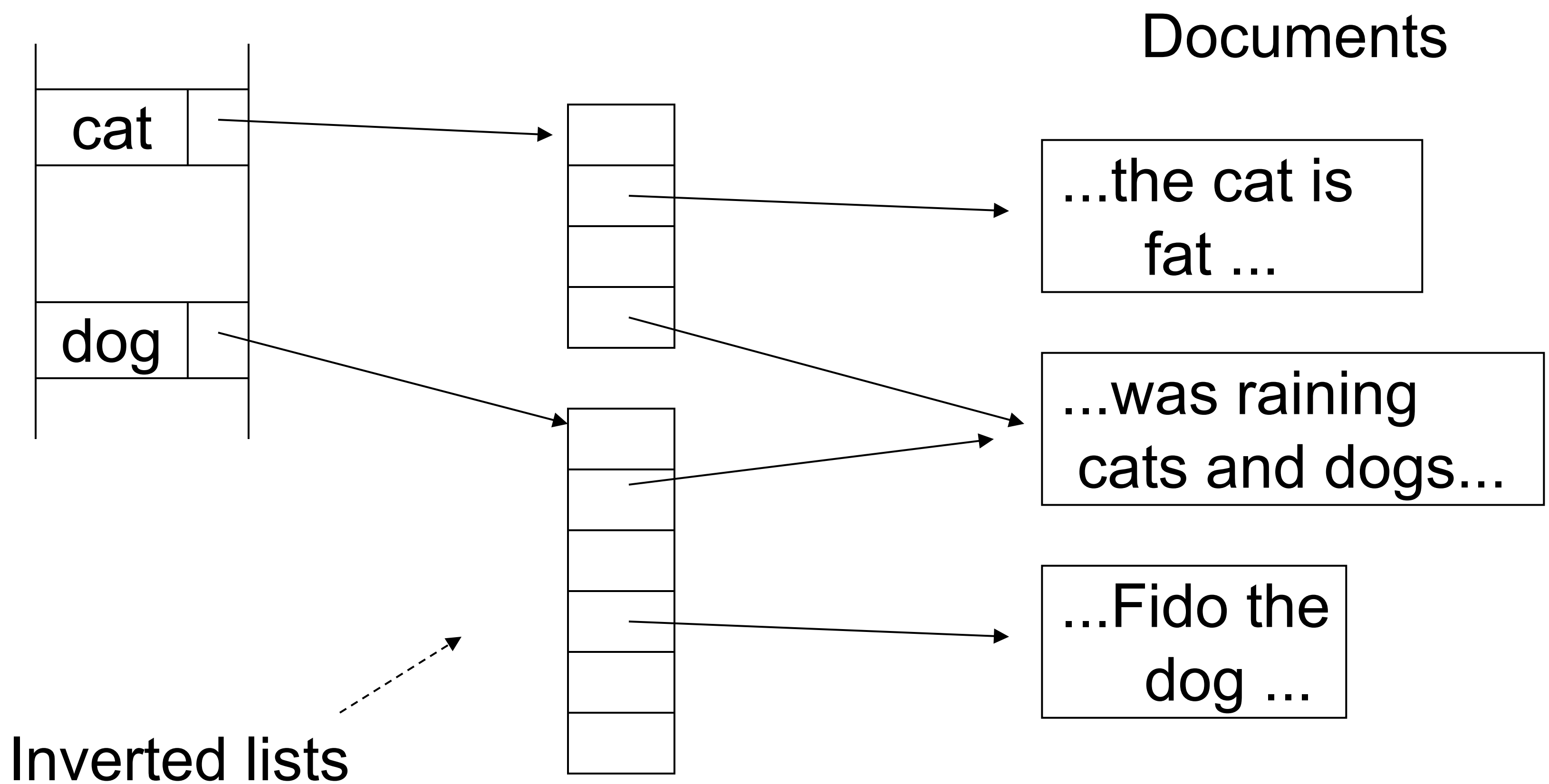
Documents

...the cat is fat ...

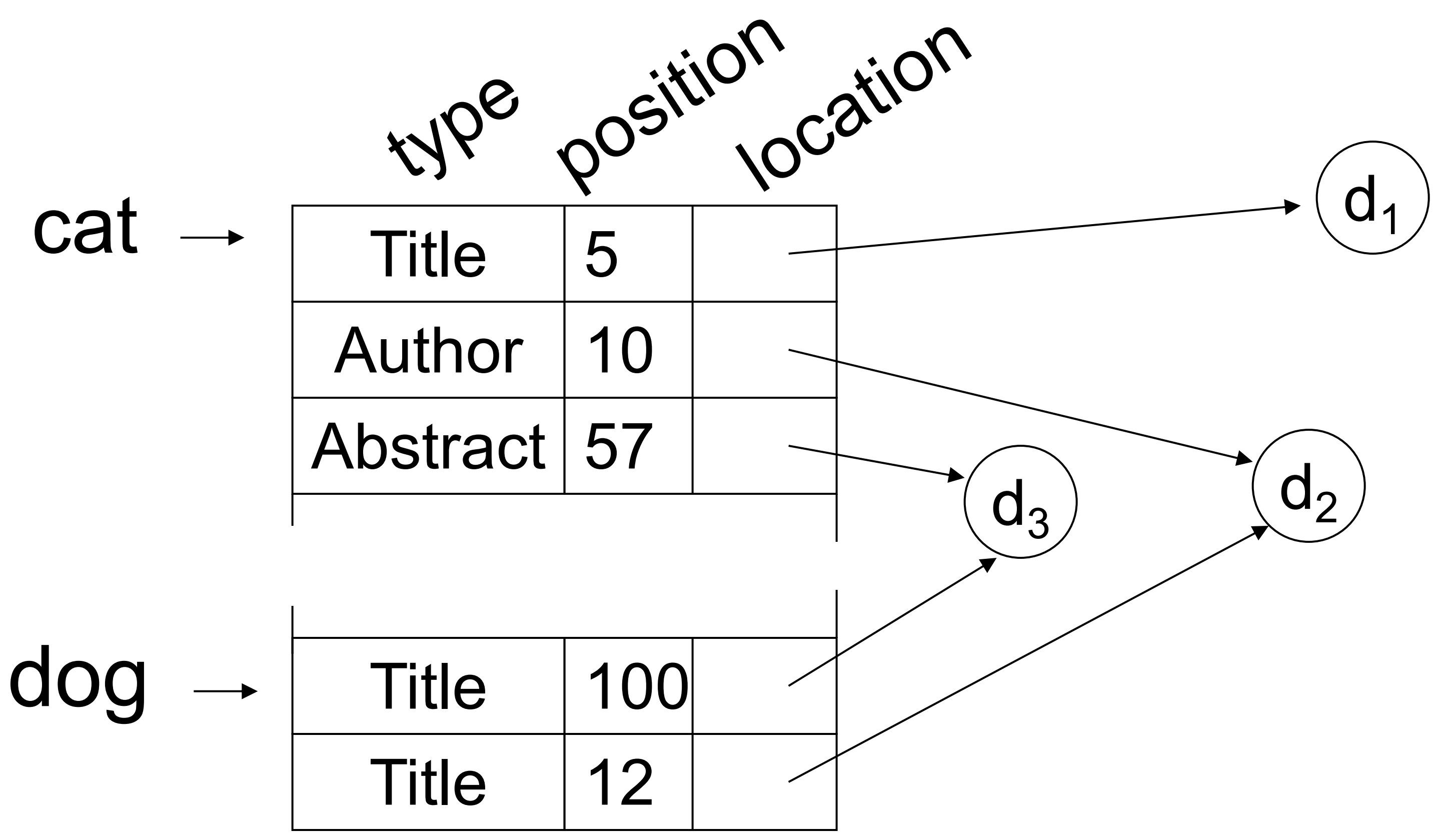
...was raining cats and dogs...

...Fido the dog ...

This Idea is Used in Text Information Retrieval



Common Technique: More Info in Index Entries



Answer queries like "cat within 5 words of dog"

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

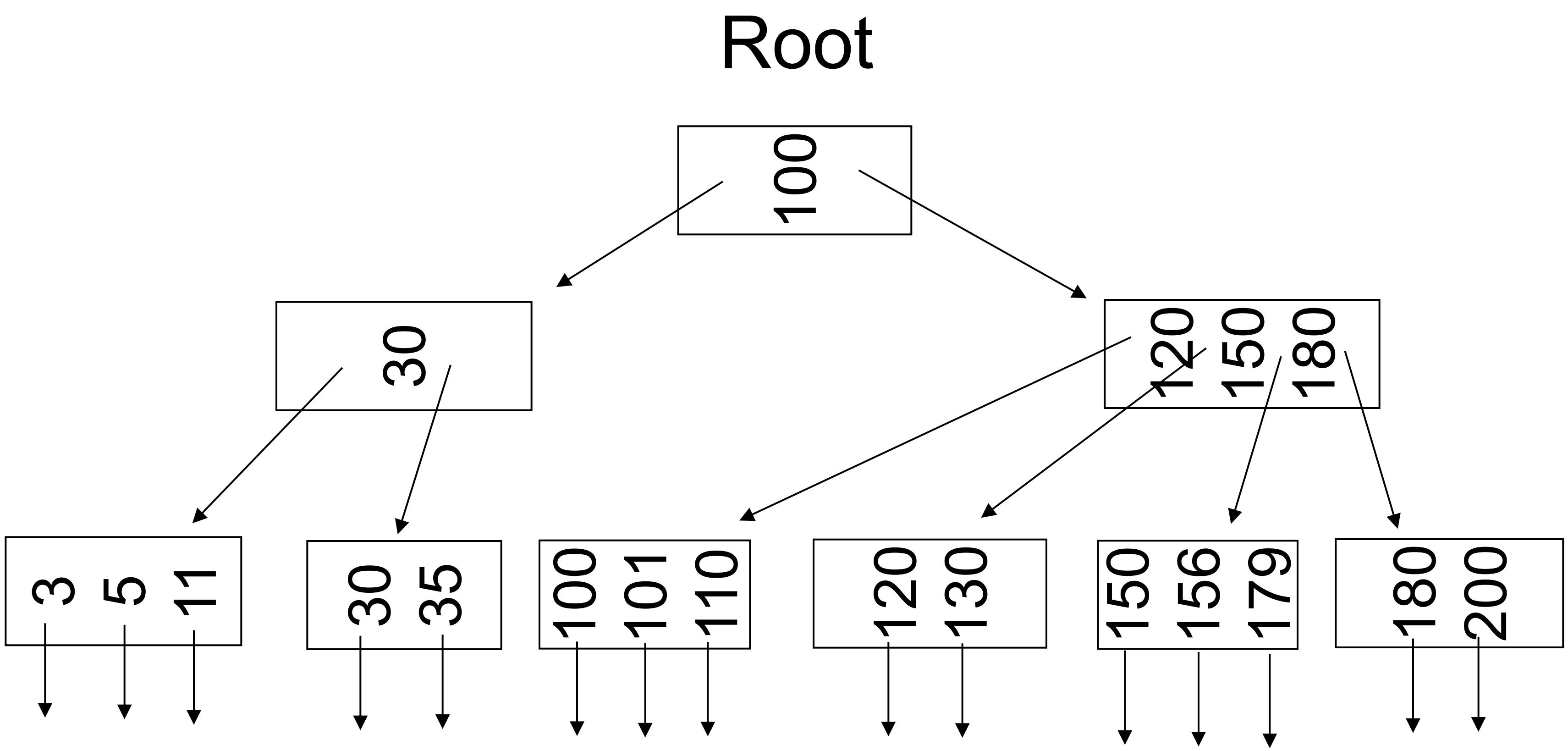
Another type of index

» Give up on sequentiality of index

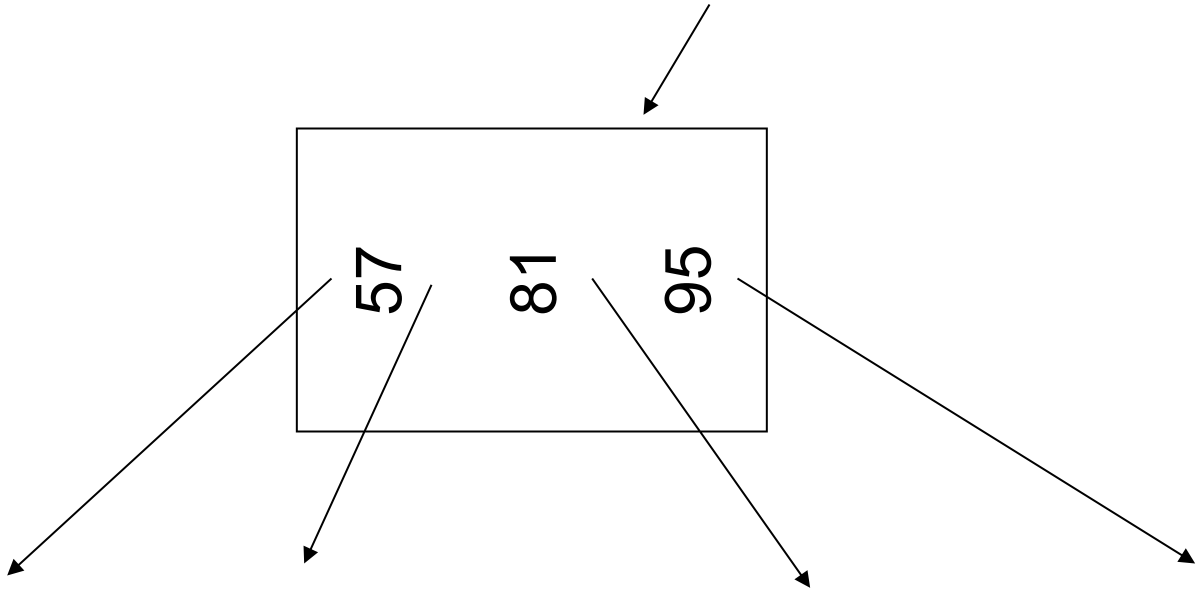
» Try to get “balance”

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)

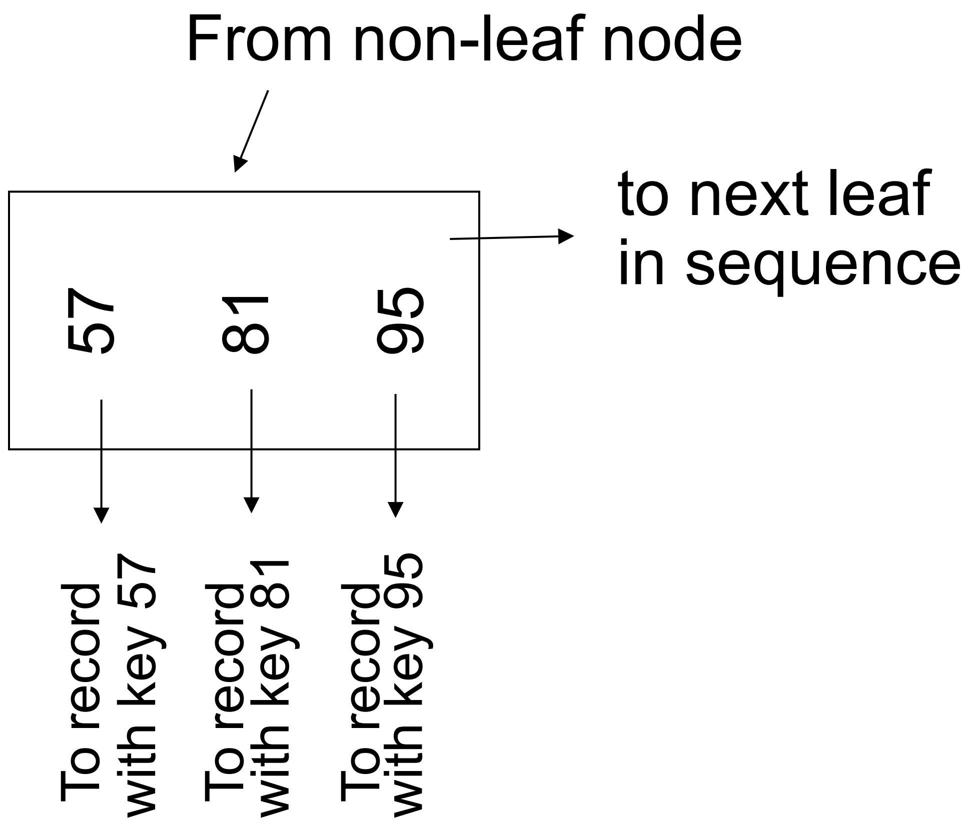


Sample Non-Leaf



to keys	to keys	to keys	to keys
< 57	$57 \leq k < 81$	$81 \leq k < 95$	≥ 95

Sample Leaf Node



Size of Nodes on Disk

n + 1 pointers
n keys

(Fixed size nodes)

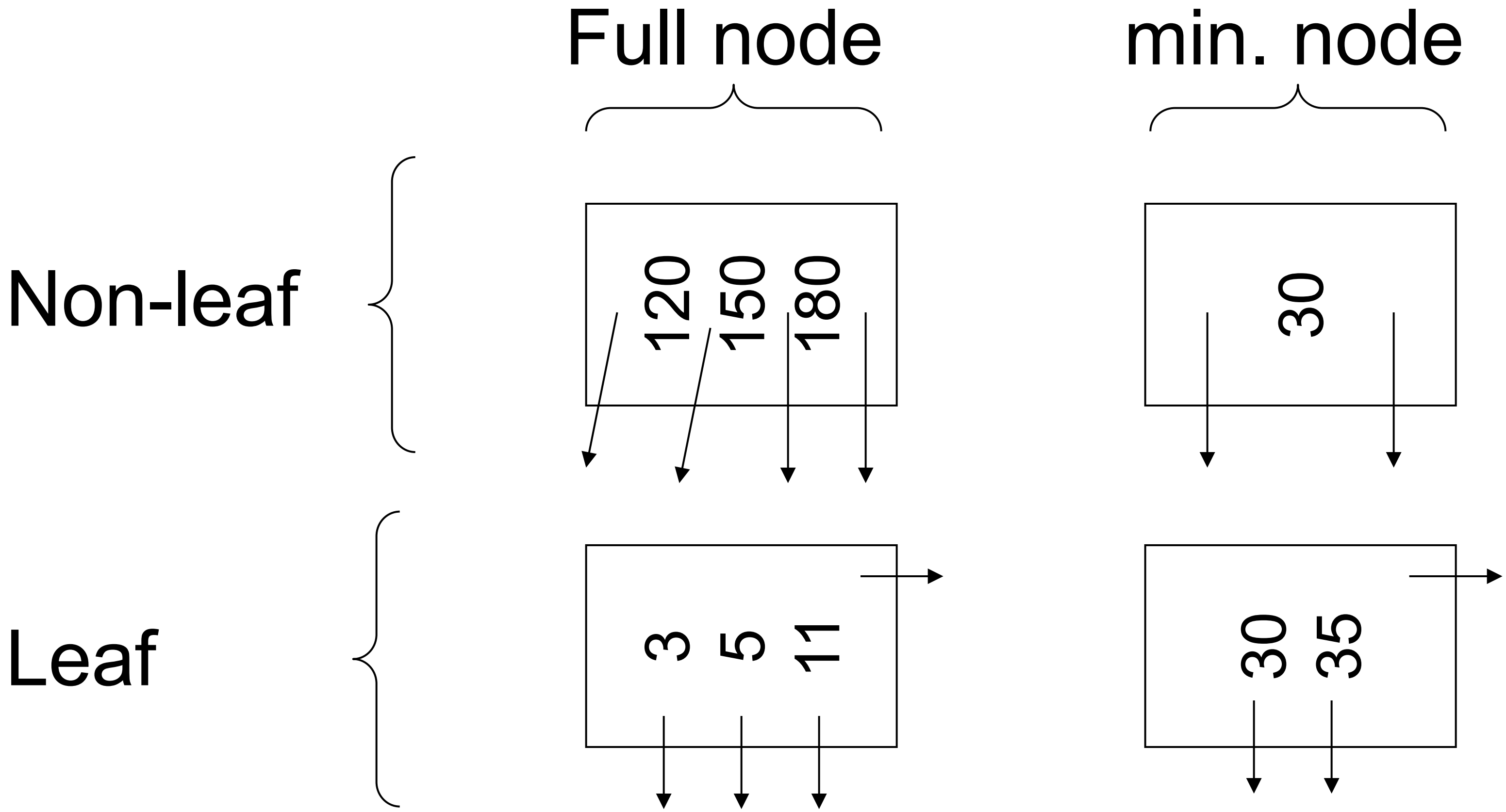
Don't Want Nodes to be Too Empty

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

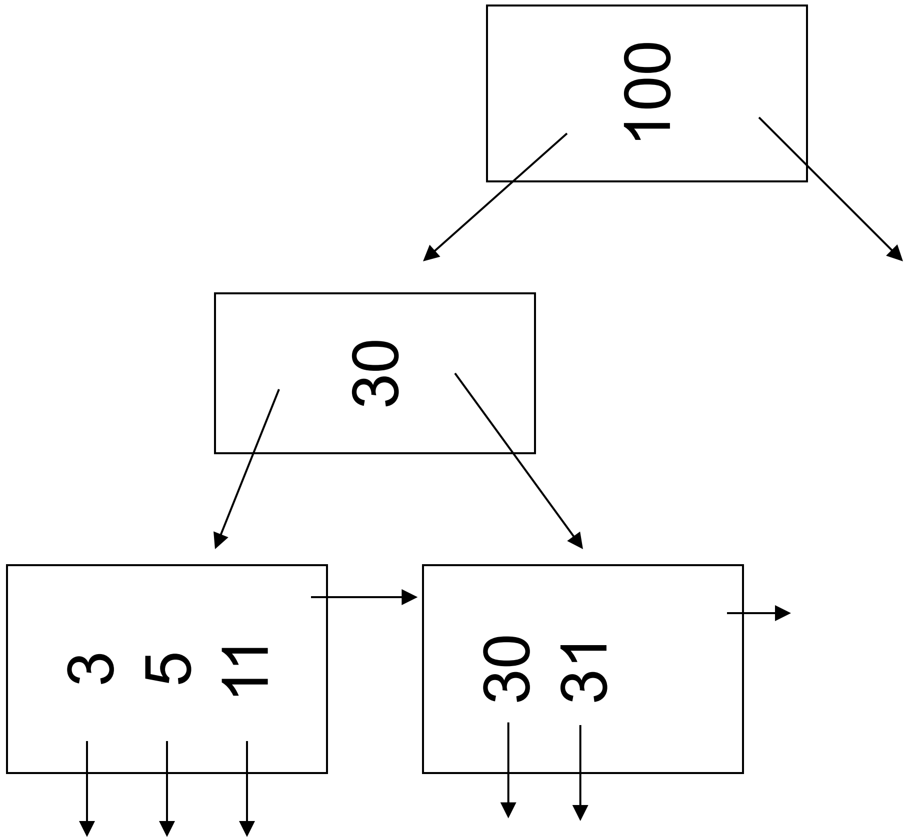
(b) leaf overflow

(c) non-leaf overflow

(d) new root

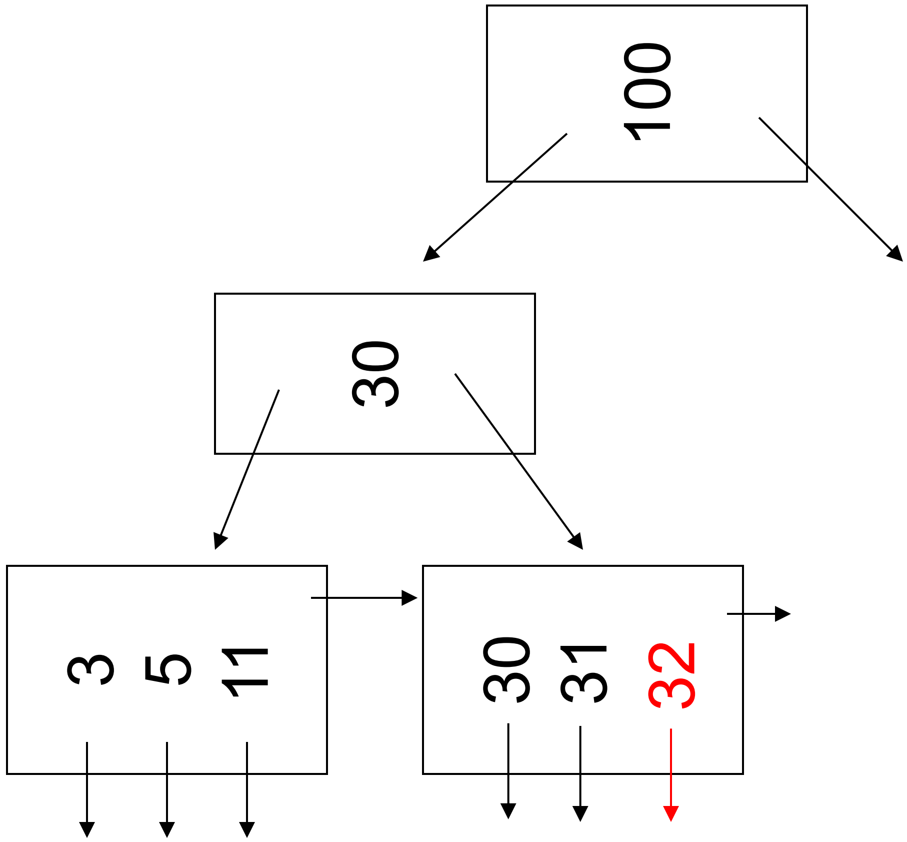
(a) Insert key = 32

n=3



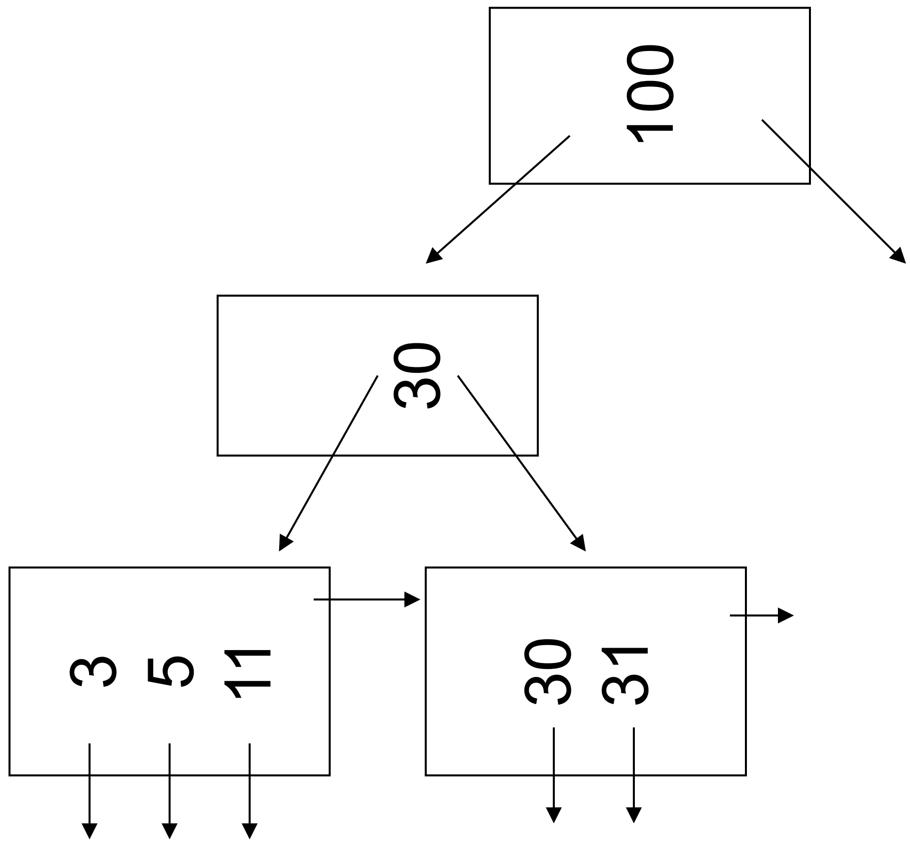
(a) Insert key = 32

n=3



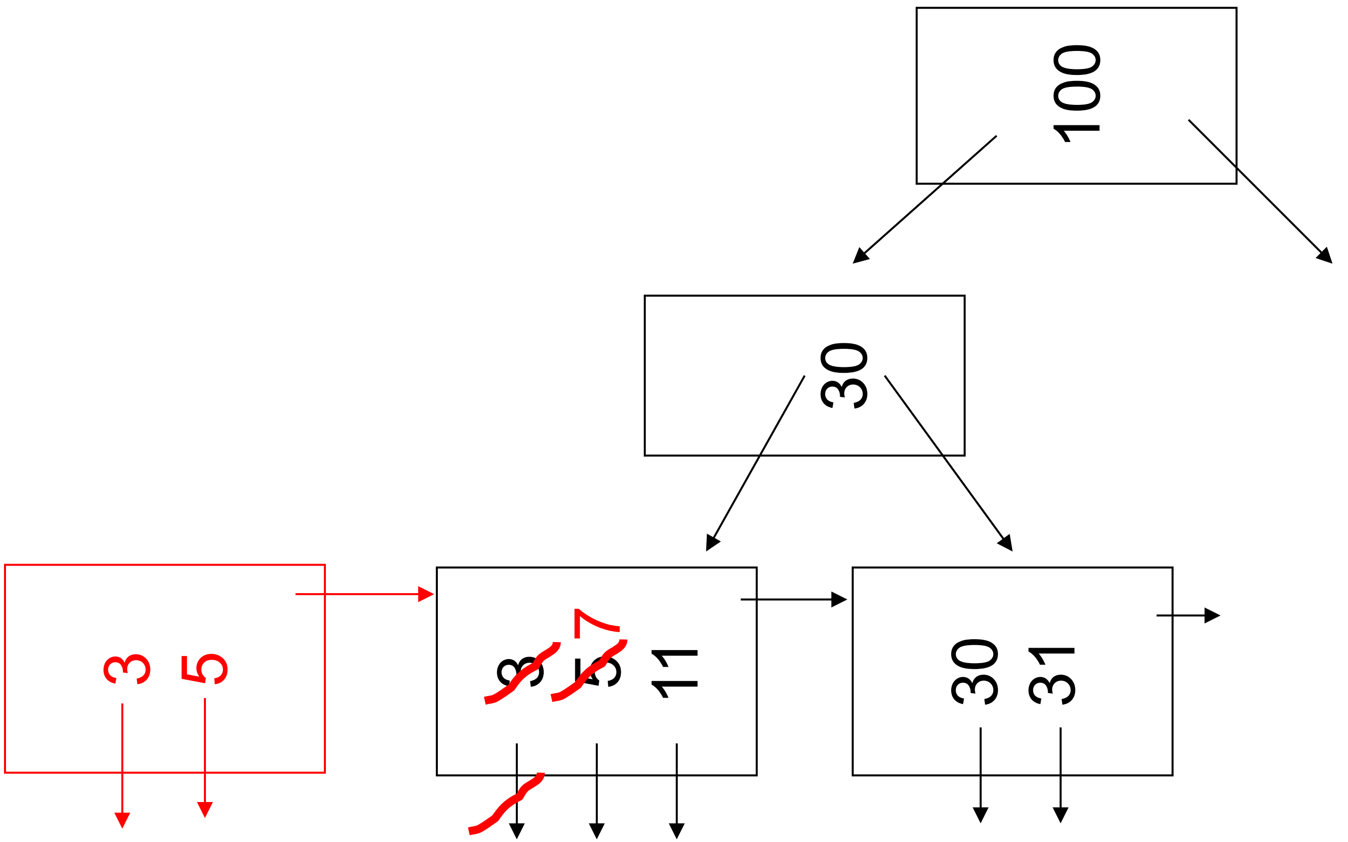
(a) Insert key = 7

n=3



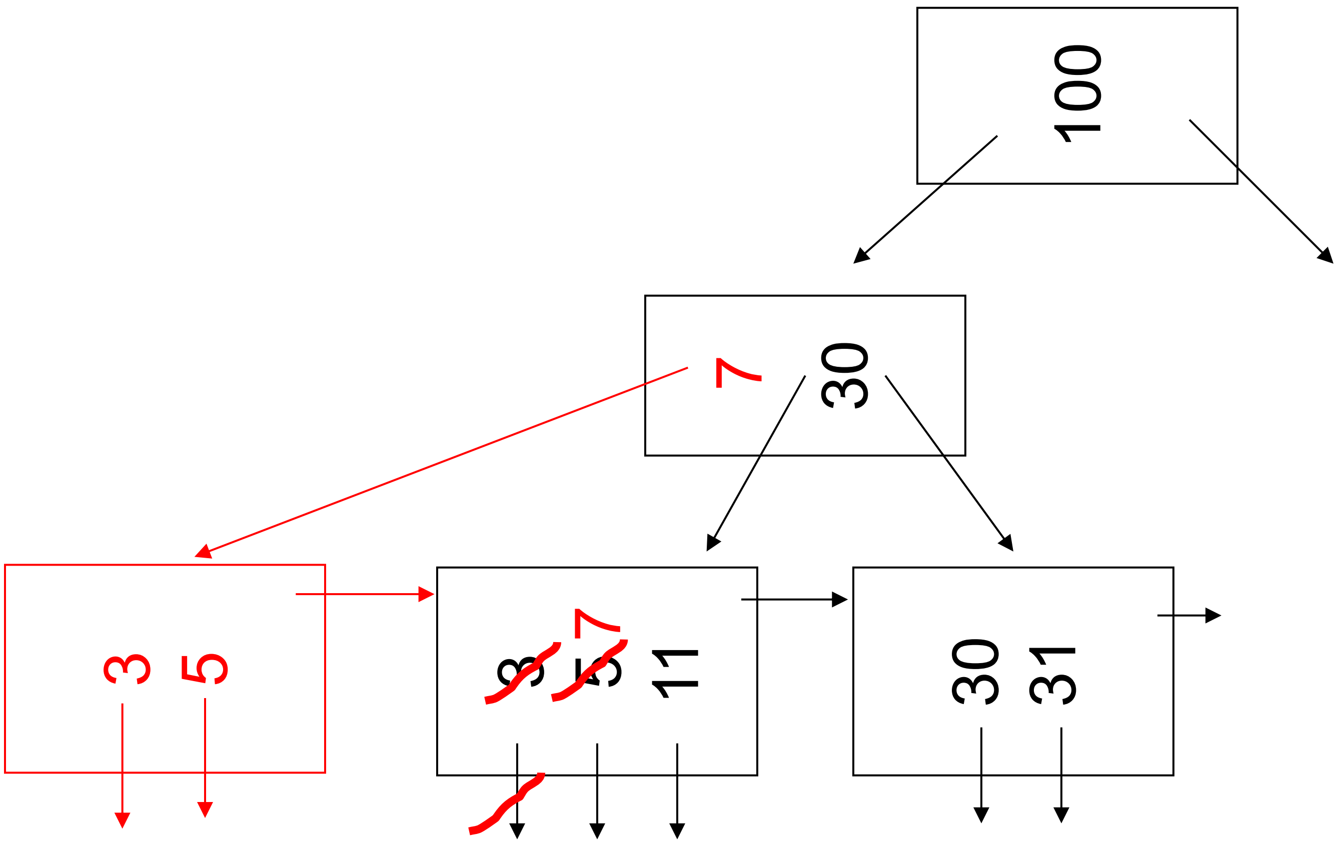
(a) Insert key = 7

n=3



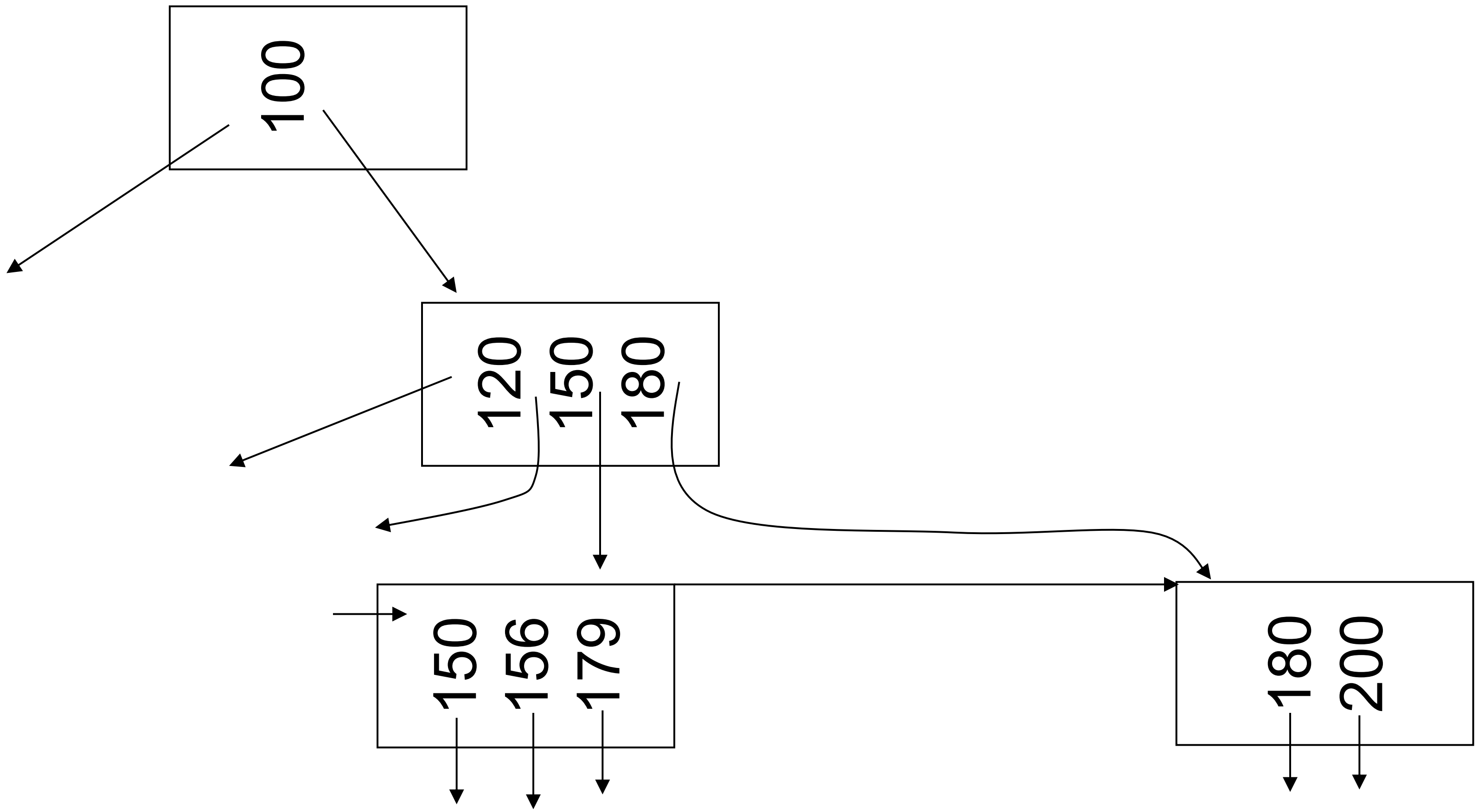
(a) Insert key = 7

n=3



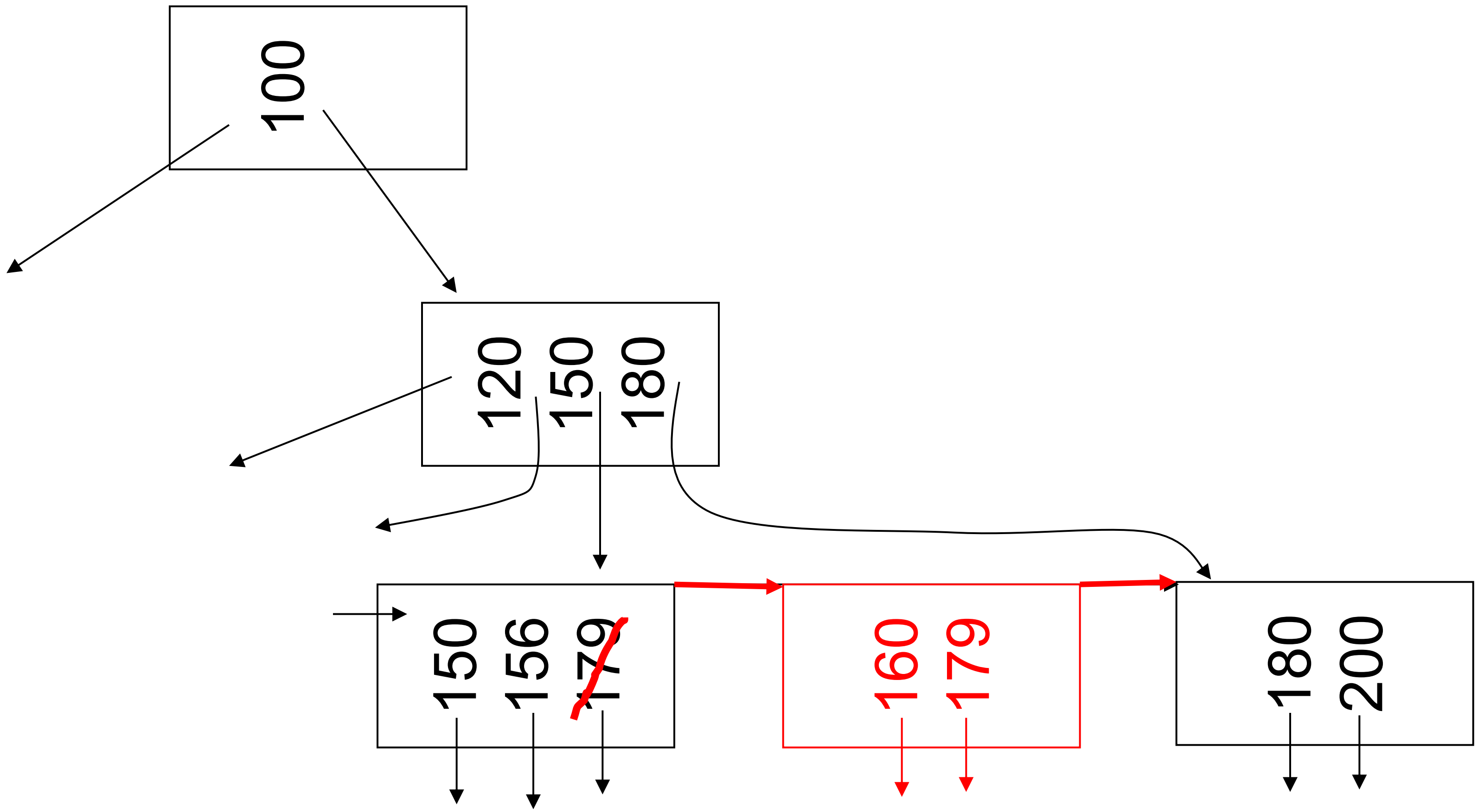
(c) Insert key = 160

n=3



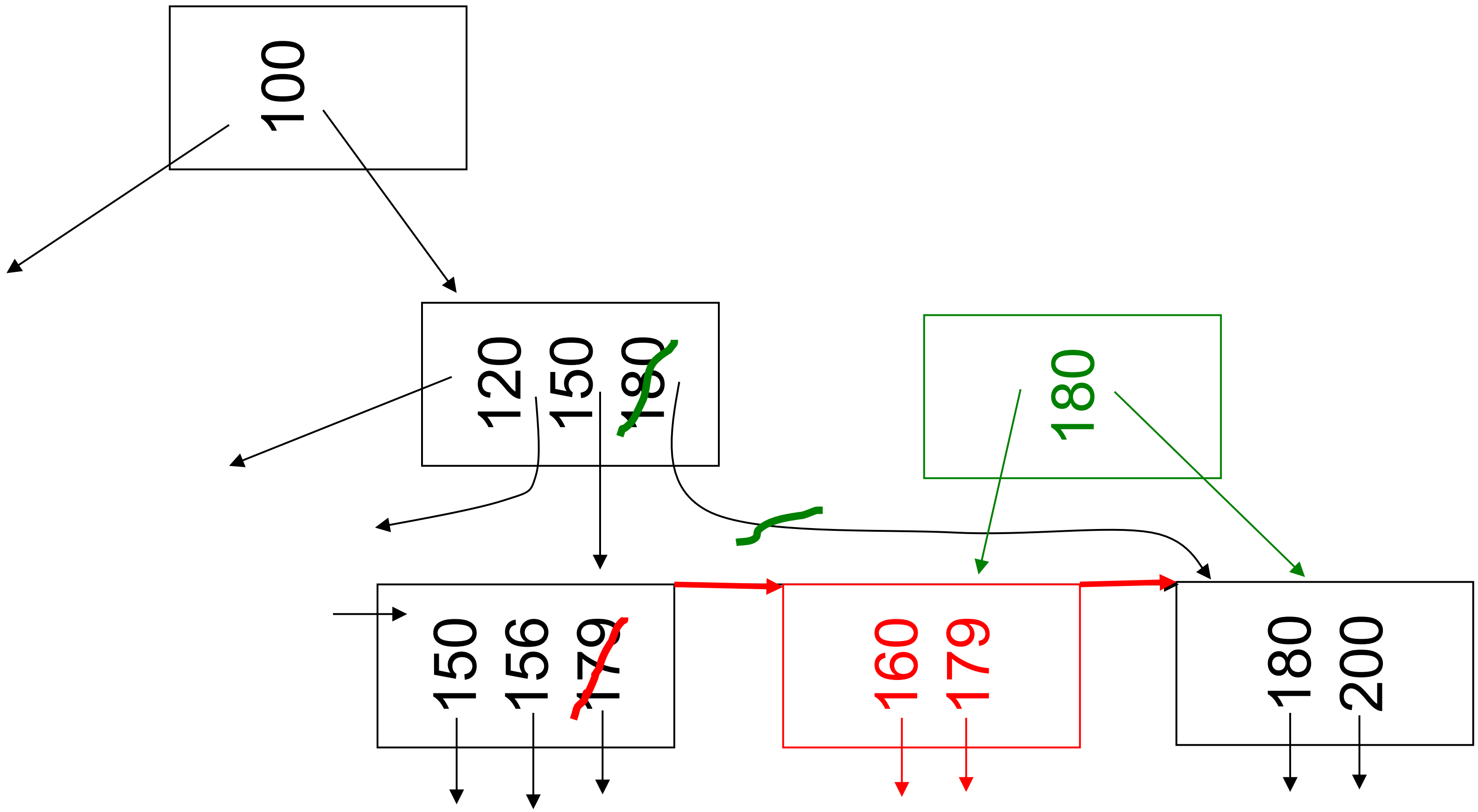
(c) Insert key = 160

n=3



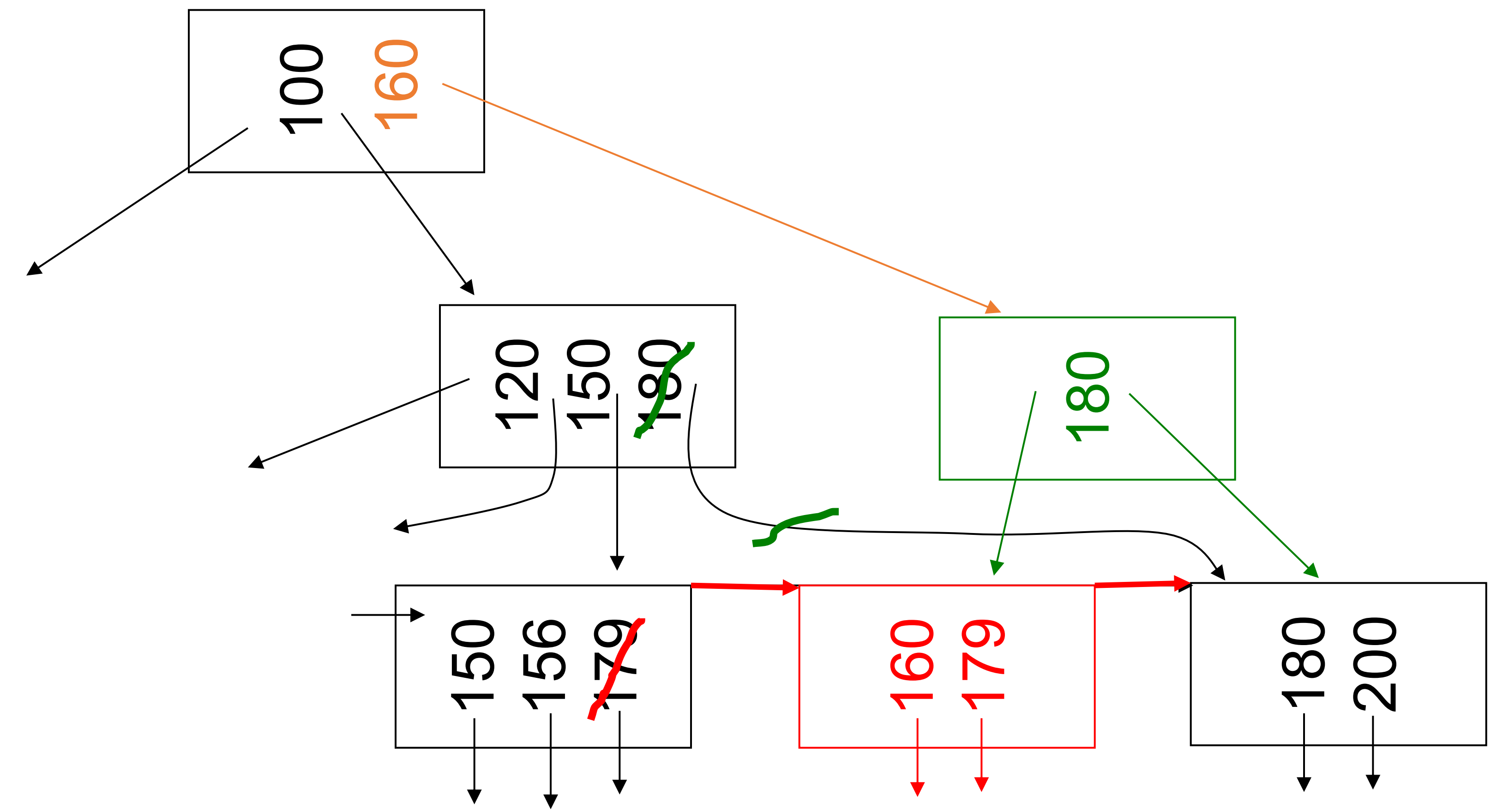
(c) Insert key = 160

n=3



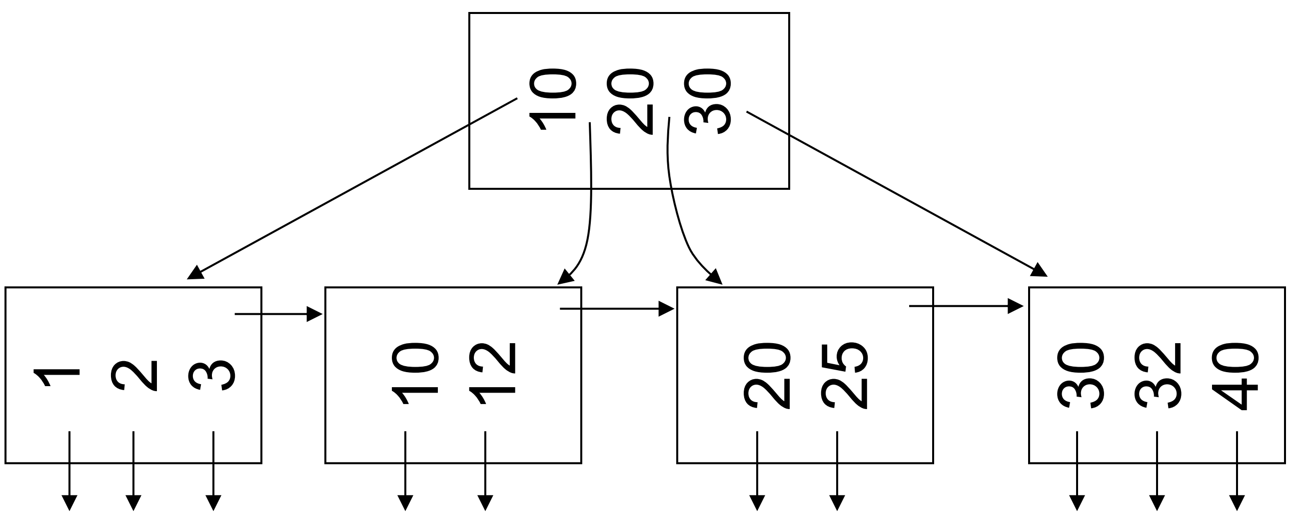
(c) Insert key = 160

n=3



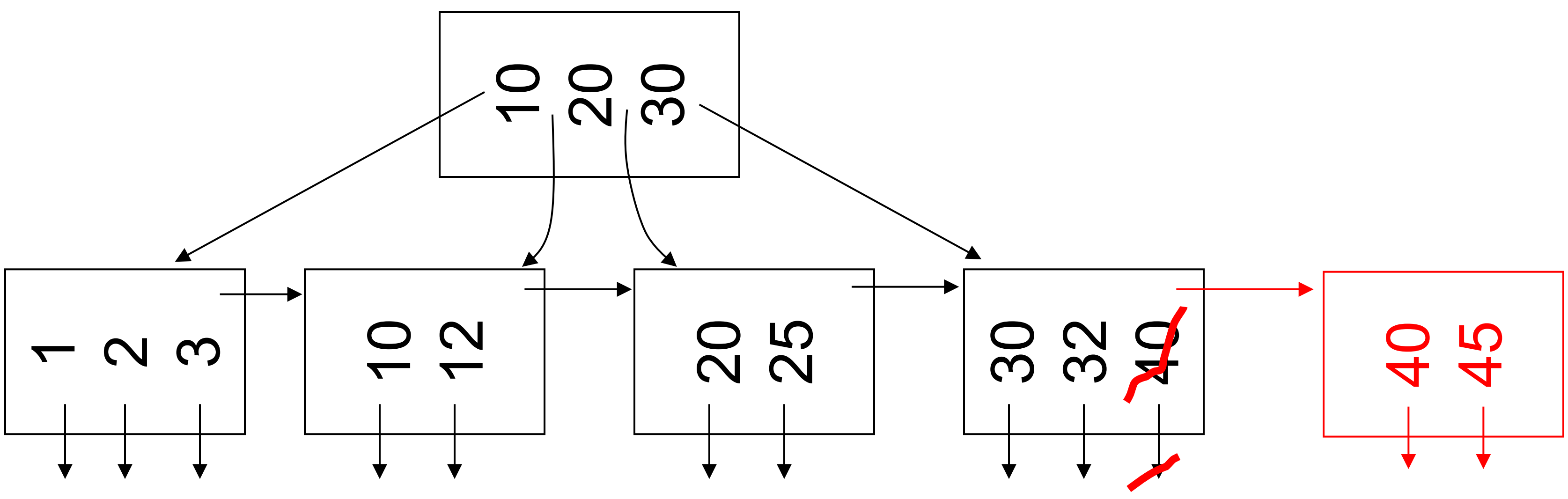
(d) New root, insert 45

n=3



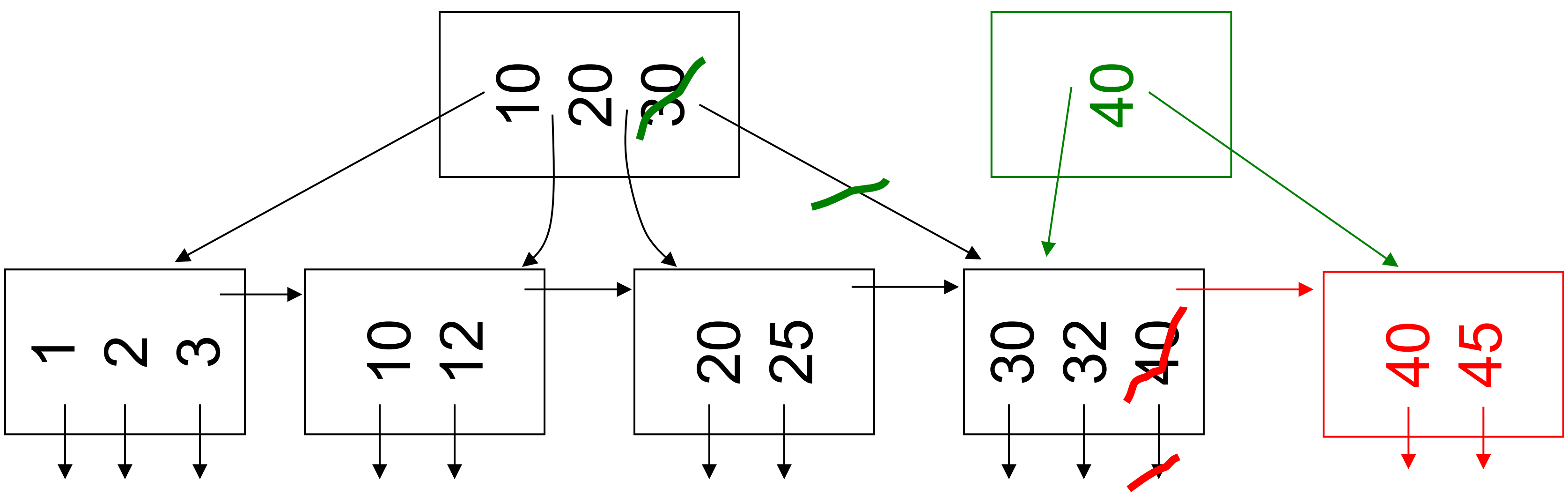
(d) New root, insert 45

n=3



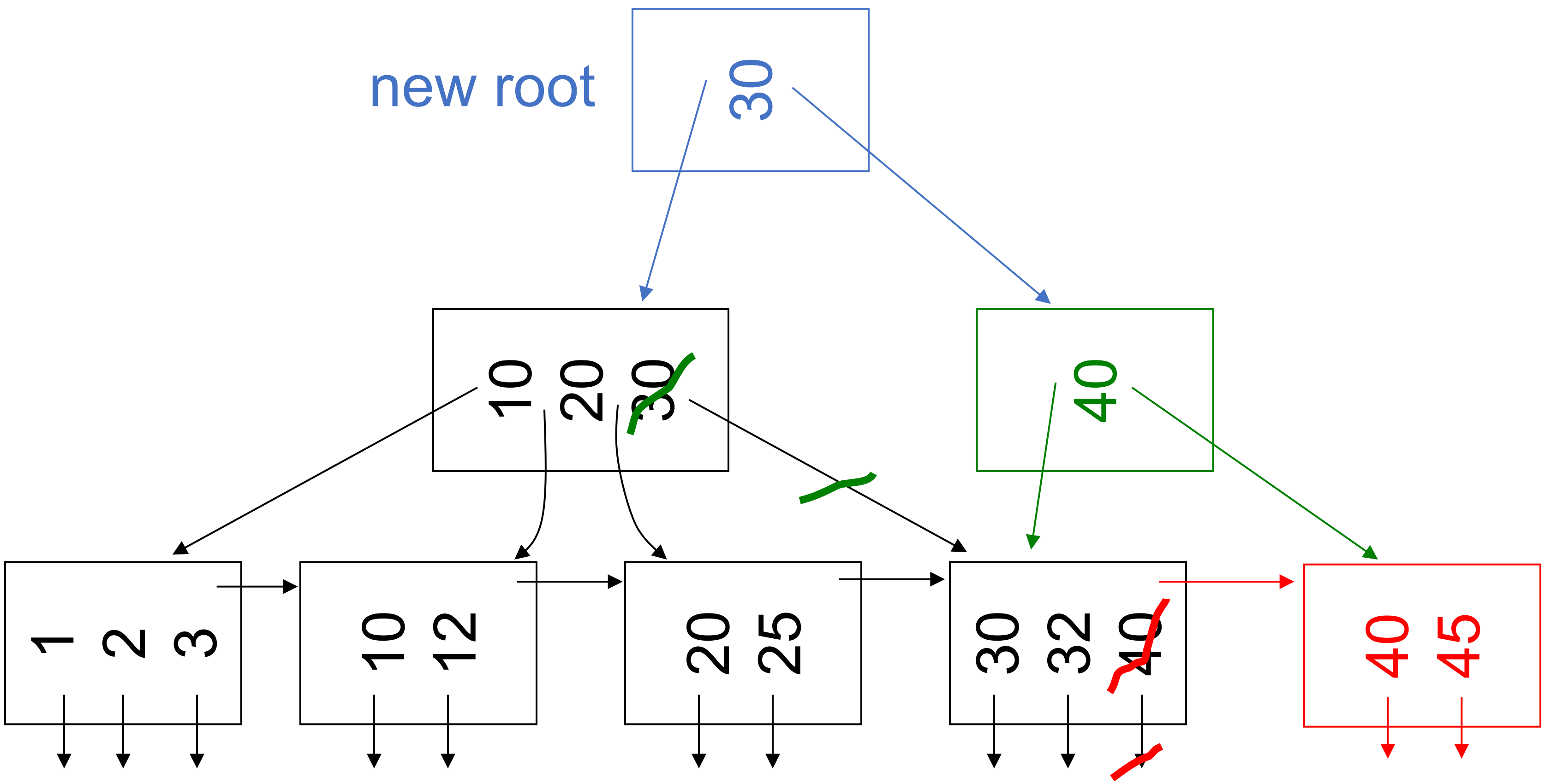
(d) New root, insert 45

n=3



(d) New root, insert 45

$\boxed{n=3}$



Deletion from B+tree

(a) Simple case: no example

(b) Coalesce with neighbor (sibling)

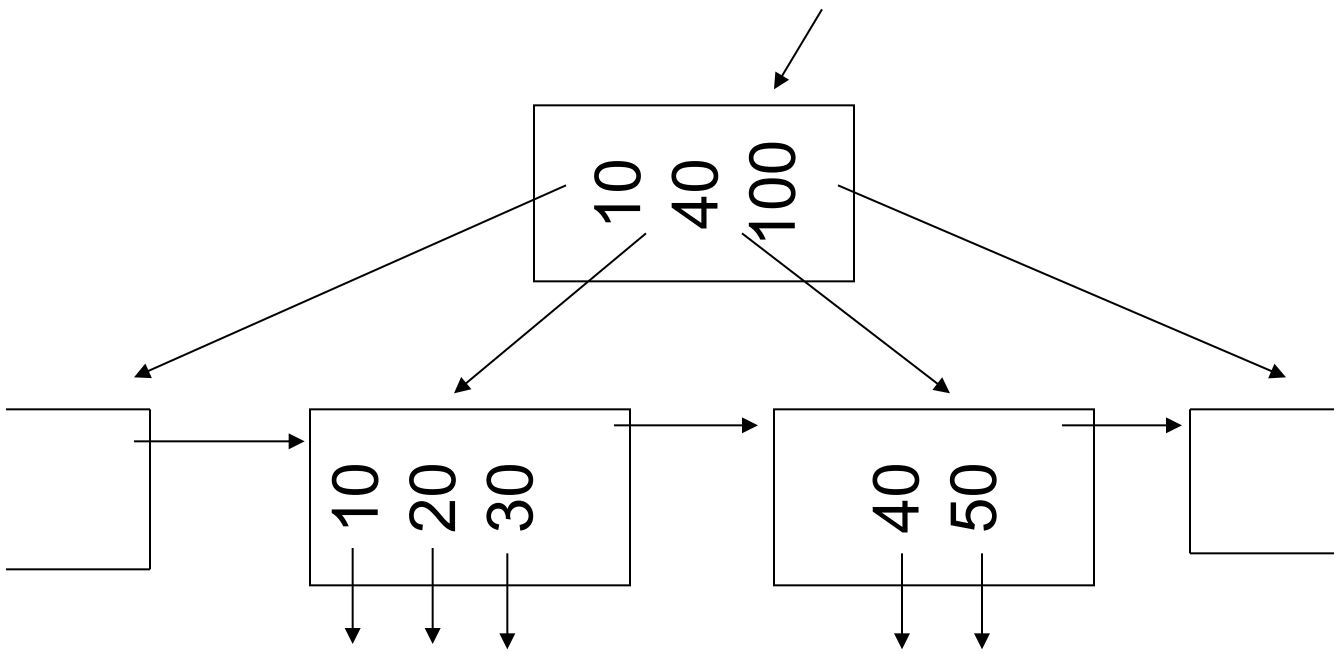
(c) Re-distribute keys

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

(b) Coalesce with sibling

» Delete 50

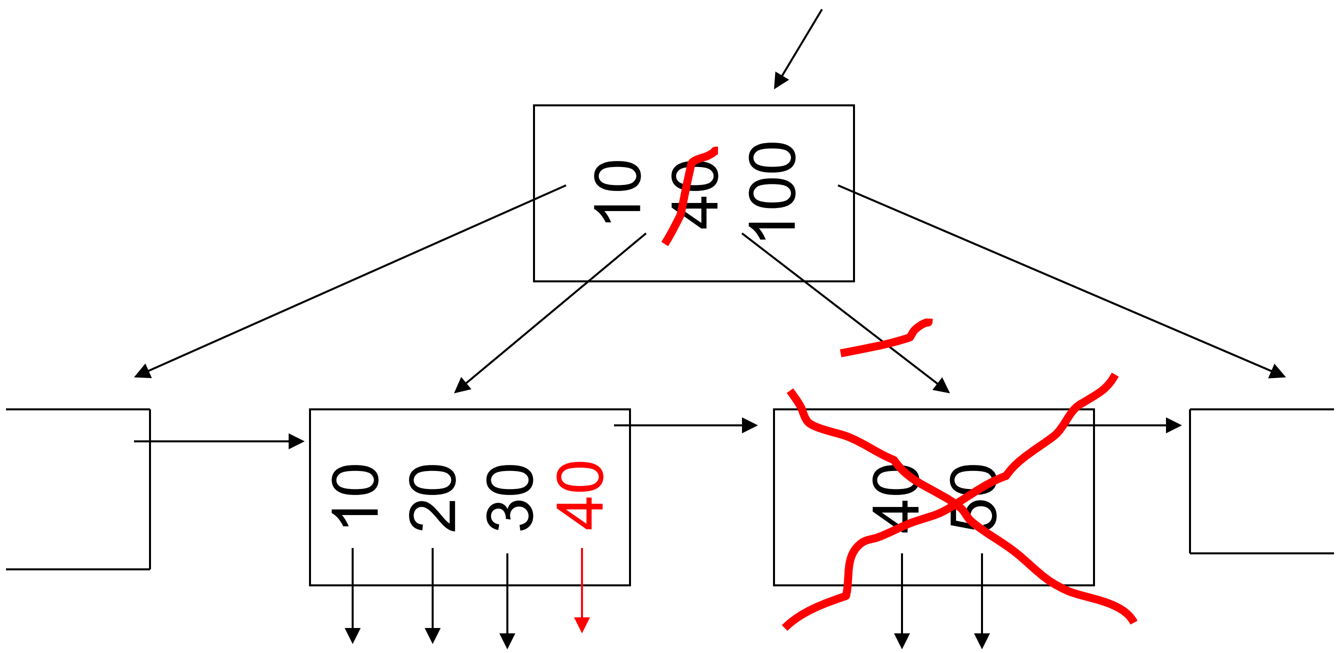
n=4



(b) Coalesce with sibling

» Delete 50

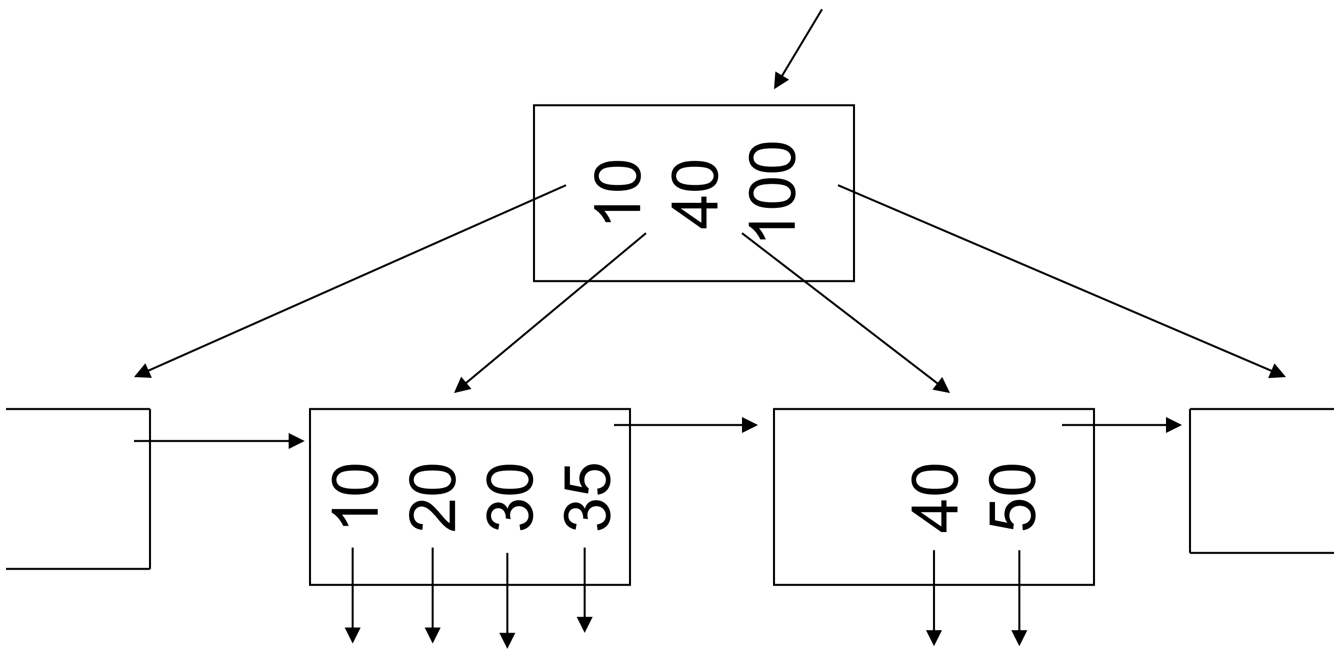
n=4



(c) Redistribute keys

» Delete 50

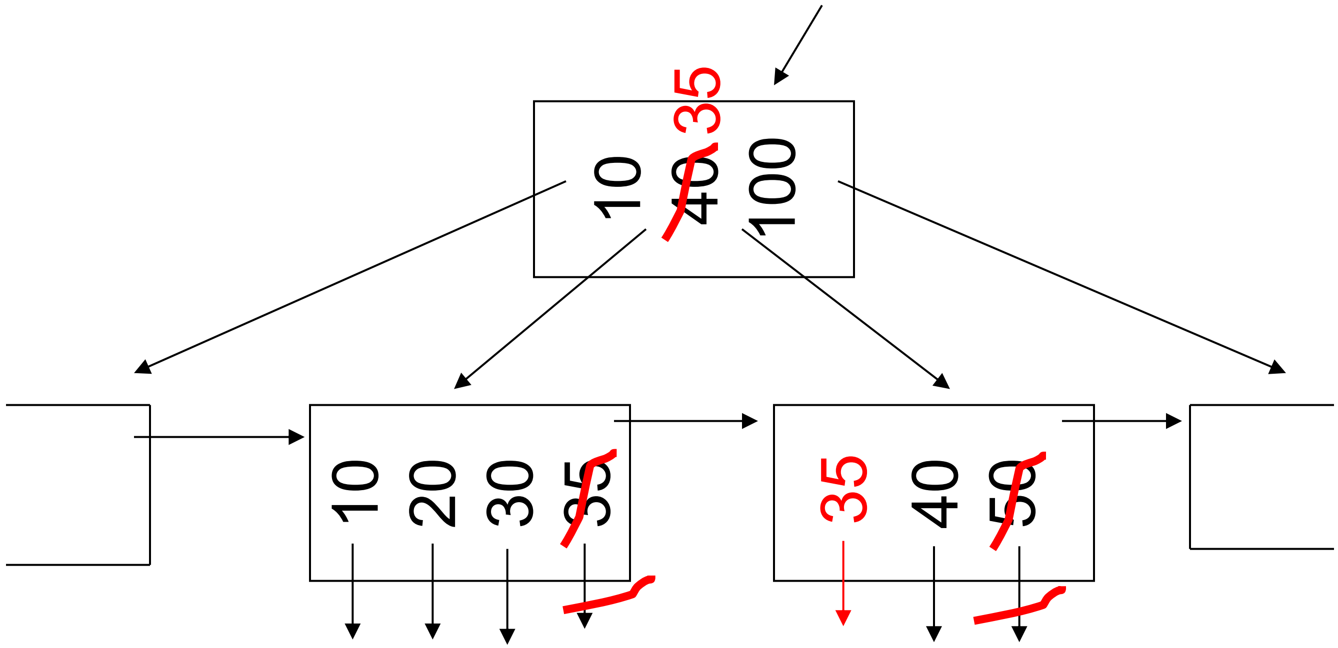
n=4



(c) Redistribute keys

» Delete 50

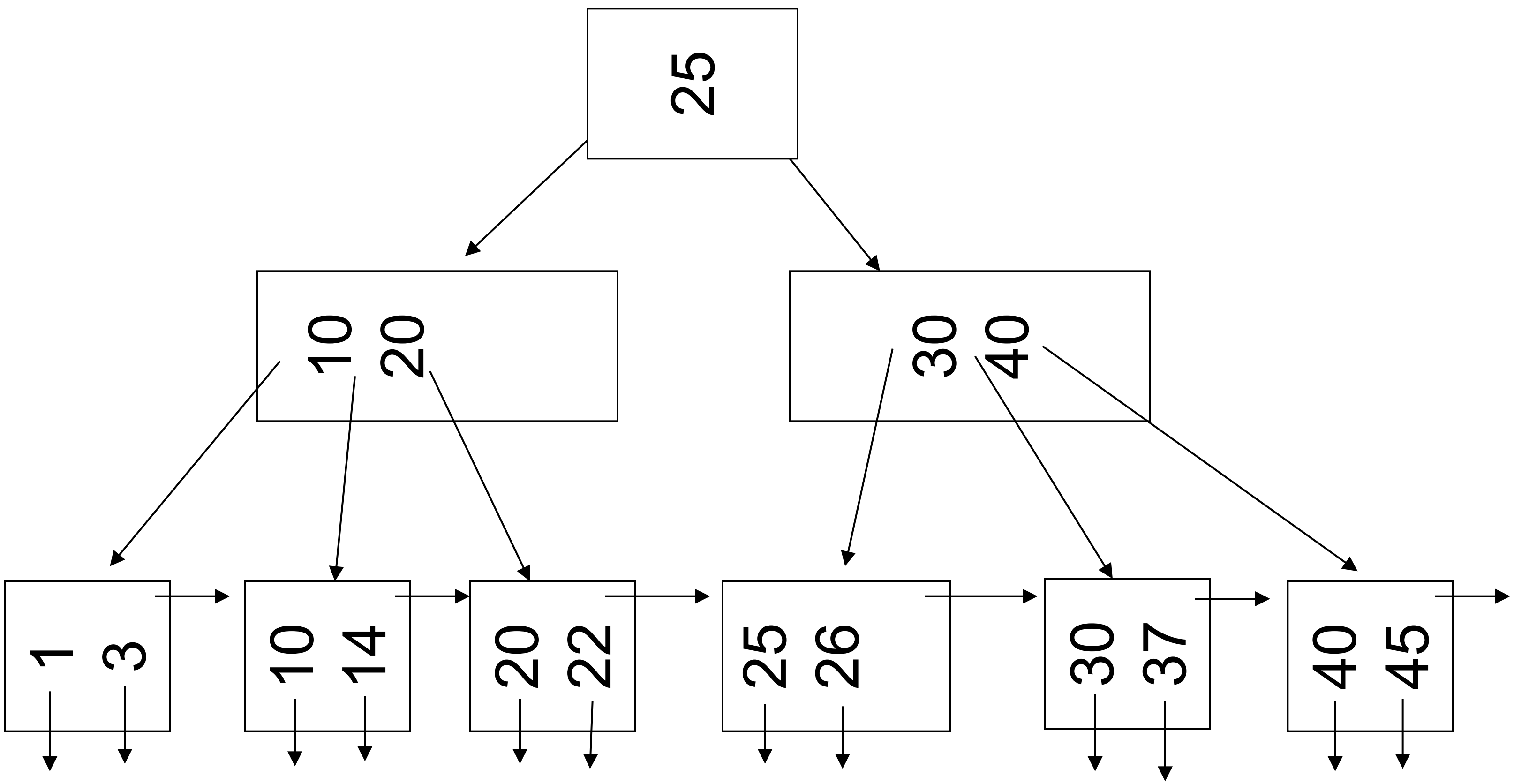
n=4



(d) Non-leaf coalesce

n=4

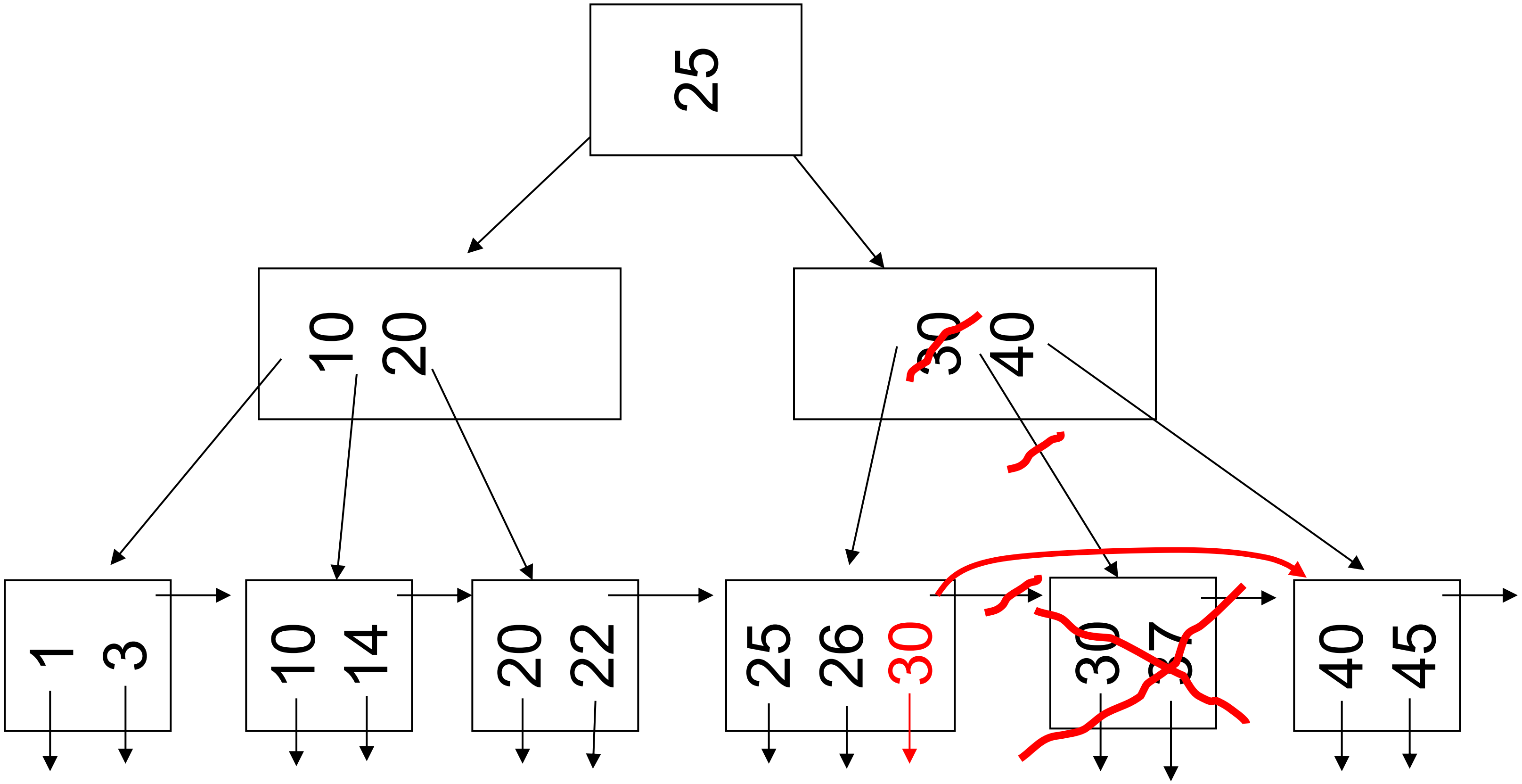
– Delete 37



(d) Non-leaf coalesce

– Delete 37

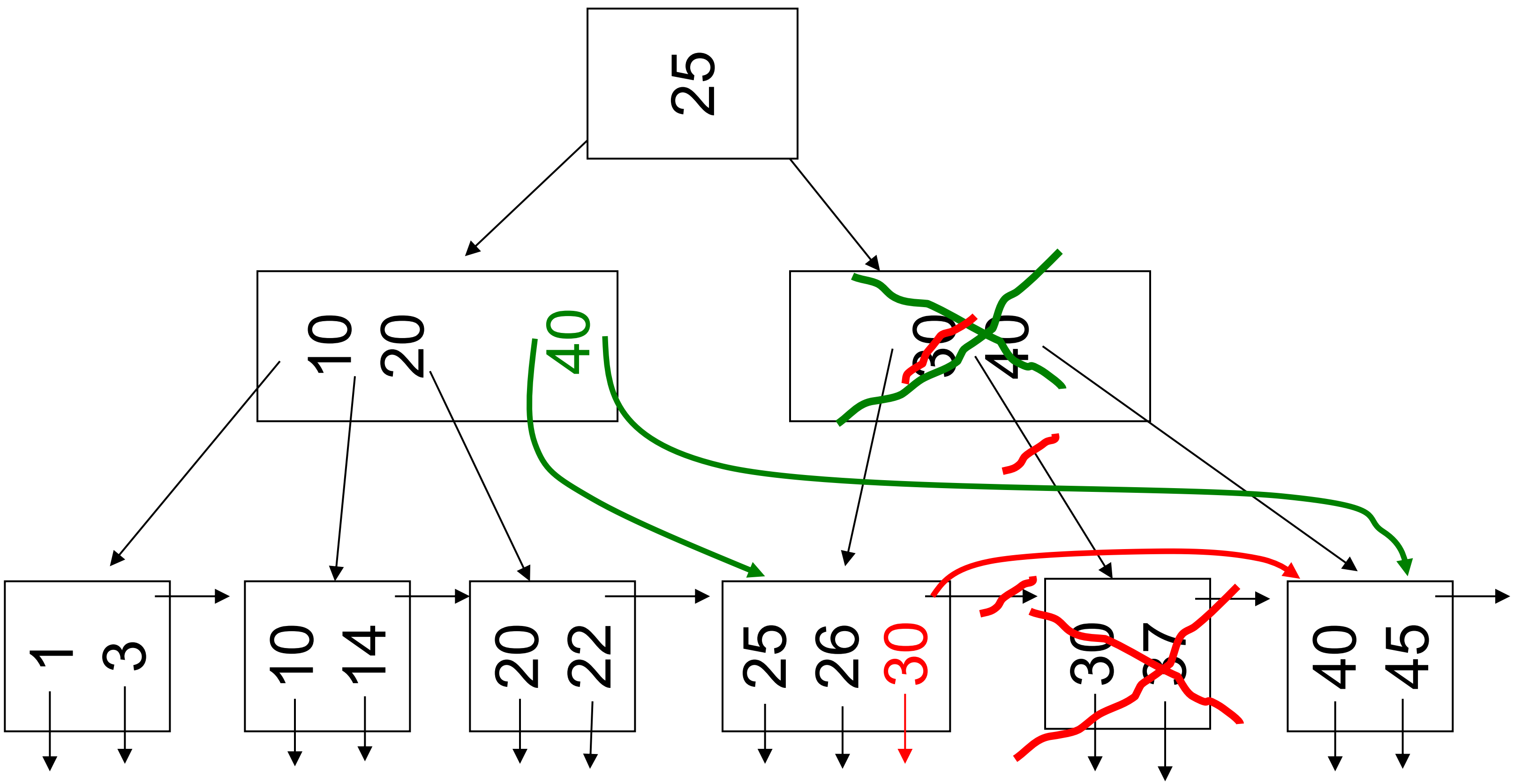
n=4



(d) Non-leaf coalesce

– Delete 37

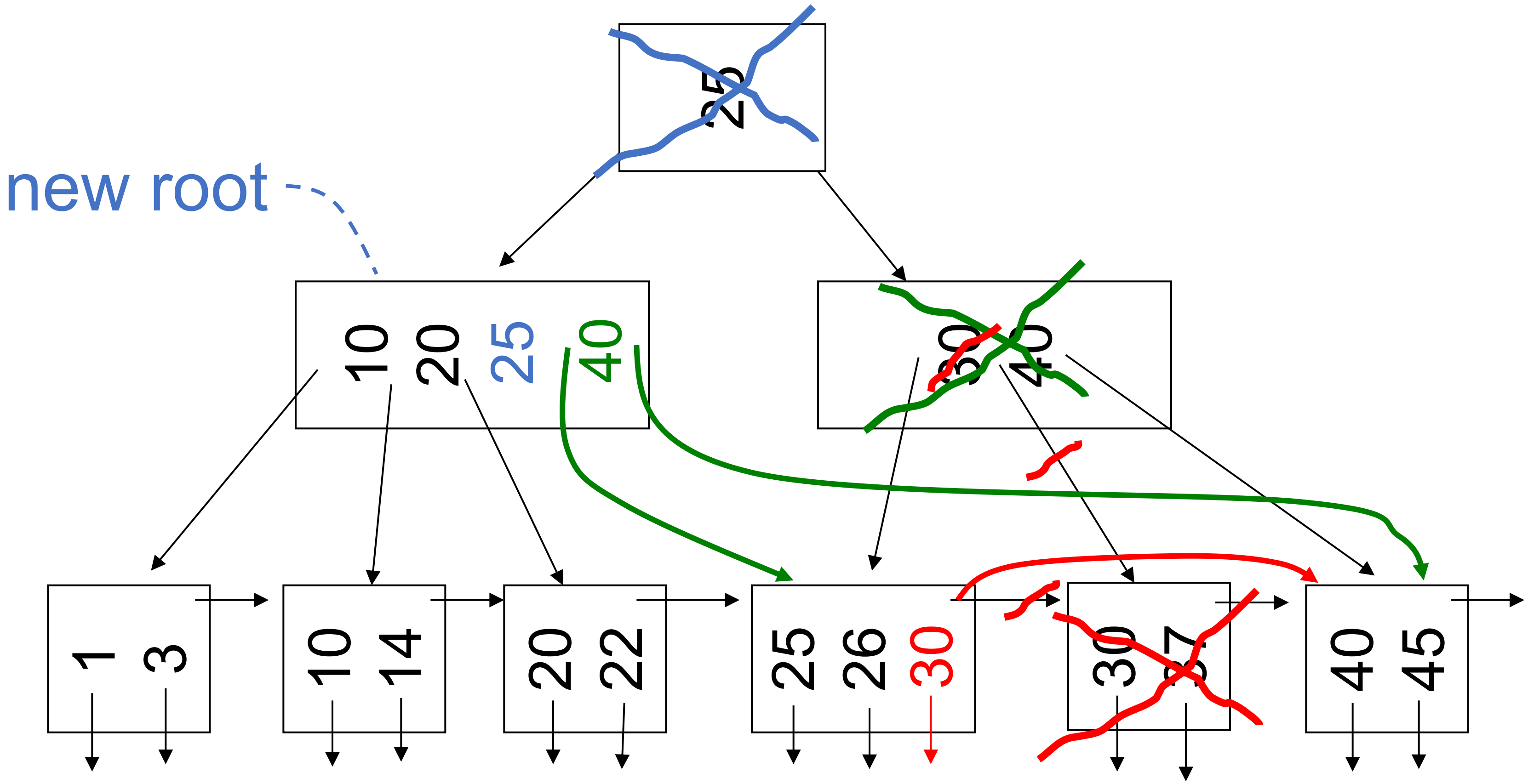
n=4



(d) Non-leaf coalesce

– Delete 37

n=4



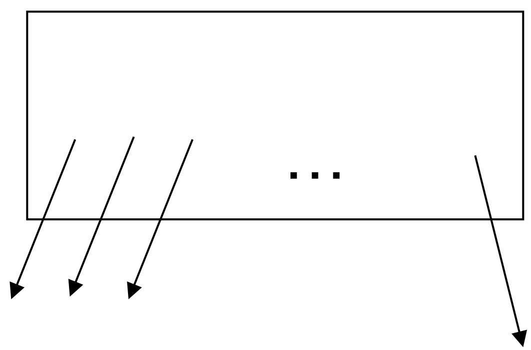
B+ Tree Deletion in Practice

Often, coalescing is not implemented

» Too hard and not worth it! (Most datasets just grow in size over time.)

Interesting Problem:

For B+ tree, how large should n be?



n is number of keys / node

Sample Assumptions:

(1) Time to read node from disk is
$(S + Tn)$ msec.

Sample Assumptions:

(1) Time to read node from disk is
$(S + Tn)$ msec.

(2) Once block in memory, use binary search to locate key:
$(a + b \log_2 n)$ msec.

For some constants a, b; Assume $a \ll S$

Sample Assumptions:

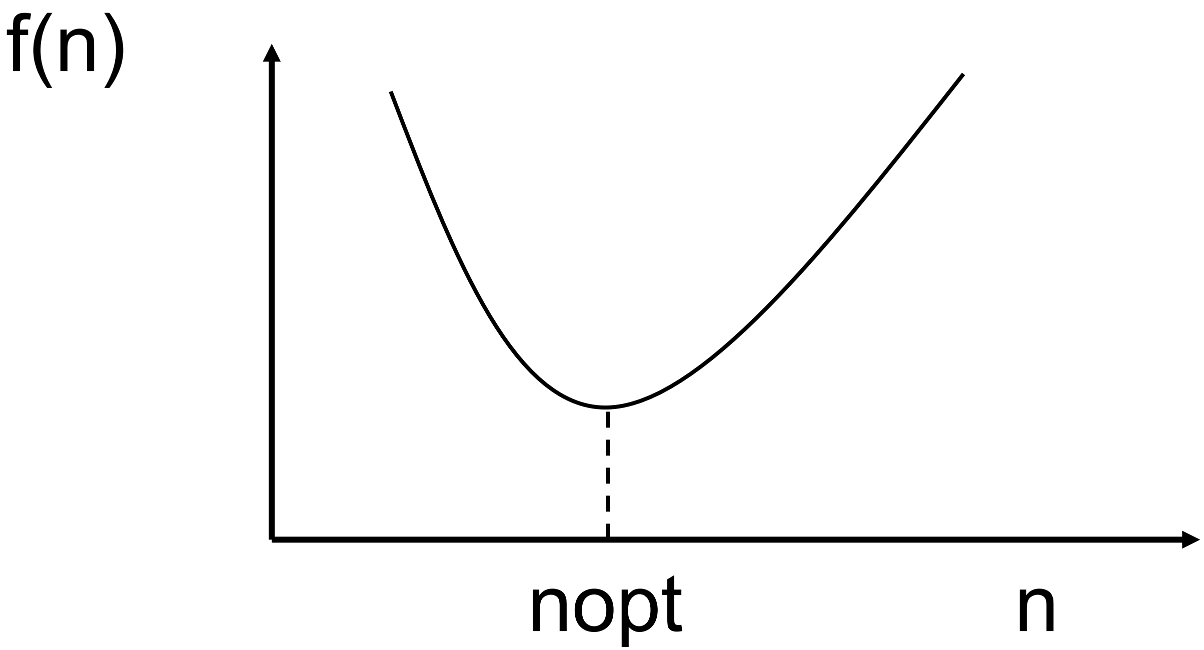
(1) Time to read node from disk is
$(S + Tn)$ msec.

(2) Once block in memory, use binary search to locate key:
$(a + b \log_2 n)$ msec.

For some constants a, b; Assume $a << S$

(3) Assume B+tree is full, i.e., # nodes to examine is $\log_n N$ where N = # records

Can Get:
f(n) = time to find a record



Find n_{opt} by setting $f'(n) = 0$

Answer is n_{opt} = “a few hundred” in practice

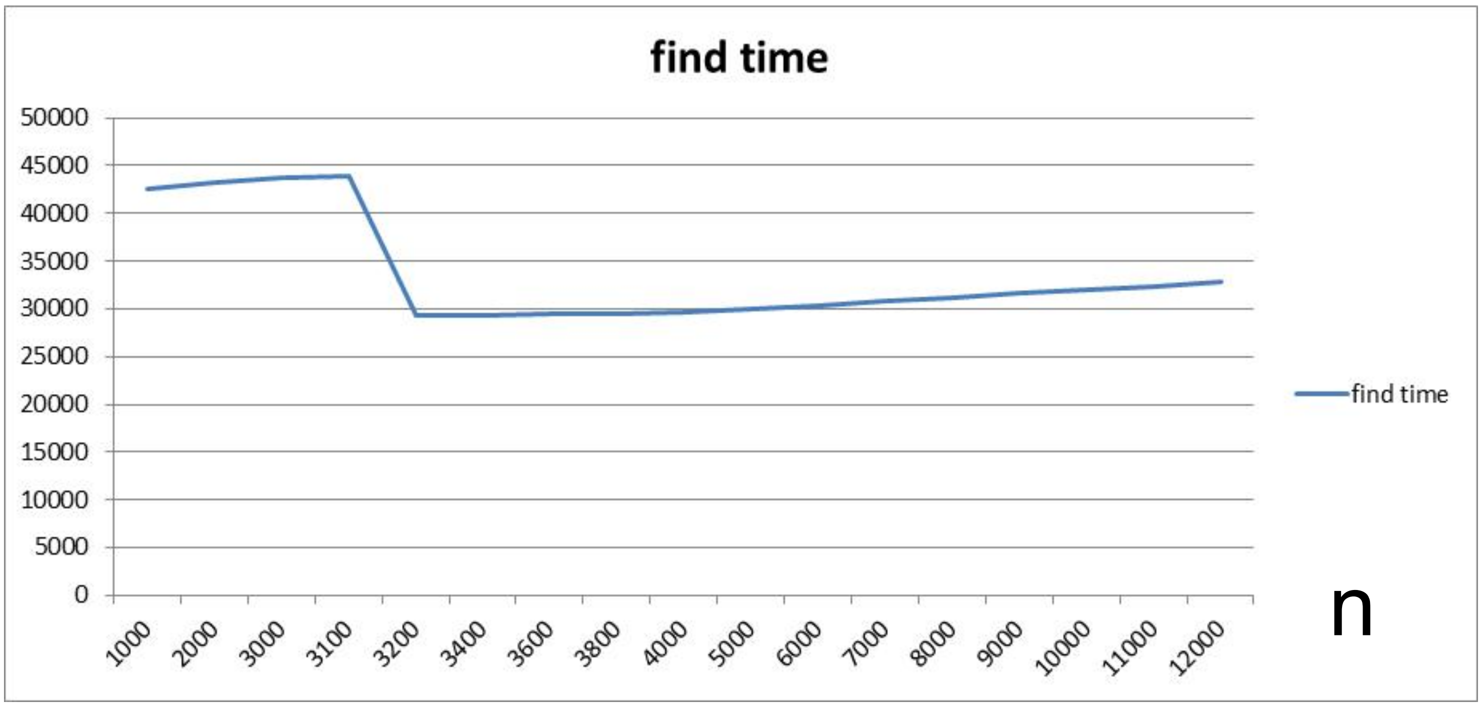
Exercise

S =	14000 μs
T =	0.2 μs
b =	0.002 μs
a =	0 μs
N =	10,000,000

$$f(n) = \log_n N * (S + T\,n + a + b \log_2 n)$$

N = 10 Million Records

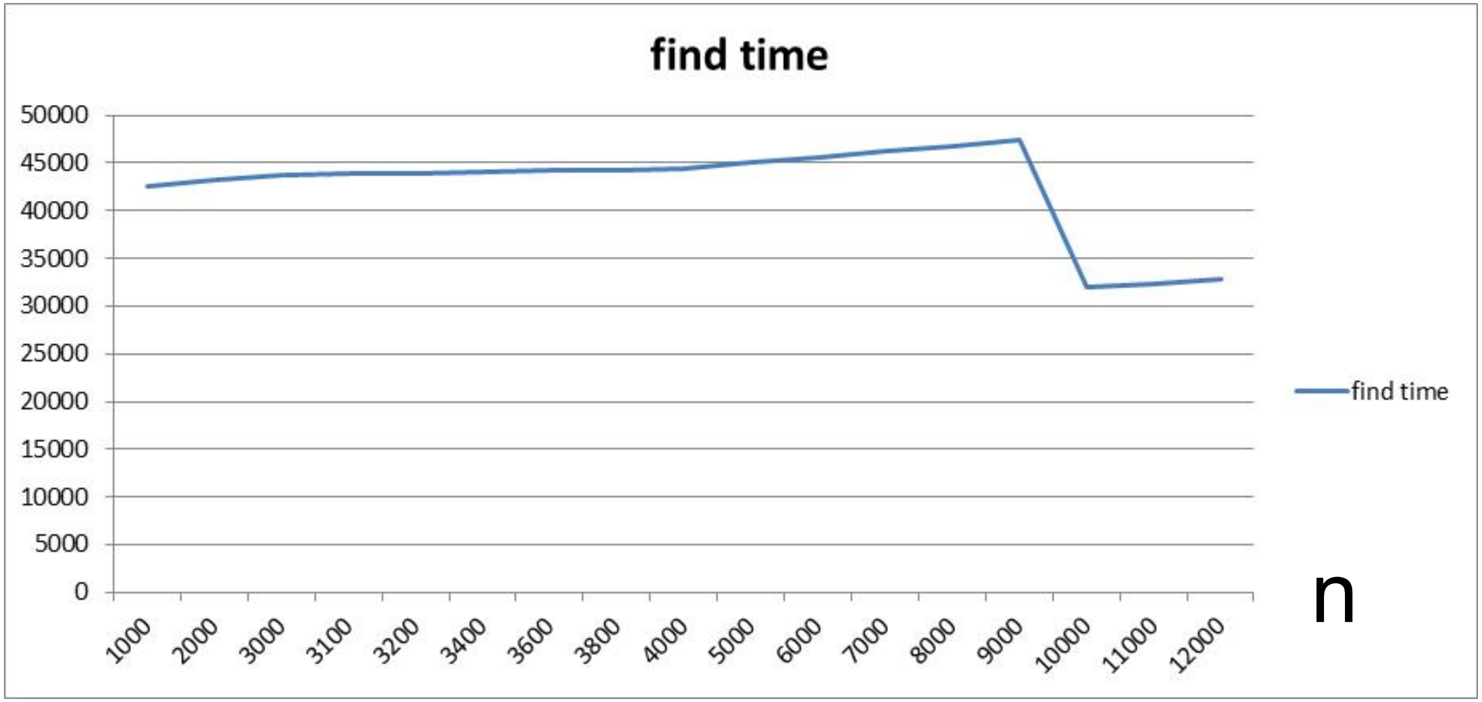
S=	14000
T=	0.2
b=	0.002
a=	0
N=	10,000,000



times in microseconds

N = 100 Million Records

S=	14000
T=	0.2
b=	0.002
a=	0
N=	100,000,000



n

times in microseconds

Some Types of Indexes

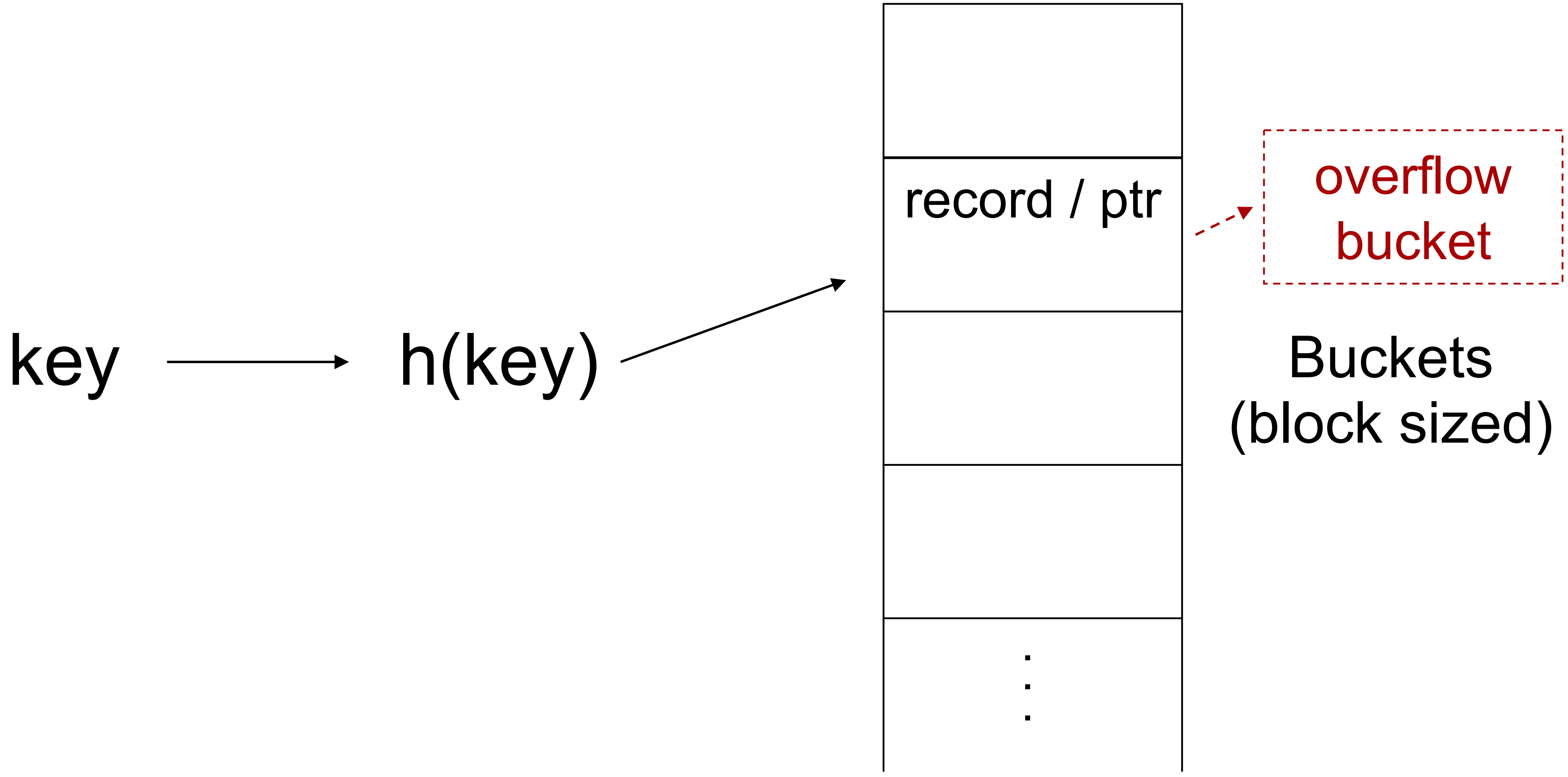
Conventional indexes

B-trees

Hash indexes

Multi-key indexing

Hash Indexes



Chaining is used to handle bucket overflow

Hash vs Tree Indexes

+ O(1) instead of O(log N) disk accesses

– Can't efficiently do range queries

Challenge: Resizing

Hash tables try to keep occupancy in a fixed range (50-80%) and slow down beyond that

» Too much chaining

How to resize the table when this happens?

» **In memory:** just move everything, amortized cost is pretty low

» **On disk:** moving everything is expensive!

Extendible Hashing

Tree-like design for hash tables that allows cheap resizing while requiring 2 IOs / access

Extendible Hashing: 2 Ideas

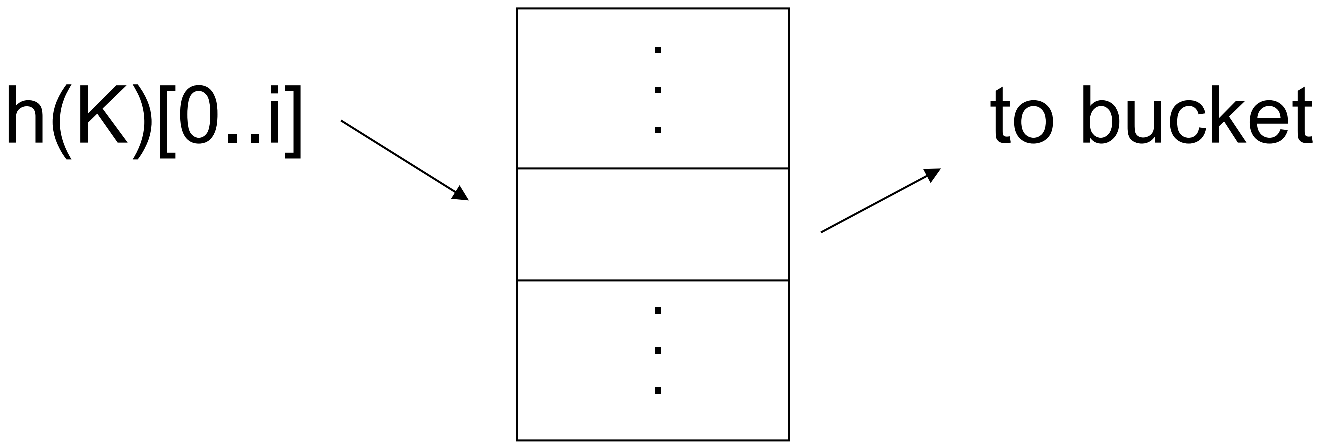
(a) Use i of b bits output by hash function

$$h(K) \rightarrow \overbrace{\boxed{\underbrace{001}_{i}10101}}^{b}$$

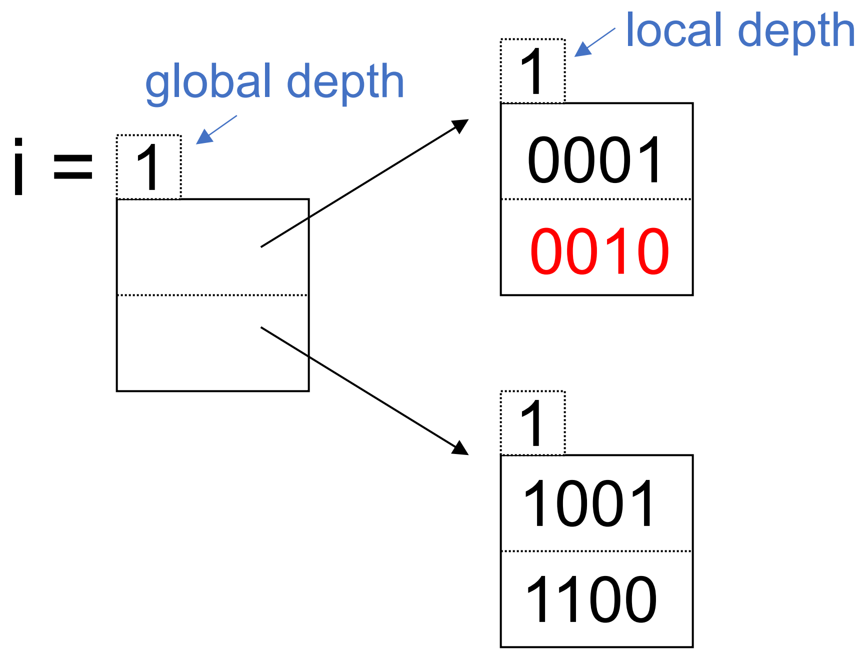
i will grow over time; the first i bits of each key's hash are used to map it to a bucket

Extendible Hashing: 2 Ideas

(b) Use a directory with pointers to buckets

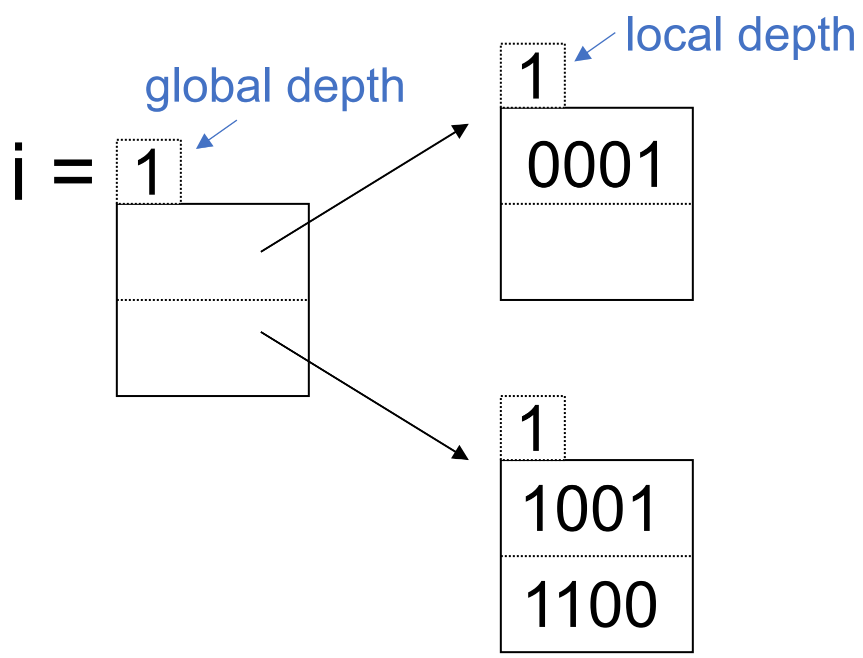


Example: 4-bit h(K), 2 keys/bucket



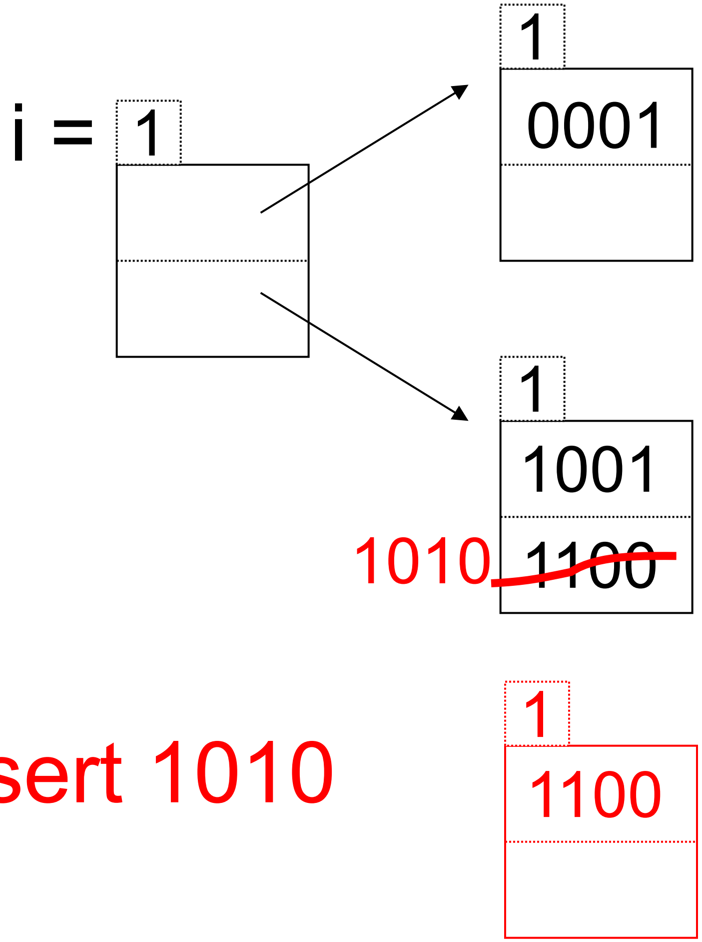
Insert 0010

Example: 4-bit h(K), 2 keys/bucket



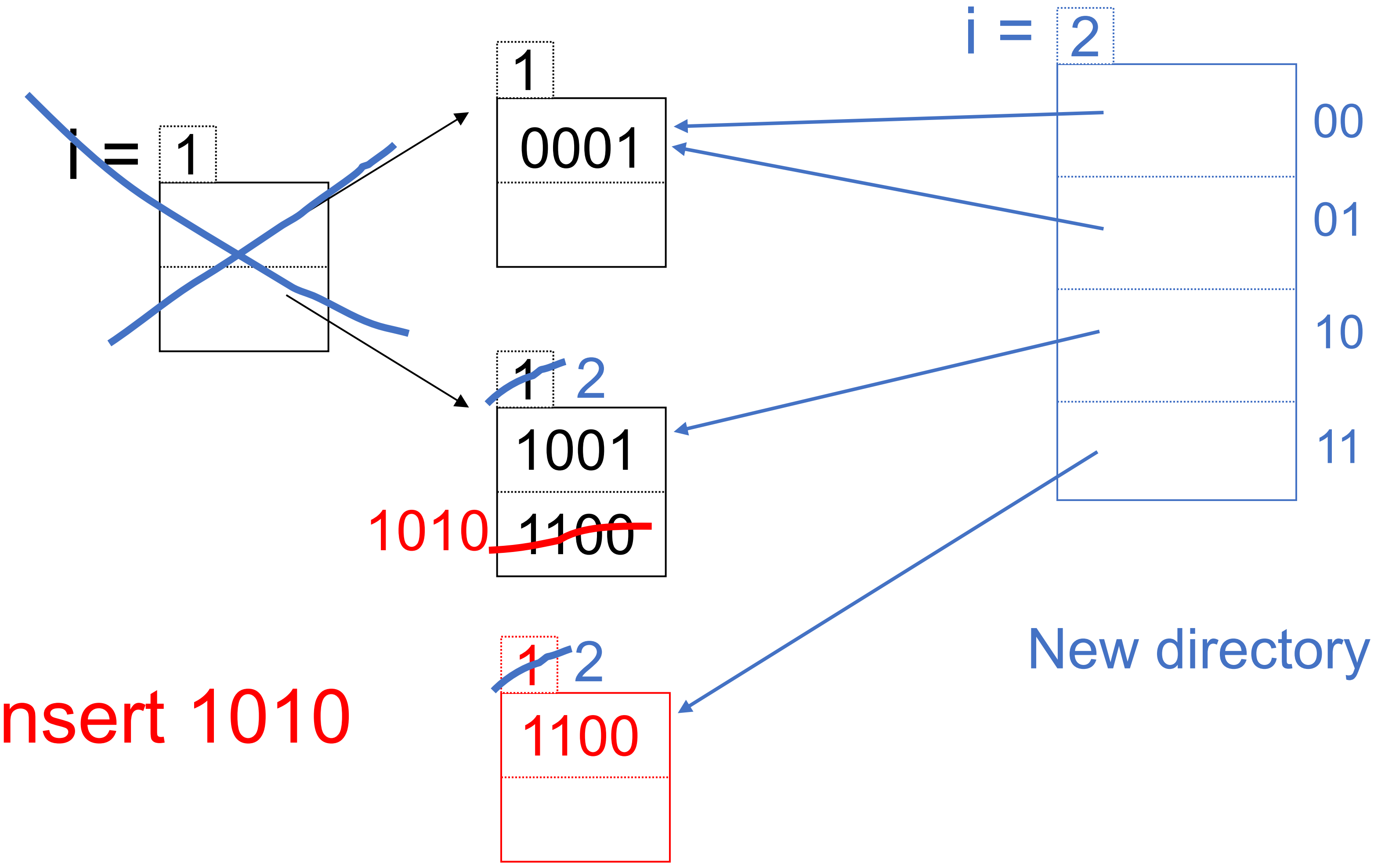
Insert 1010

Example: 4-bit h(K), 2 keys/bucket



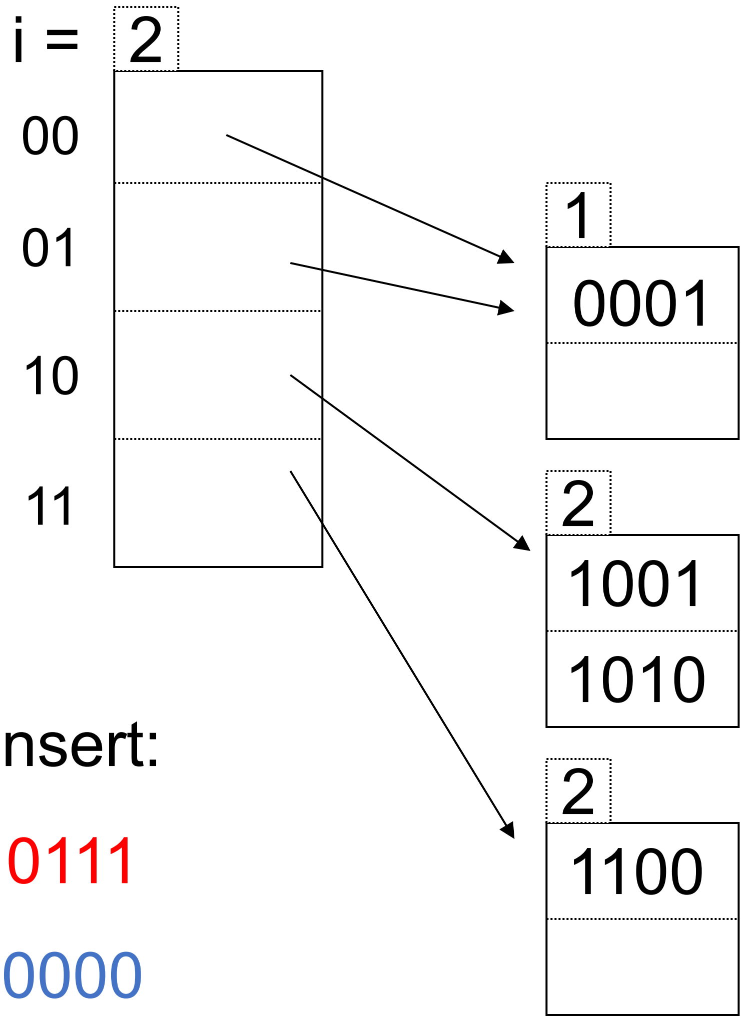
Insert 1010

Example: 4-bit h(K), 2 keys/bucket



Insert 1010

Example

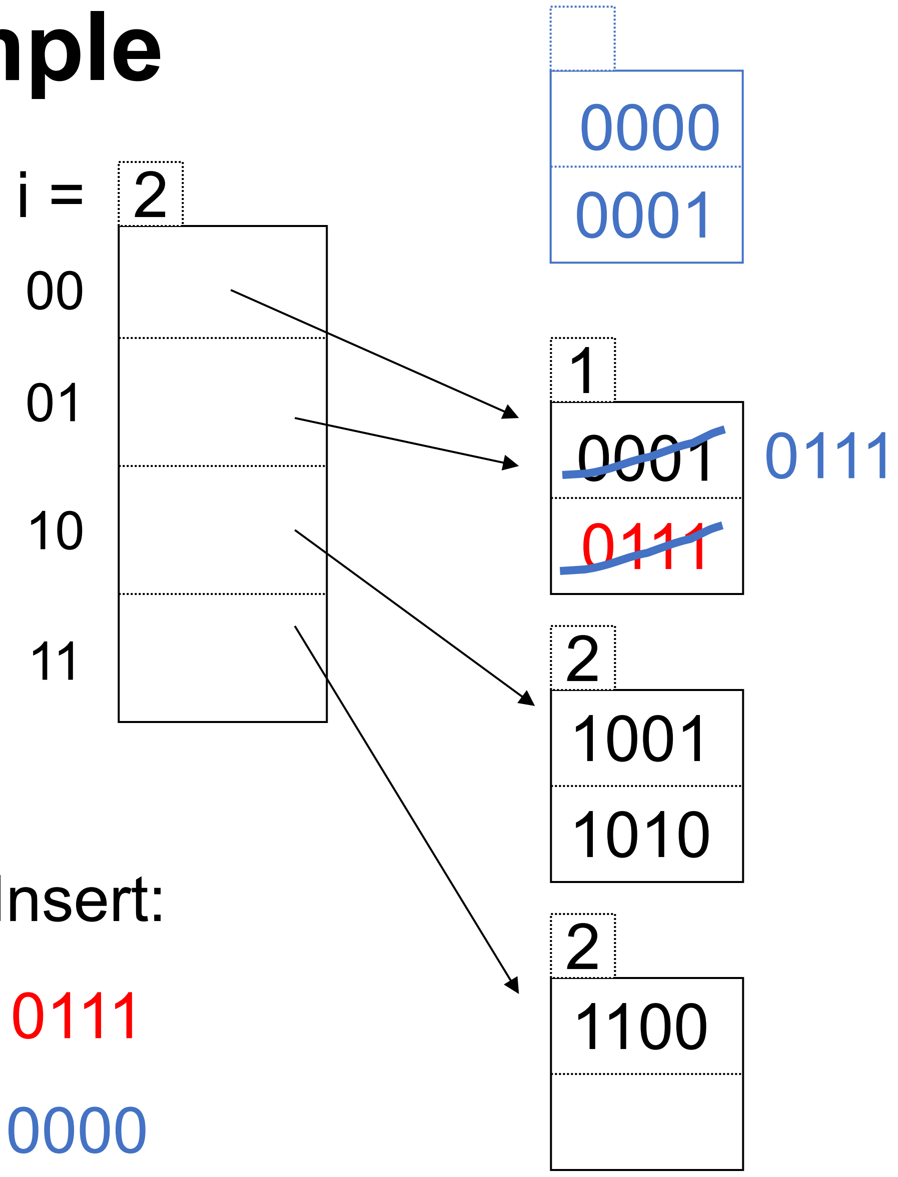


Insert:

0111

0000

Example

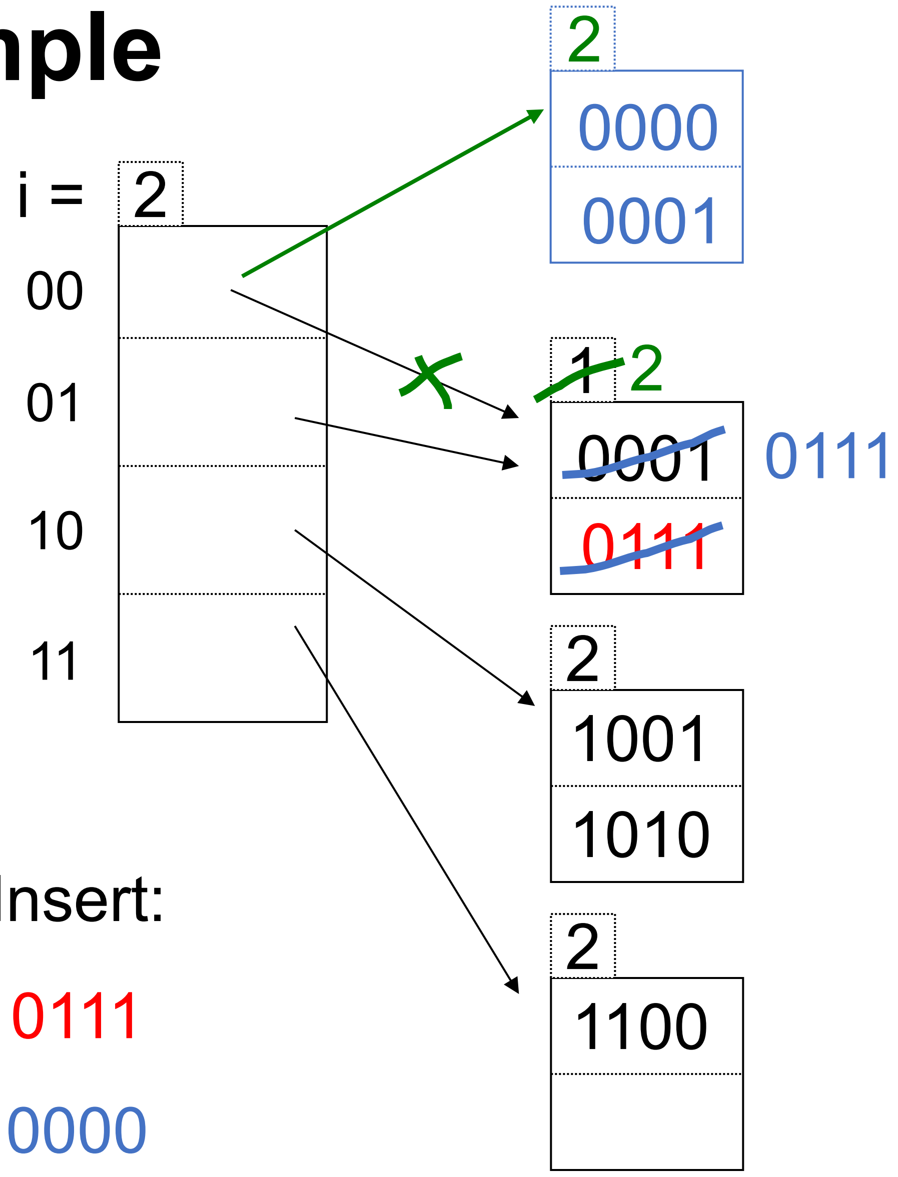


Insert:

0111

0000

Example

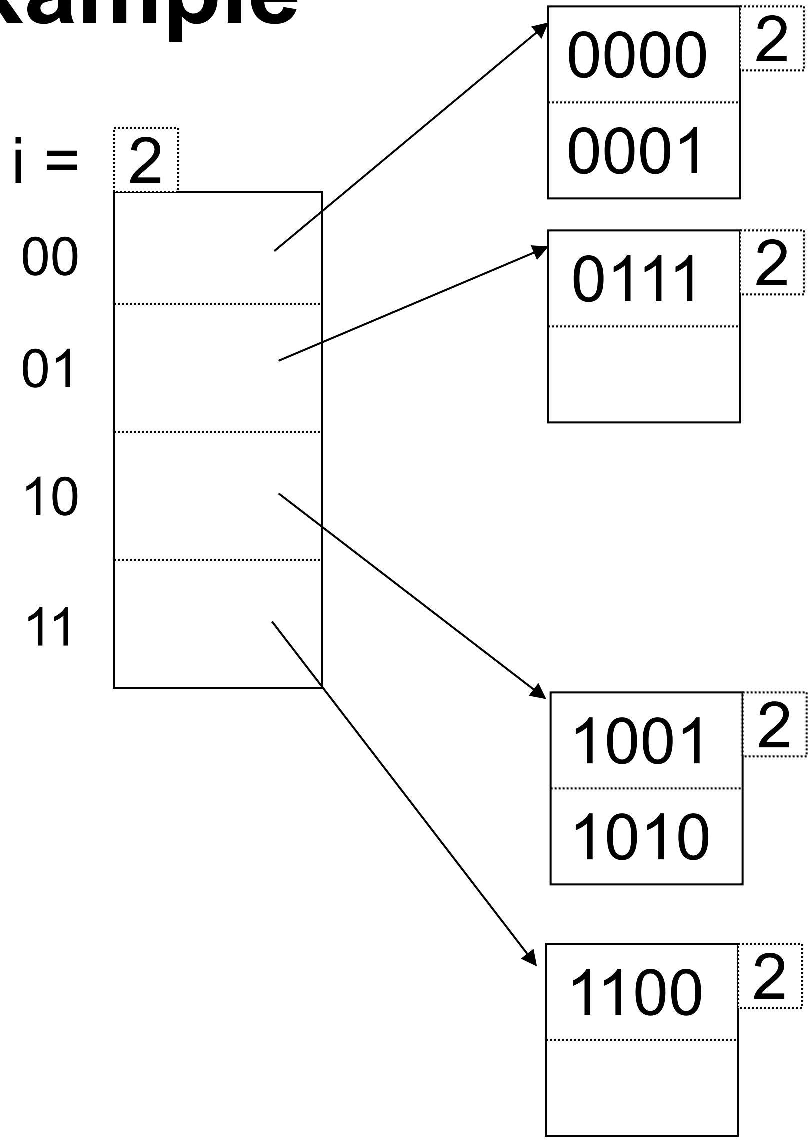


Insert:

0111

0000

Example



Note: still need chaining if values of h(K) repeat and fill a bucket

Some Types of Indexes

Conventional indexes

B-trees

Hash indexes

Multi-key indexing

Motivation

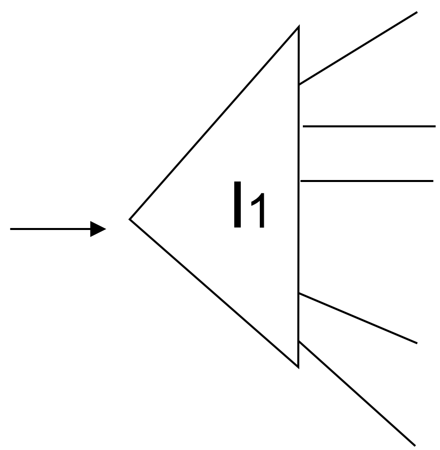
Example: find records where

DEPT = “Toy” AND SALARY > 50k

Strategy I:

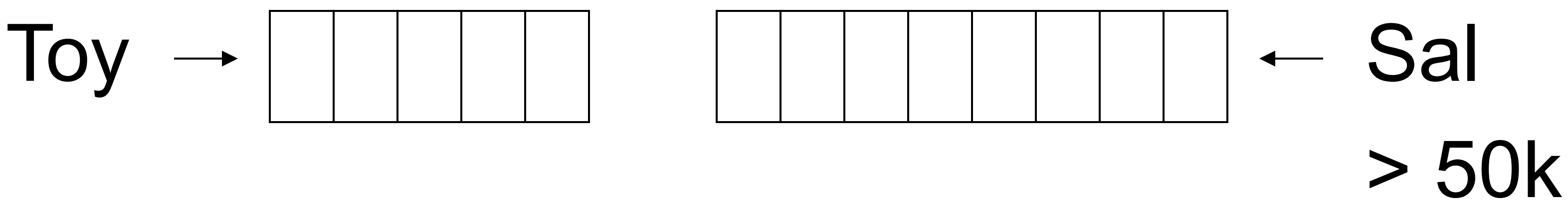
Use one index, say Dept.

Get all Dept = “Toy” records
and check their salary



Strategy II:

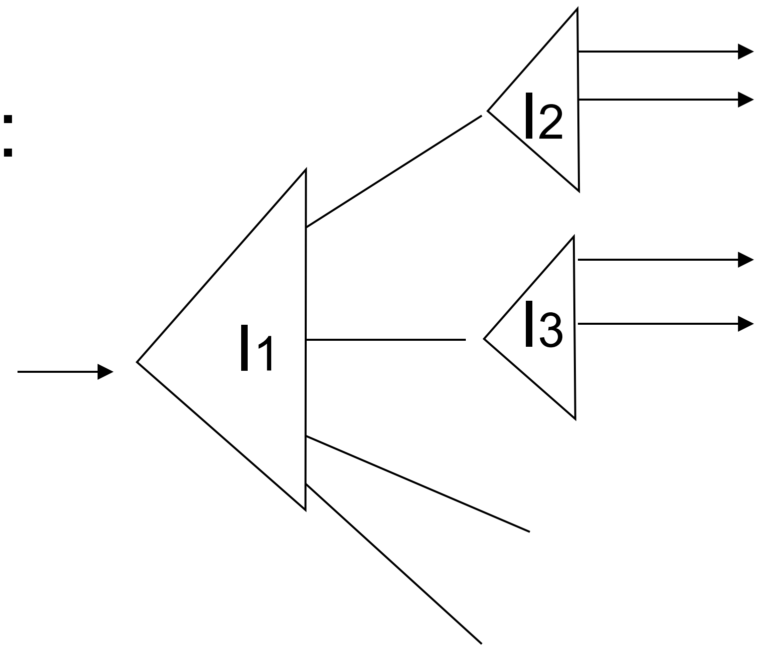
Use 2 indexes; manipulate pointers



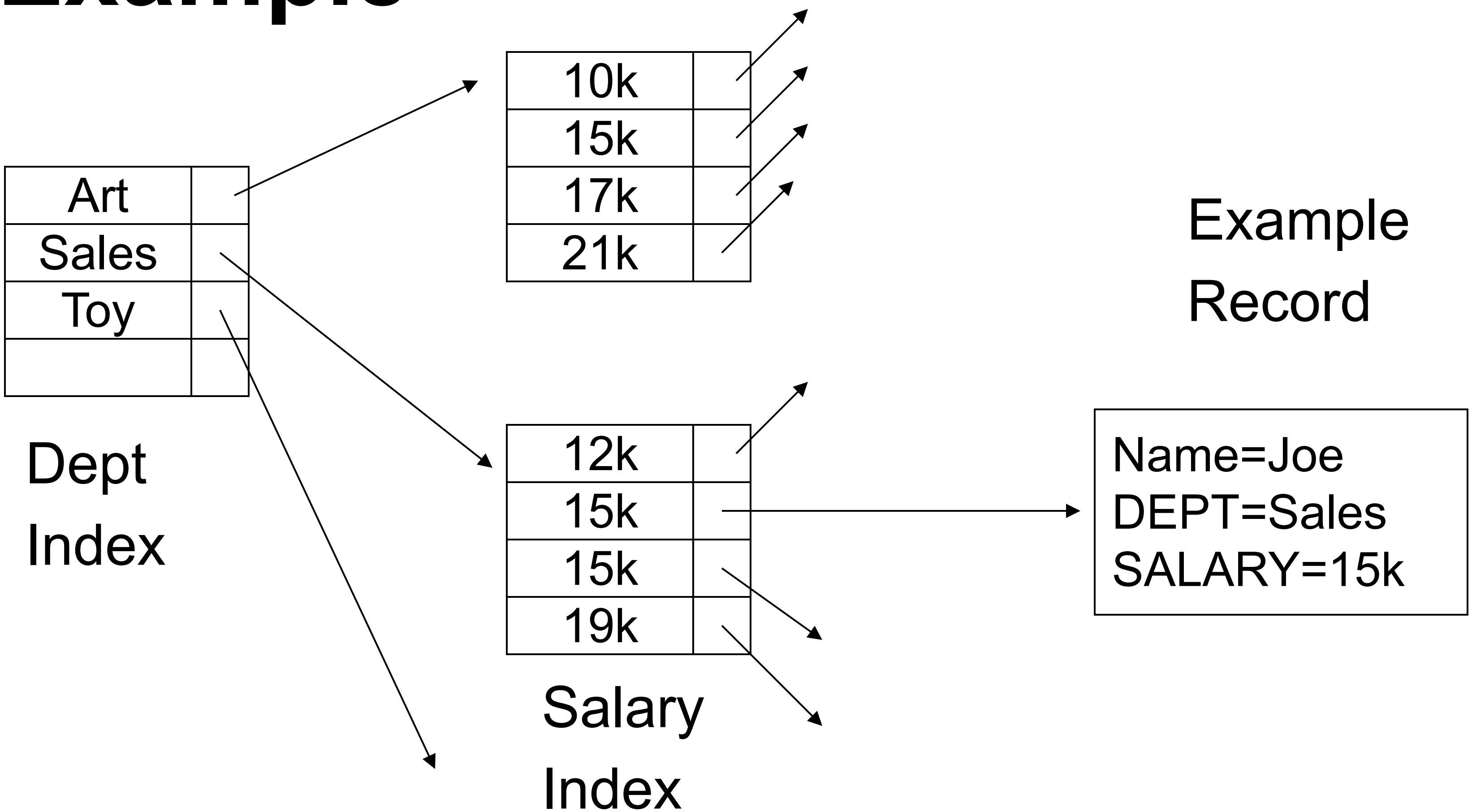
Strategy III:

Multi-key index

One idea:



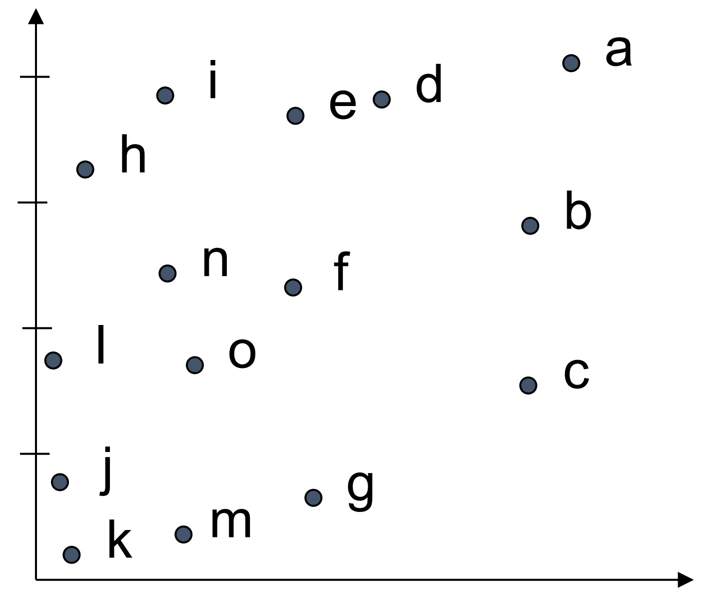
Example



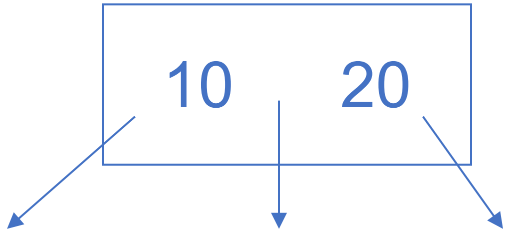
Art
Sales
Toy
Dept
Index
10k
15k
17k
21k
12k
15k
15k
19k
Salary
Index
Example
Record
Name=Joe
DEPT=Sales
SALARY=15k

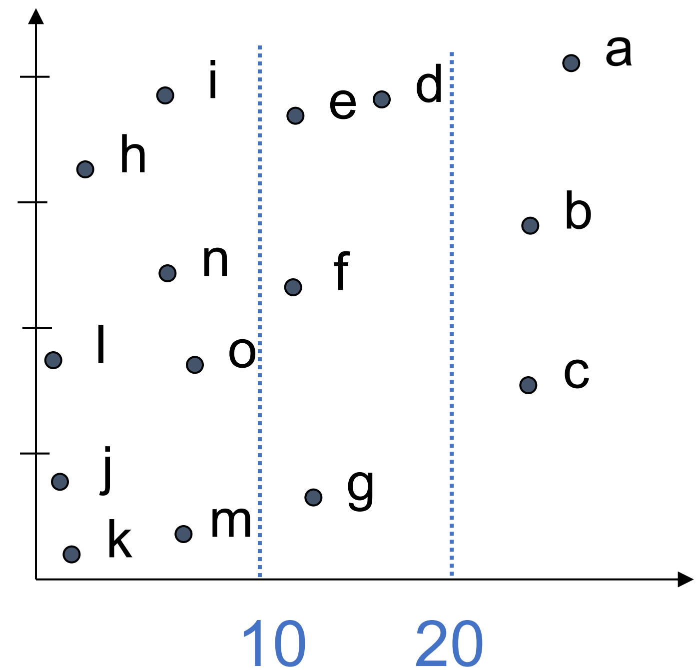
k-d Tree

Splits dimensions in any order to hold k-dimensional data

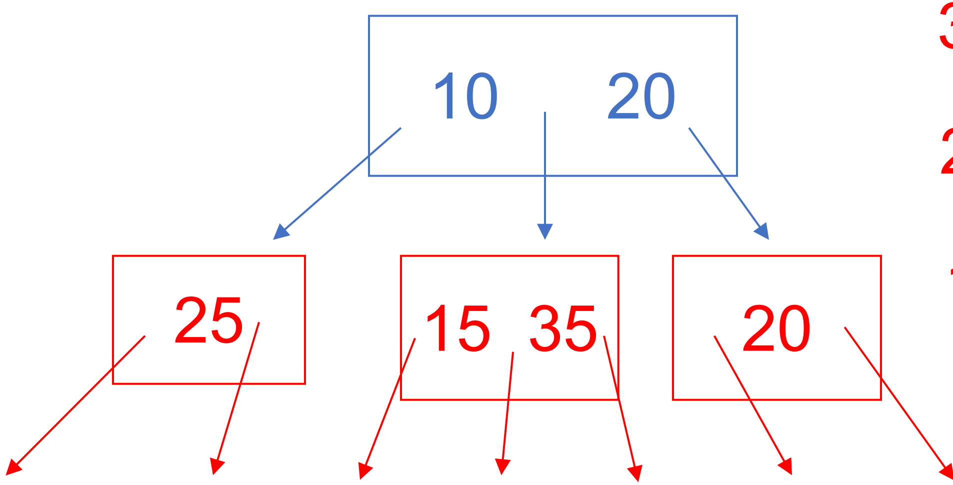


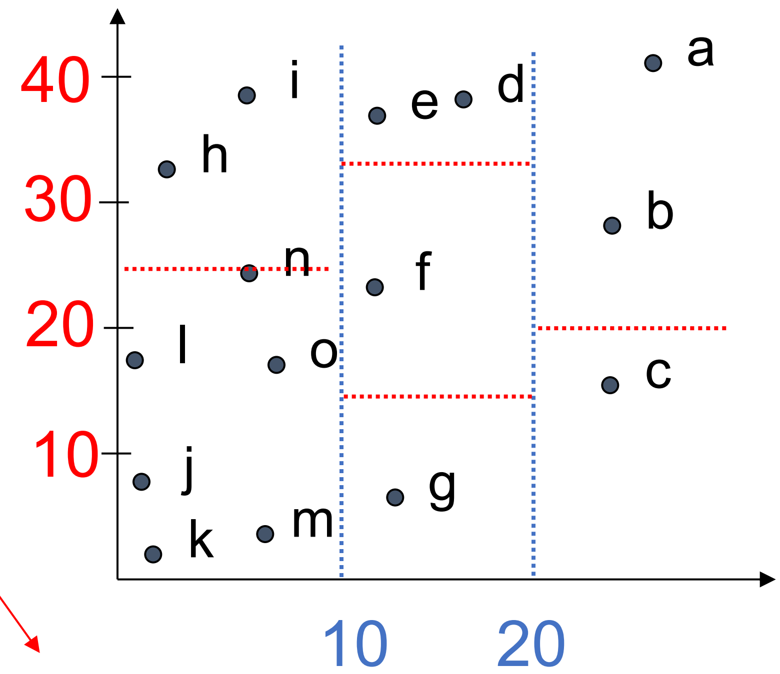
k-d Tree



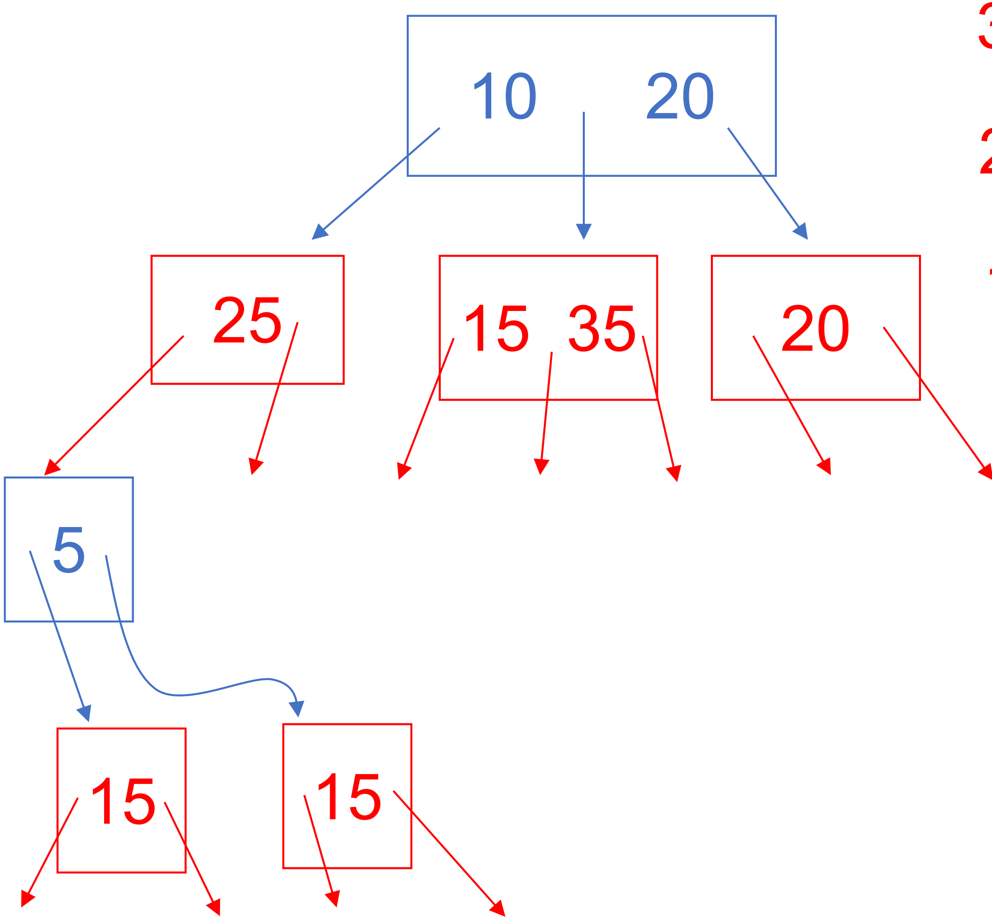


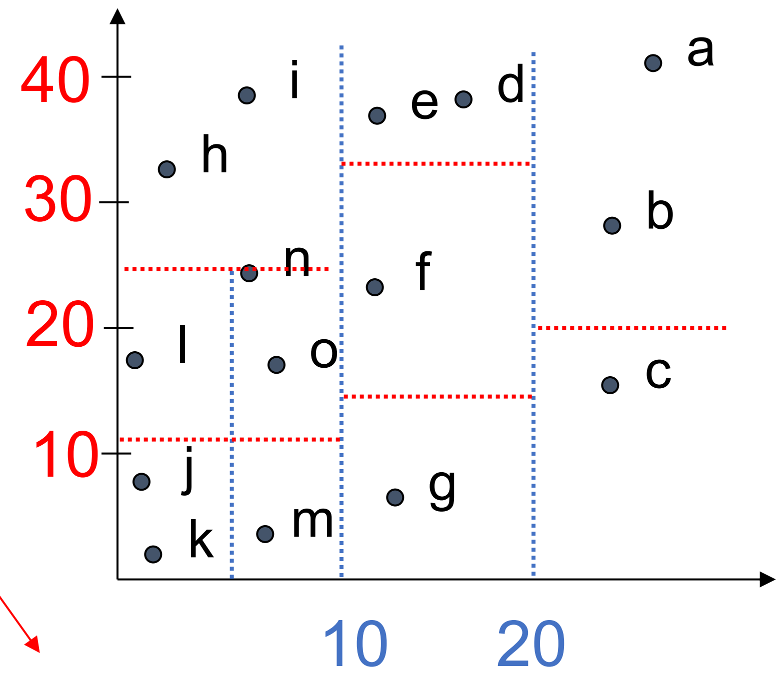
k-d Tree



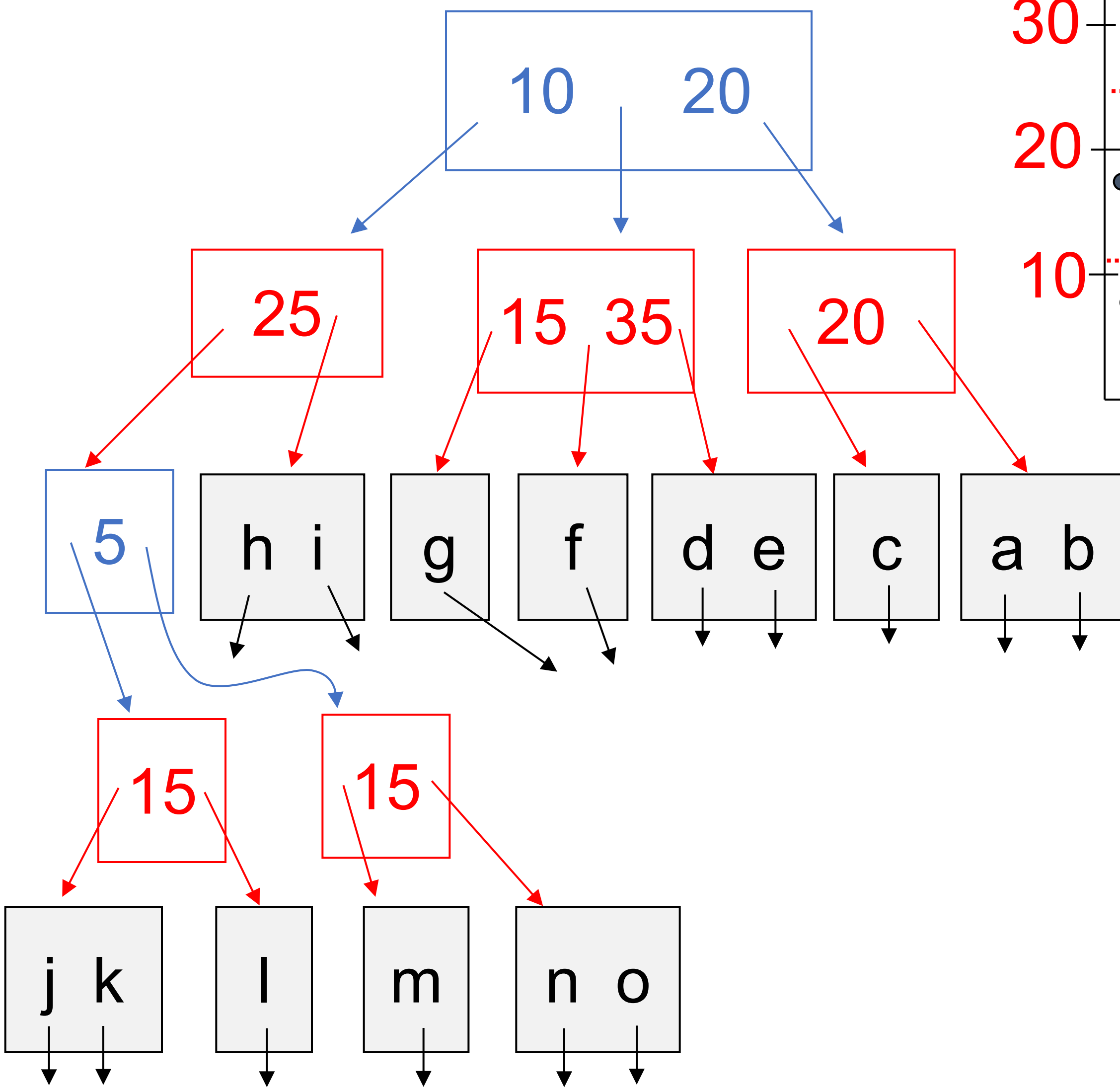


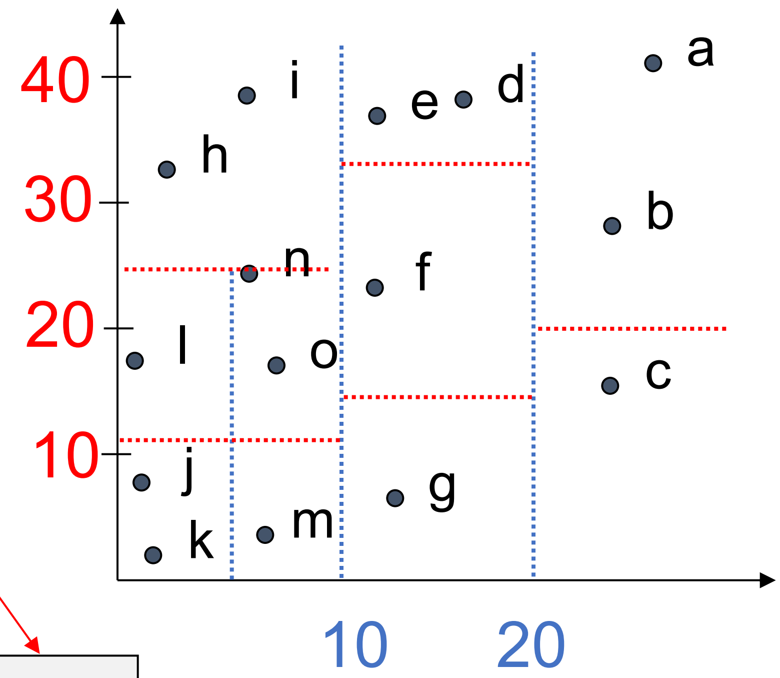
k-d Tree





k-d Tree





Efficient range queries in both dimensions

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?