

Data Storage & Indexes

Instructor: Matei Zaharia

Outline

C-Store (co-designing compute & storage)

Indexes

Outline

C-Store (co-designing compute & storage)

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

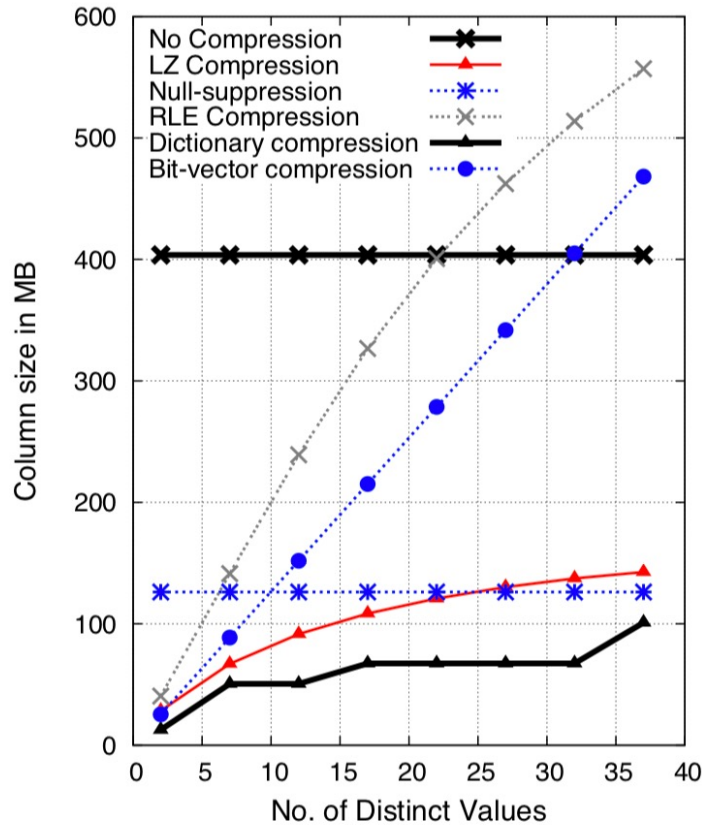
Encoding Type	Sorted?	1 value?	Pos. contig.?
RLE	yes	yes	yes
Bit-string	yes	yes	no
Null Supp.	no/yes	no	yes
Lempel-Ziv	no/yes	no	yes
Dictionary	no/yes	no	yes
Uncompressed	no/yes	no	no/yes

Using the Block API

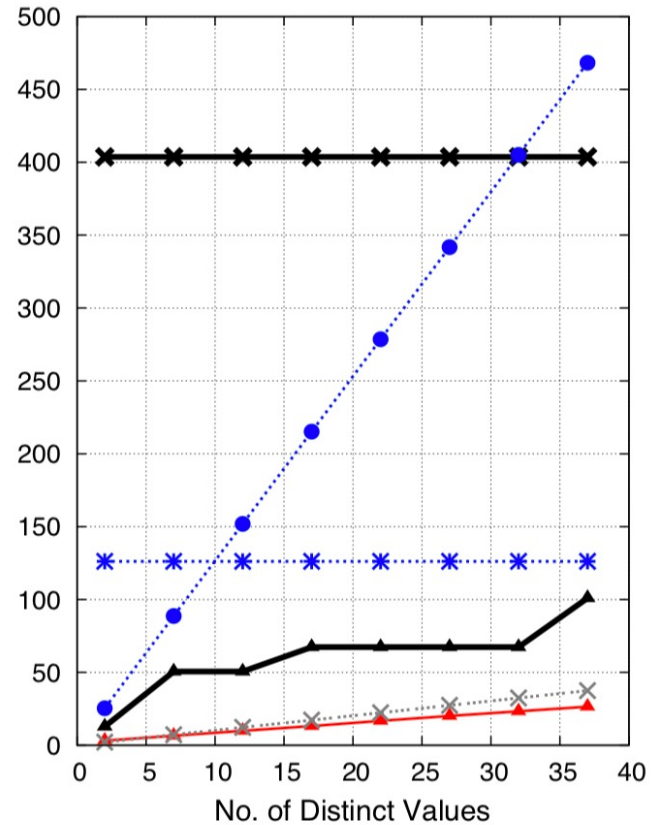
```
COUNT(COLUMN  $c_1$ )  
   $b$  = GET NEXT COMPRESSED BLOCK FROM  $c_1$   
  WHILE  $b$  IS NOT NULL  
    IF  $b$ .ISONEVALUE()  
       $x$  = FETCH CURRENT COUNT FOR  $b$ .GETSTARTVAL()  
       $x$  =  $x$  +  $b$ .GETSIZE()  
    ELSE  
       $a$  =  $b$ .ASARRAY()  
      FOR EACH ELEMENT  $i$  IN  $a$   
         $x$  = FETCH CURRENT COUNT FOR  $i$   
         $x$  =  $x$  + 1  
   $b$  = GET NEXT COMPRESSED BLOCK FROM  $c_1$ 
```

Figure 2: Pseudocode for Simple Count Aggregation

Data Size with Each Scheme



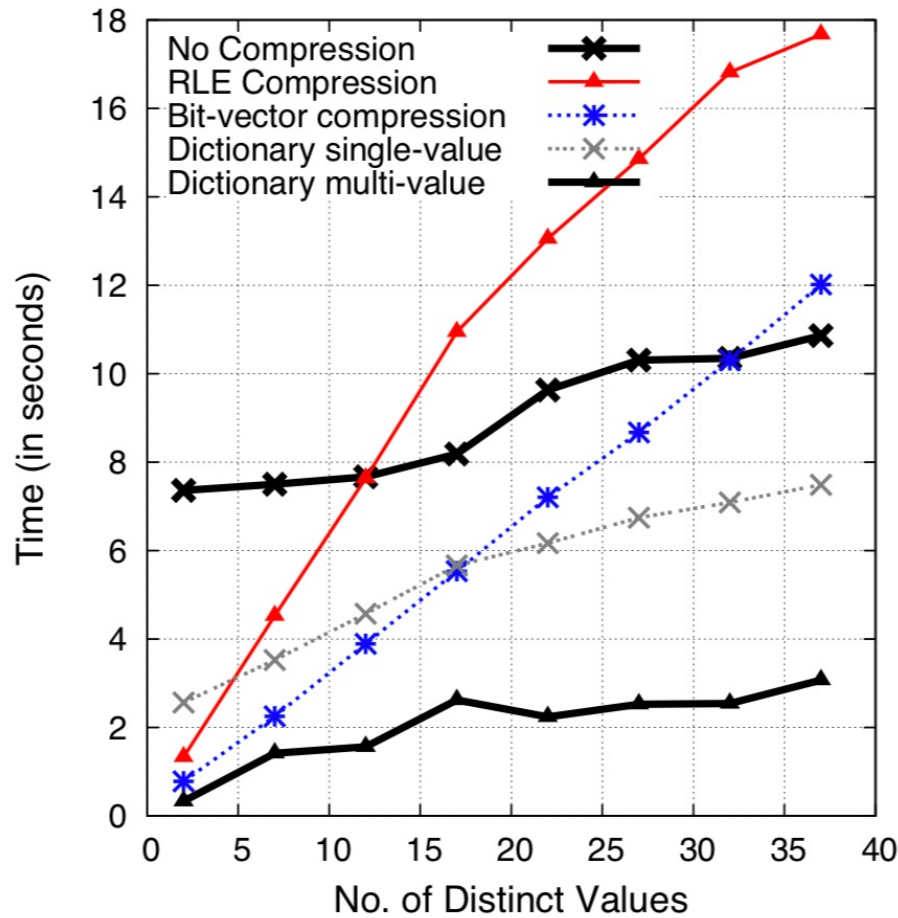
(a) Sorted runs of length 50



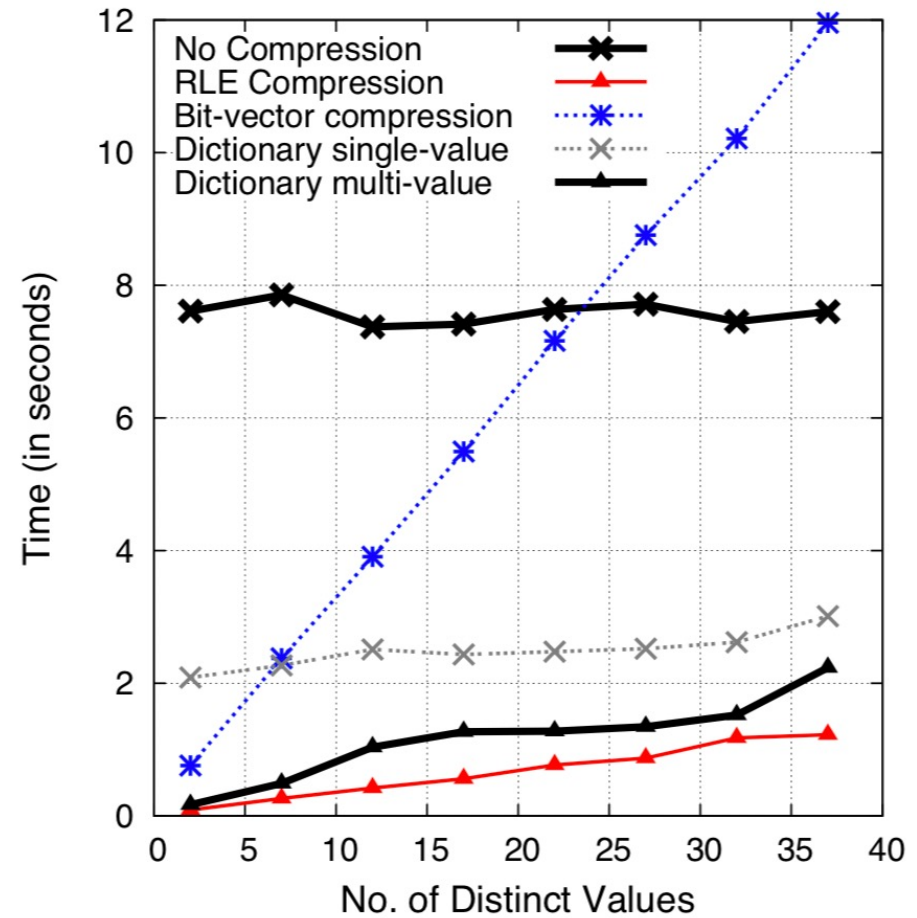
(a) Sorted runs of length 1000

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

Performance: Compressed Eval



(a) Sorted runs of length 50



(b) Sorted runs of length 1000

Fig

How would the results change on SSDs?

ata

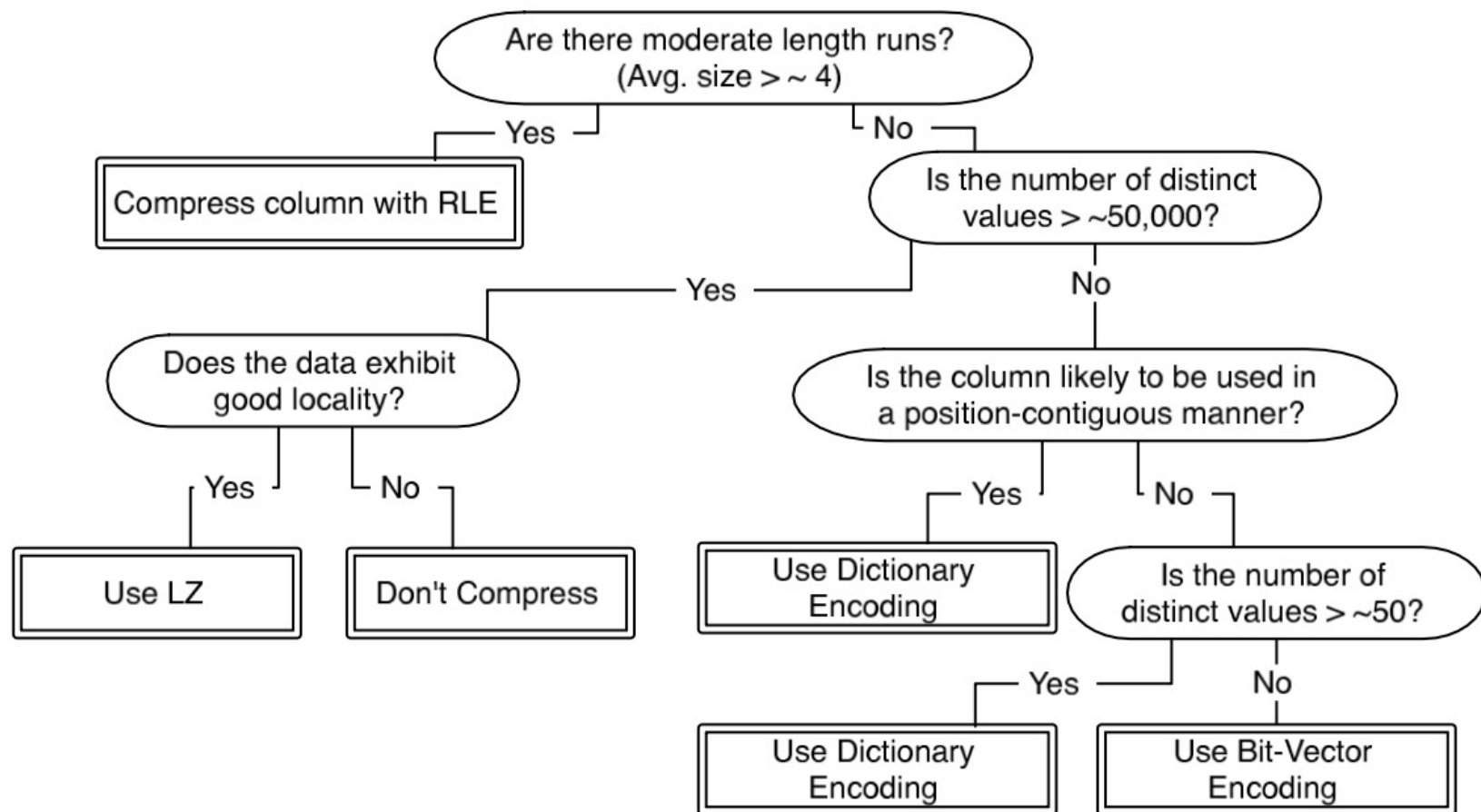


Figure 10: Decision tree summarizing our results regarding the proper selection of compression scheme.

Outline

C-Store (co-designing compute & storage)

Indexes

Key Operations on an Index

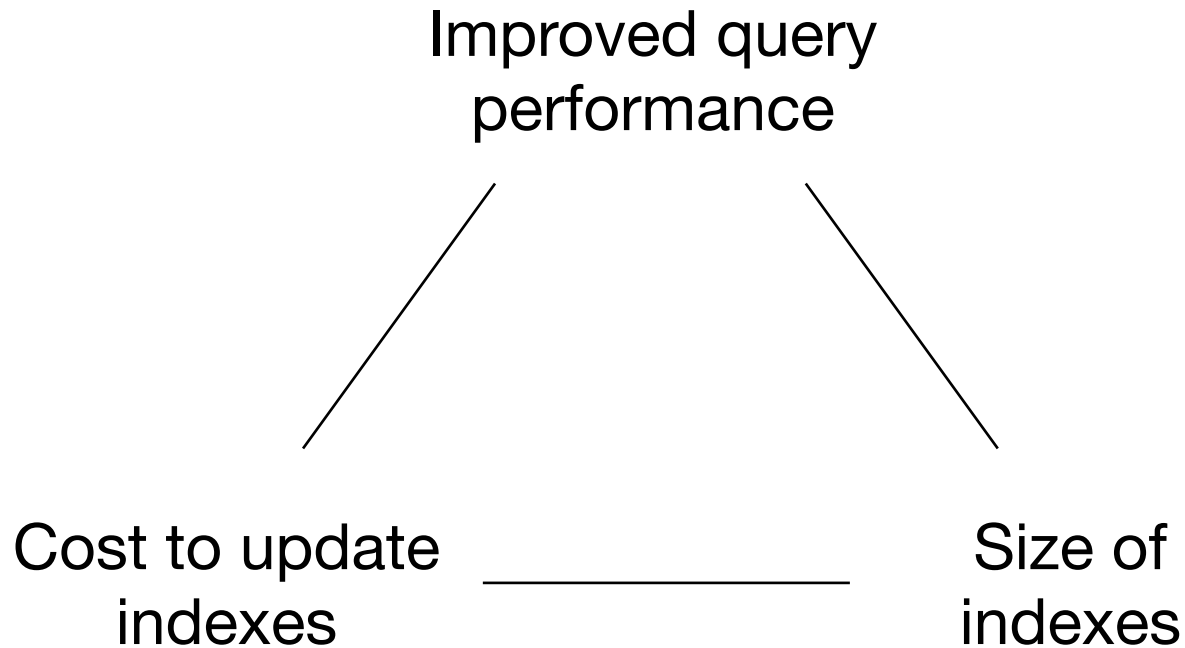
Find all records with a given **value** for a key

- » 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	

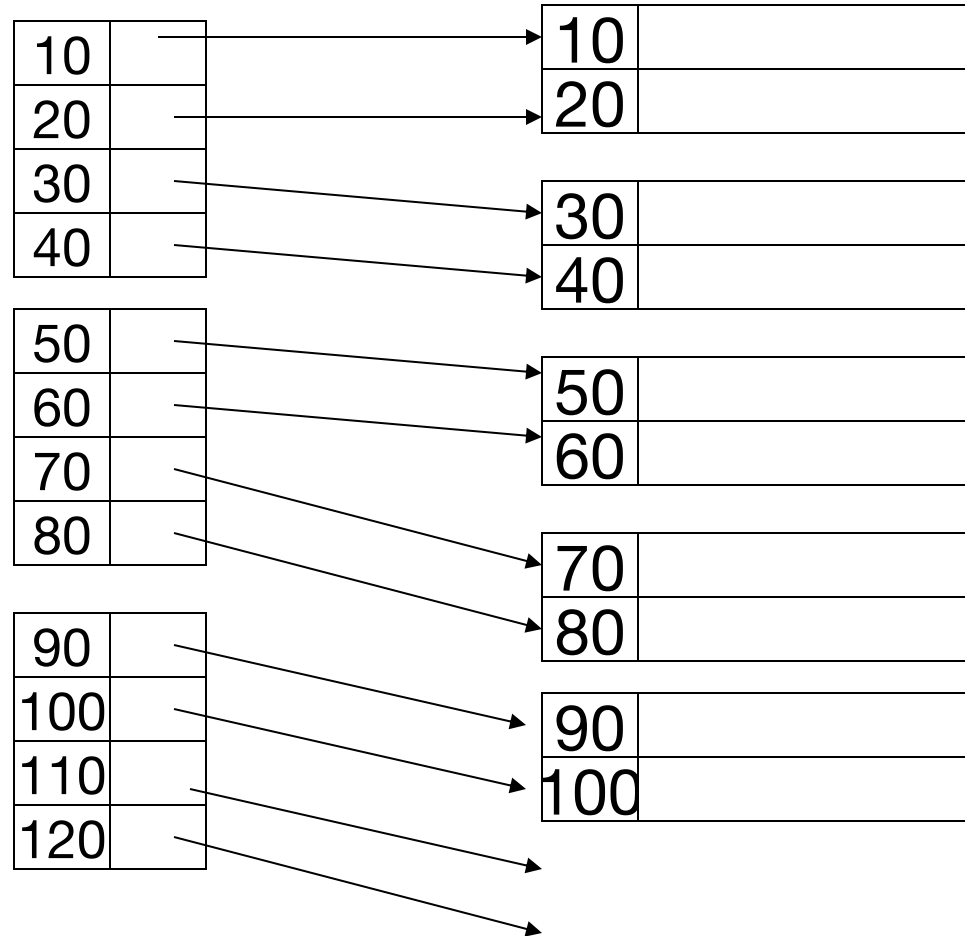
50	
60	

70	
80	

90	
100	

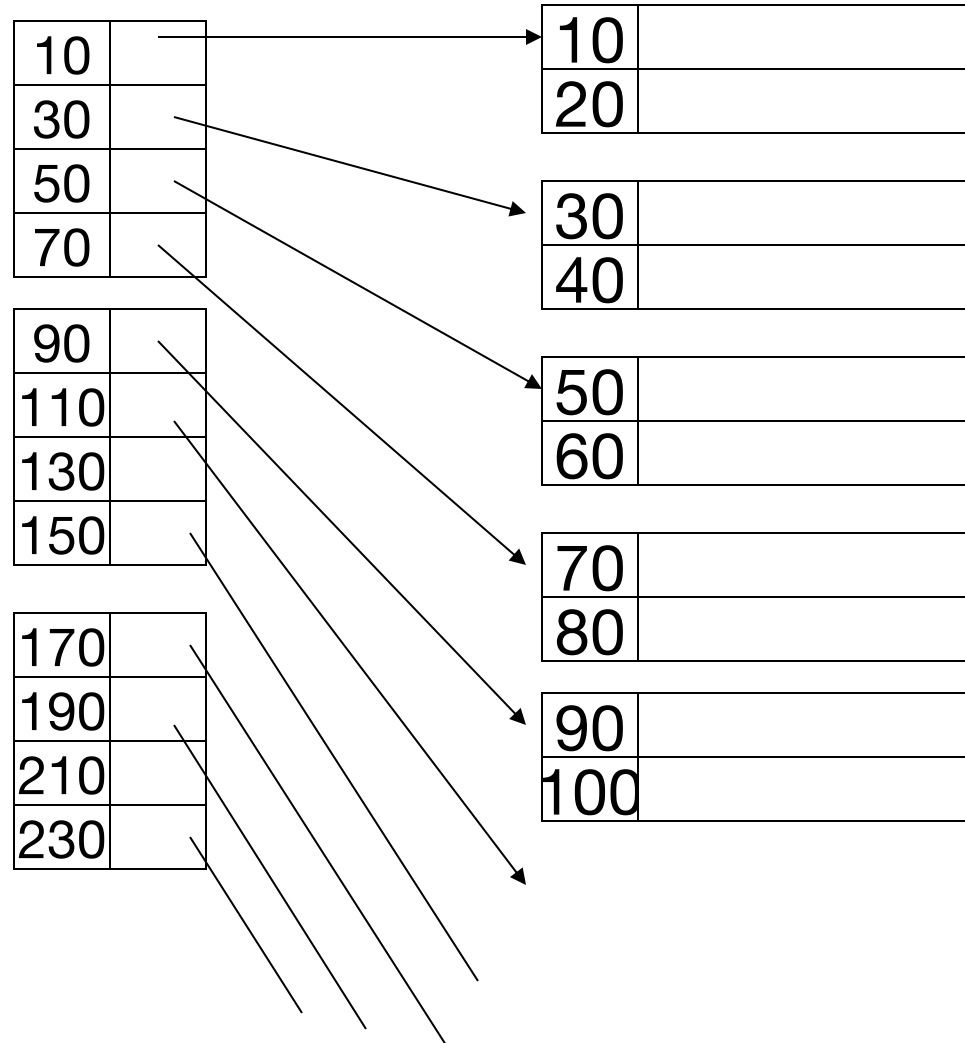
Dense Index

Sequential File



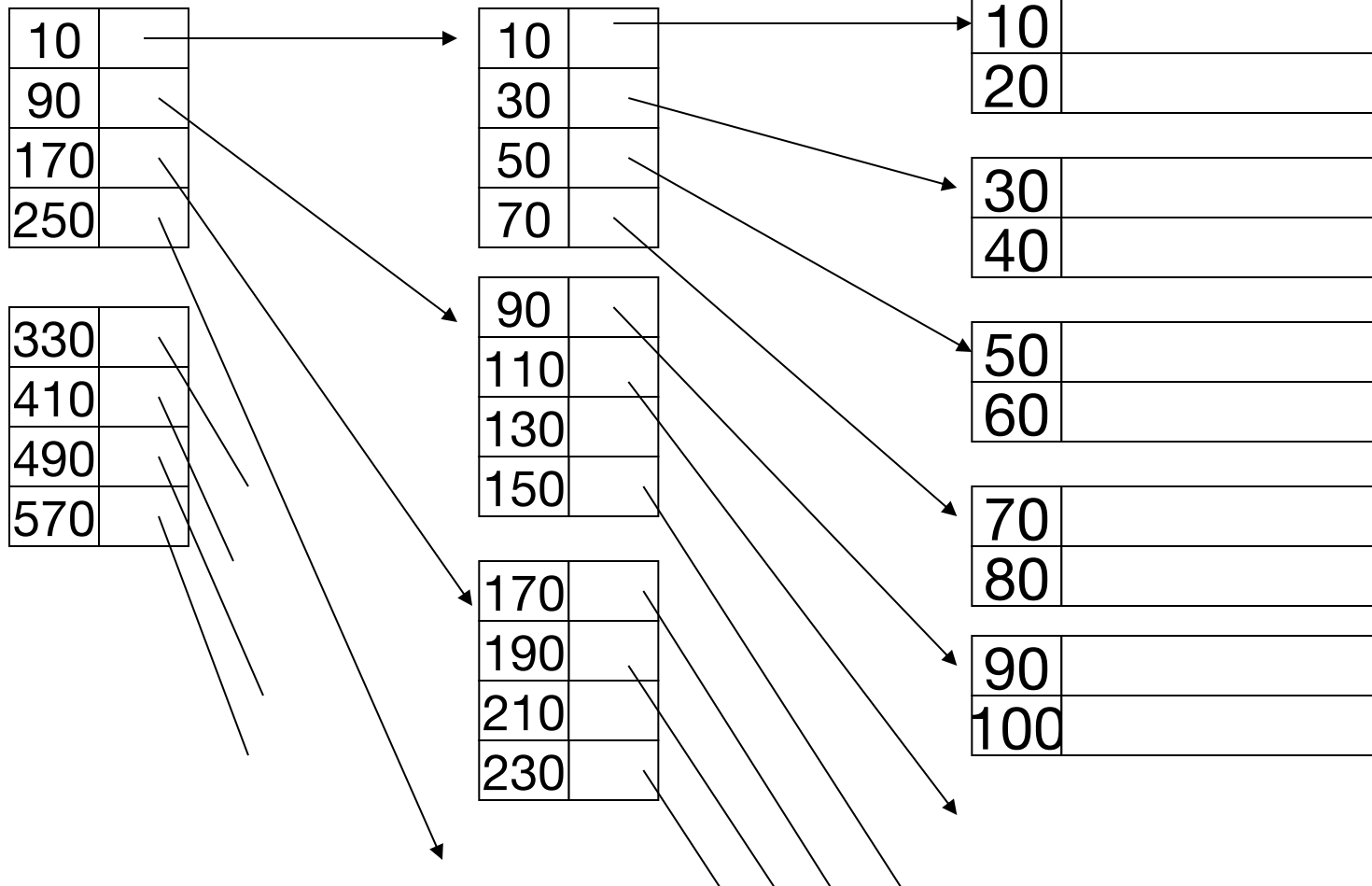
Sparse Index

Sequential File



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 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

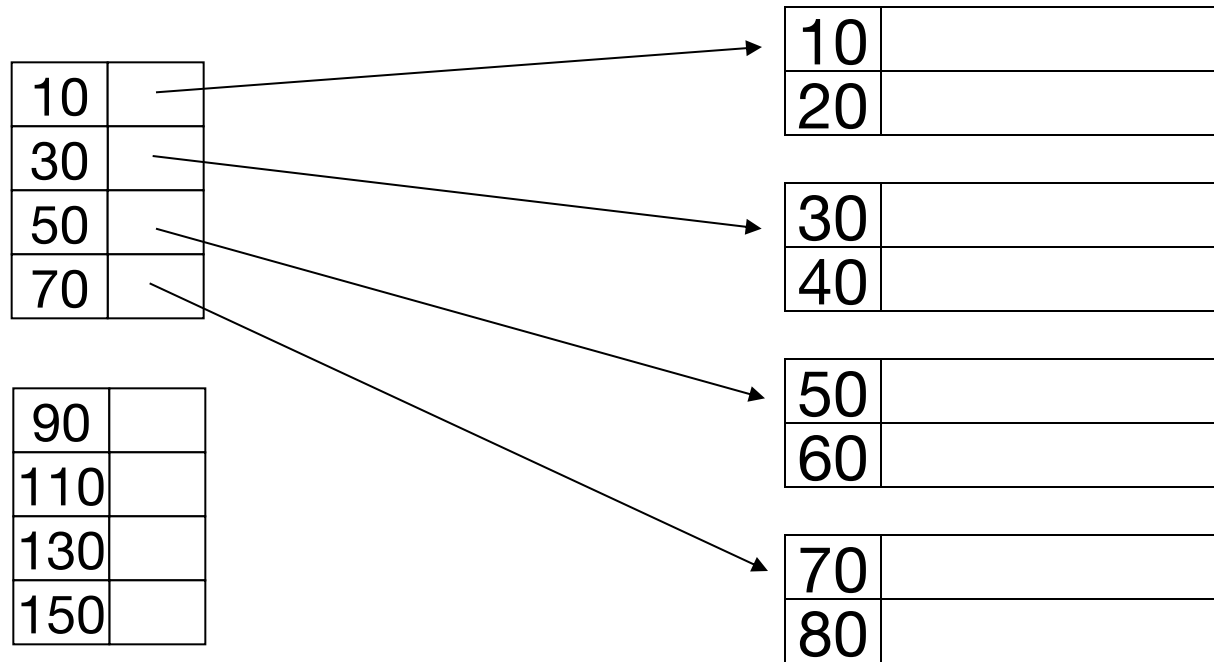
Multi-level index

Handling Duplicate Keys

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

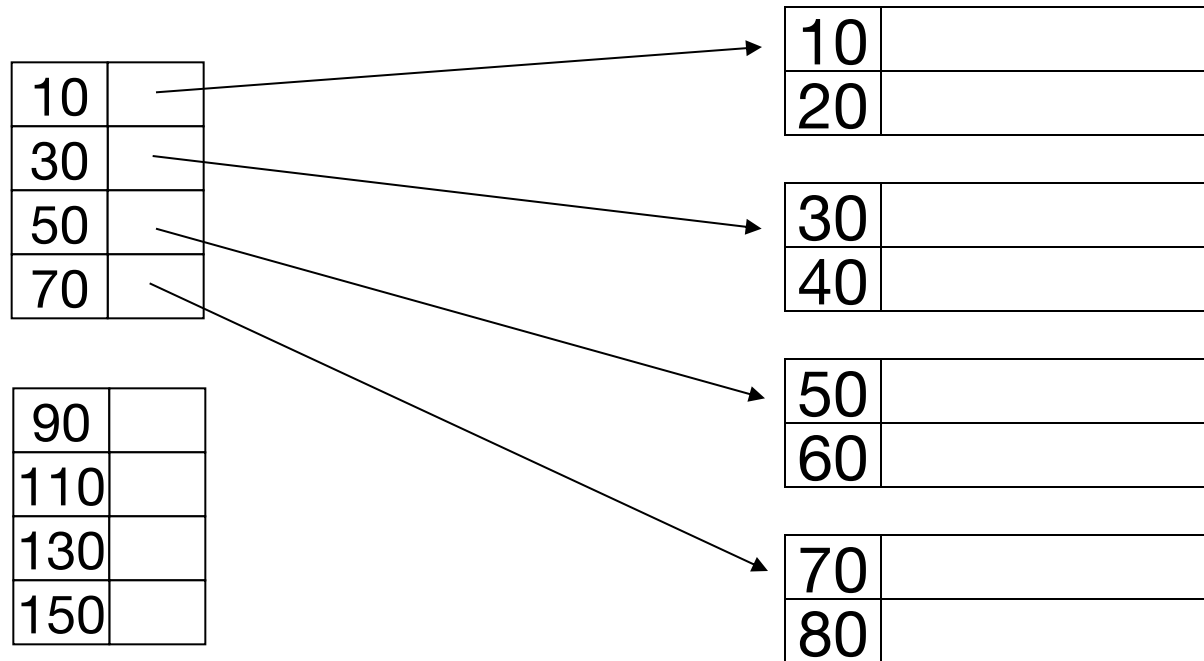
For a secondary index, need to point to a list of records since they can be anywhere

Deletion: Sparse Index



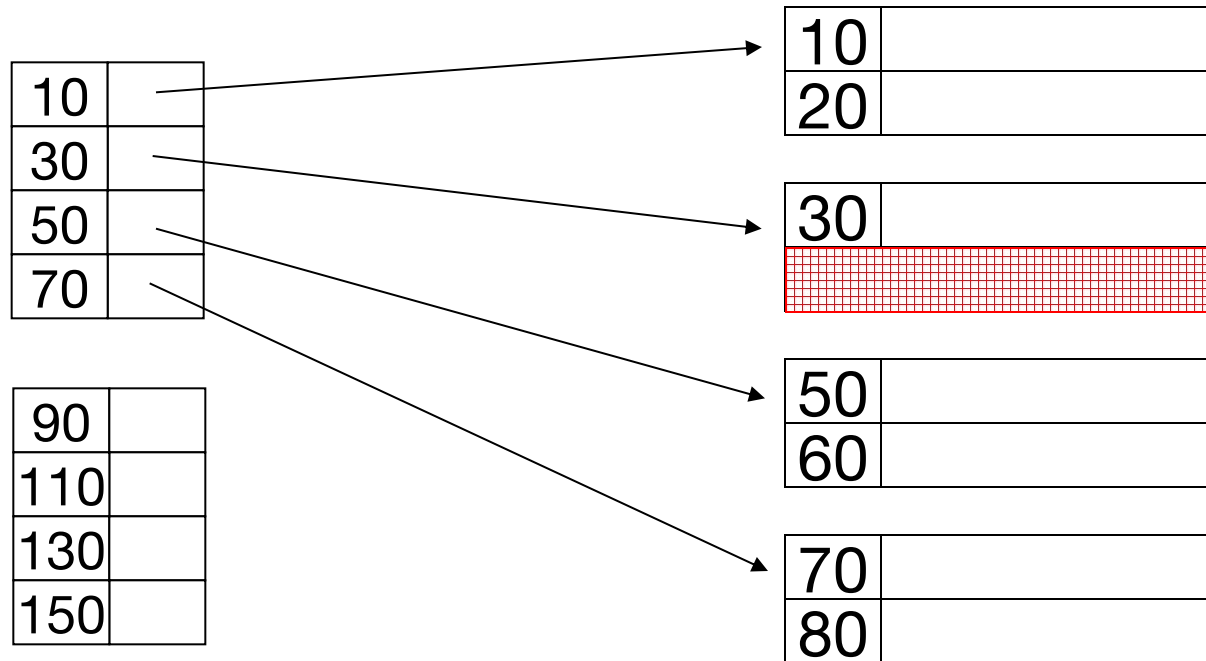
Deletion: Sparse Index

– delete record 40



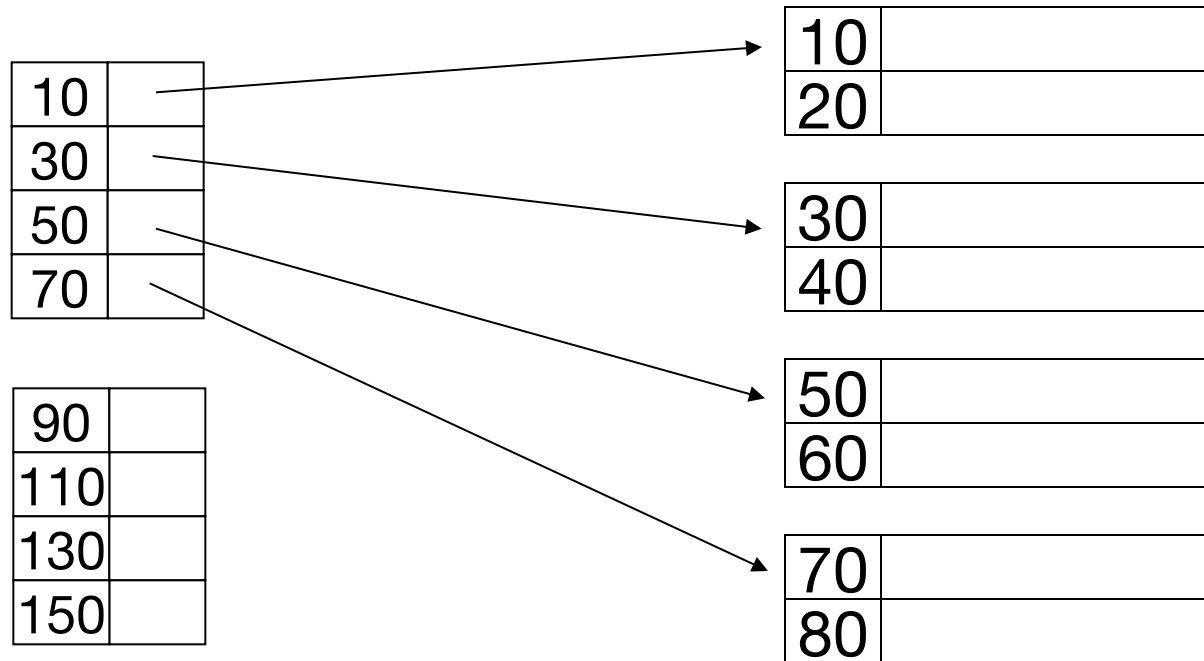
Deletion: Sparse Index

– delete record 40



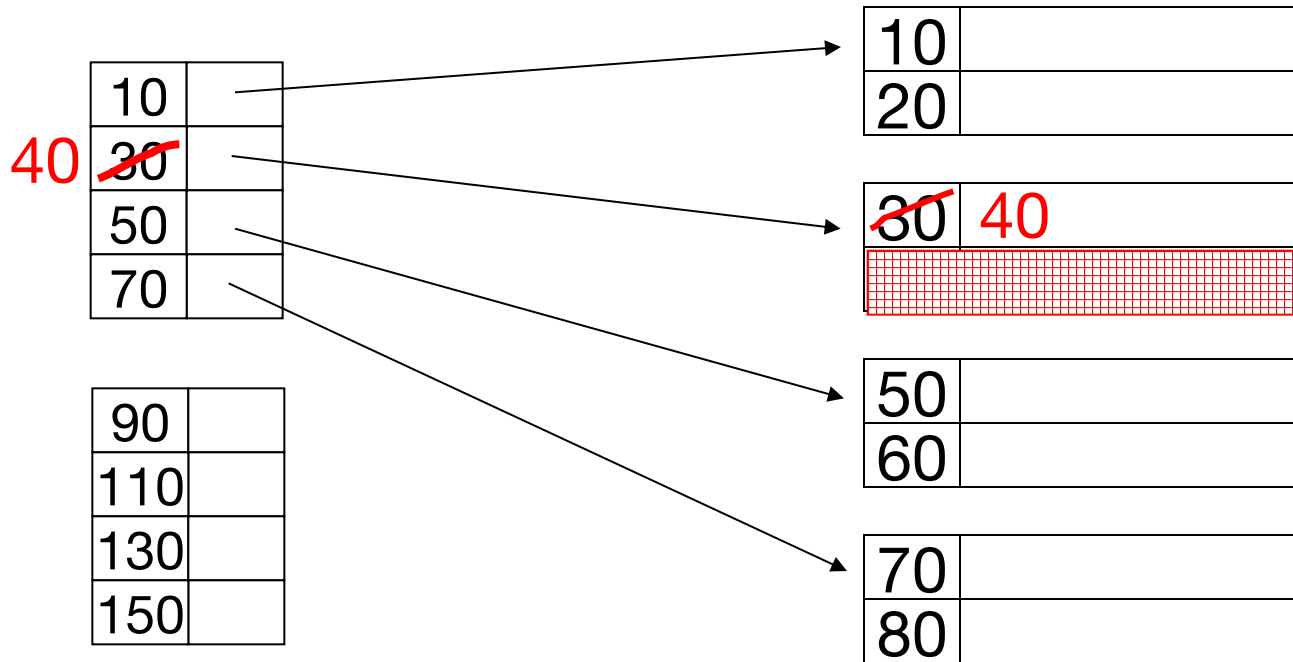
Deletion: Sparse Index

– delete record 30



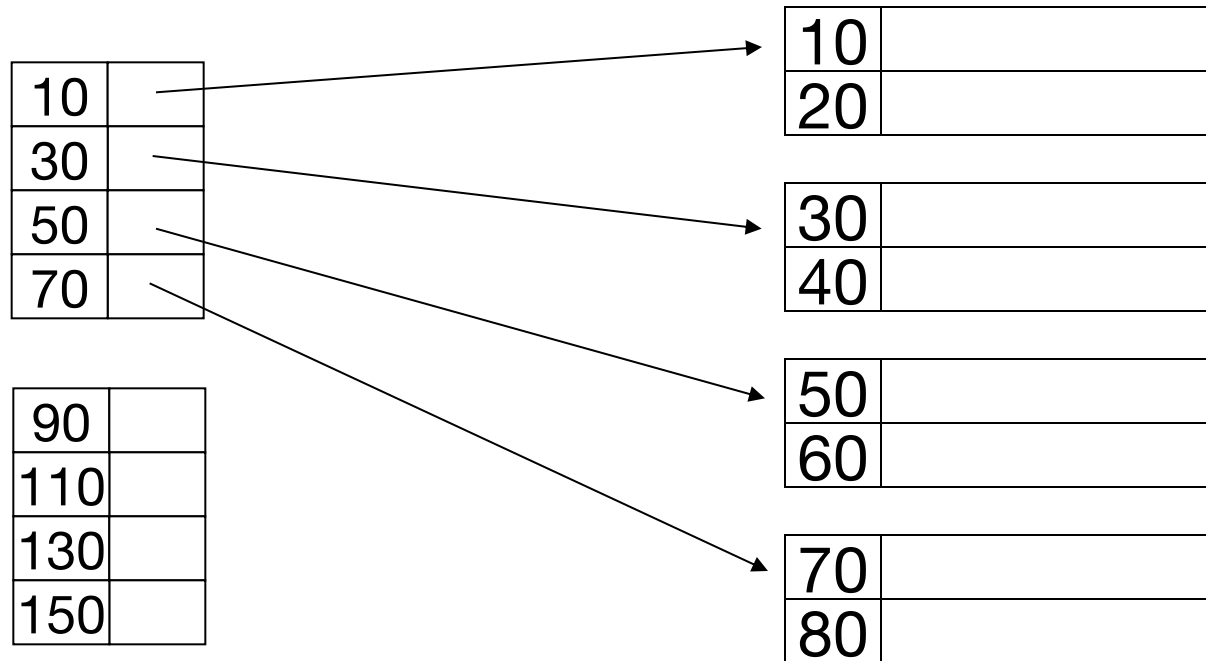
Deletion: Sparse Index

– delete record 30



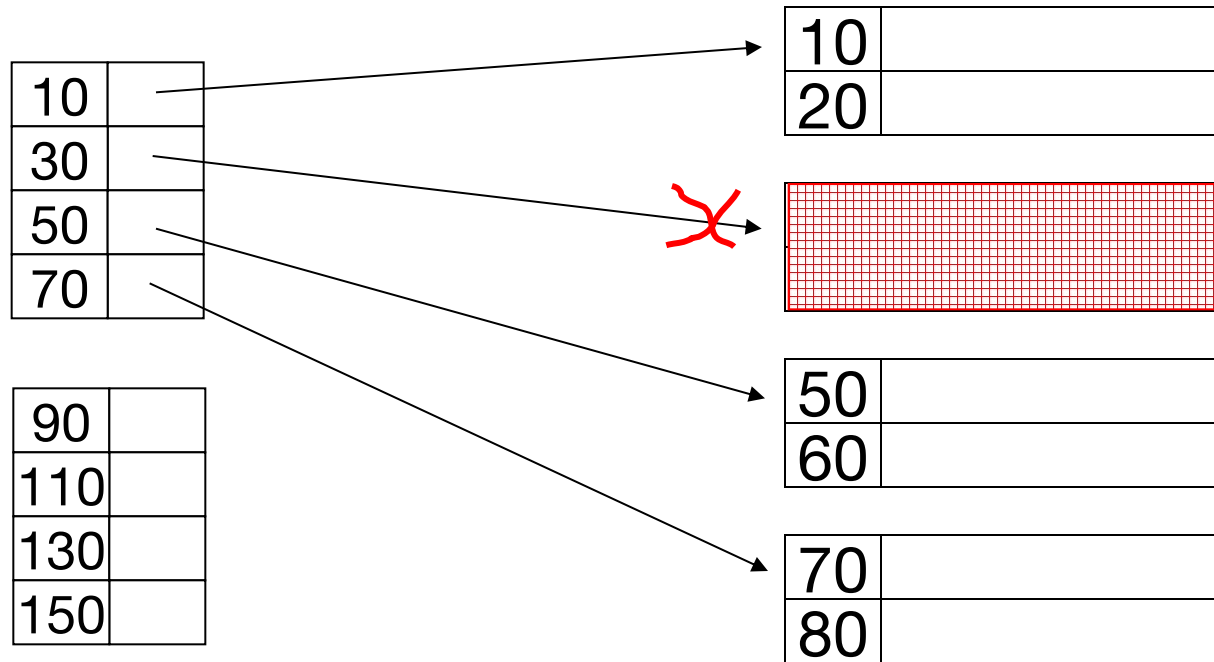
Deletion: Sparse Index

– delete records 30 & 40



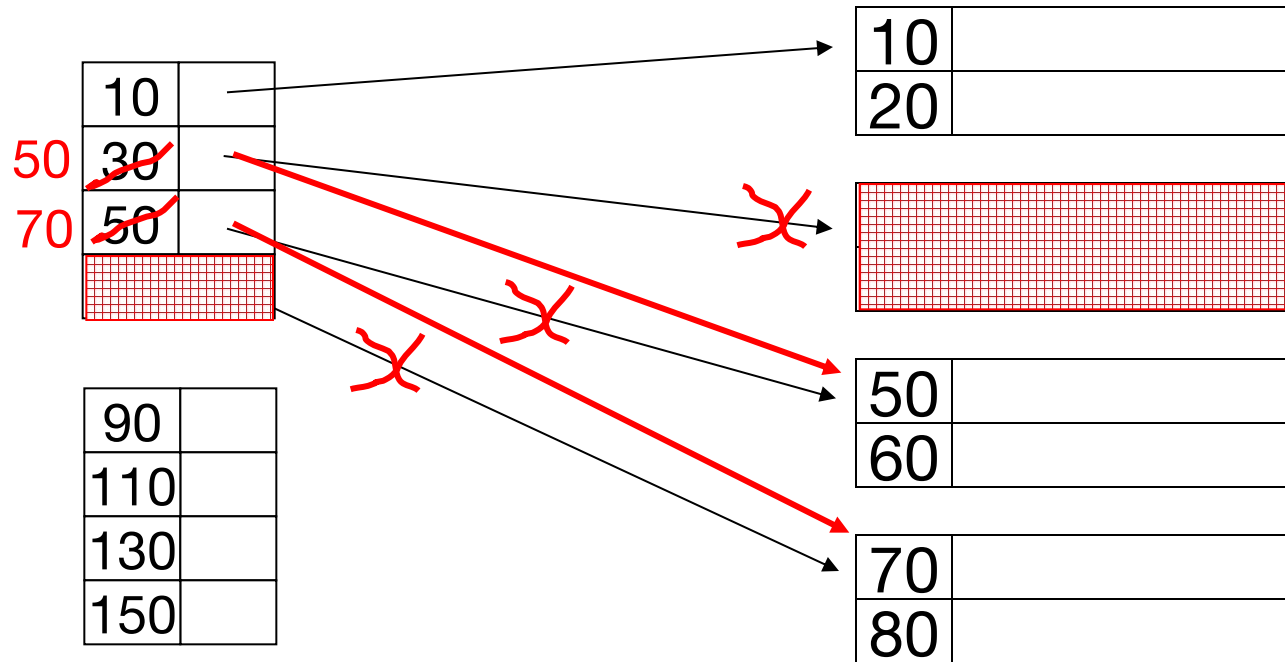
Deletion: Sparse Index

– delete records 30 & 40

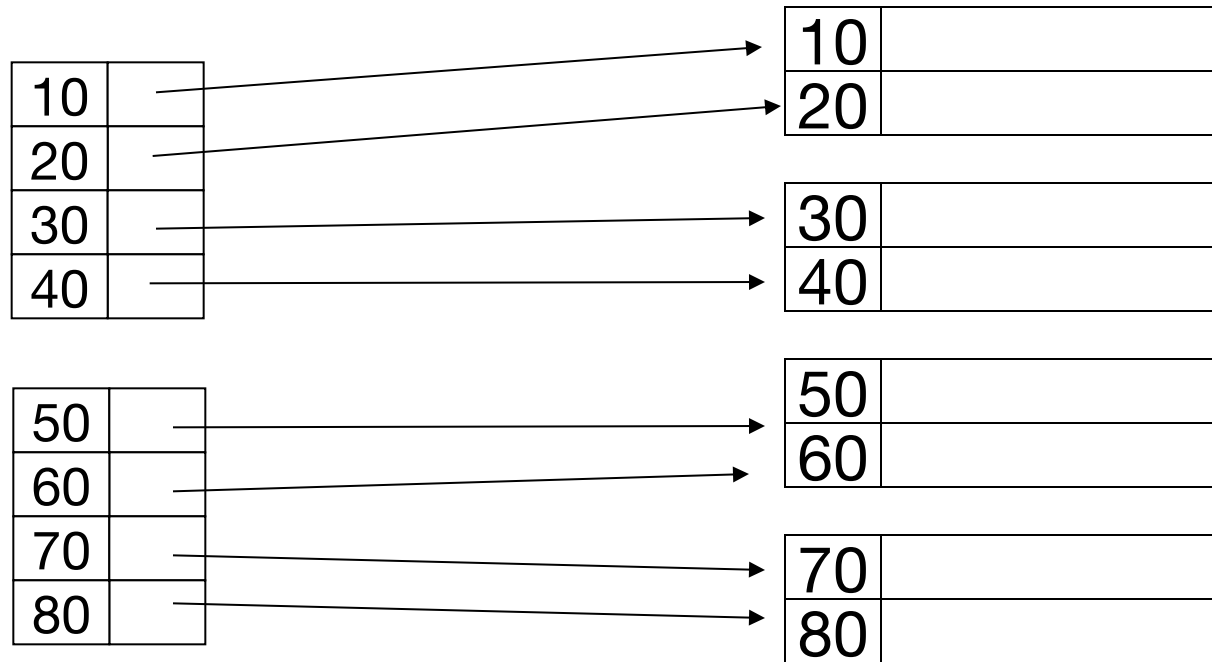


Deletion: Sparse Index

– delete records 30 & 40

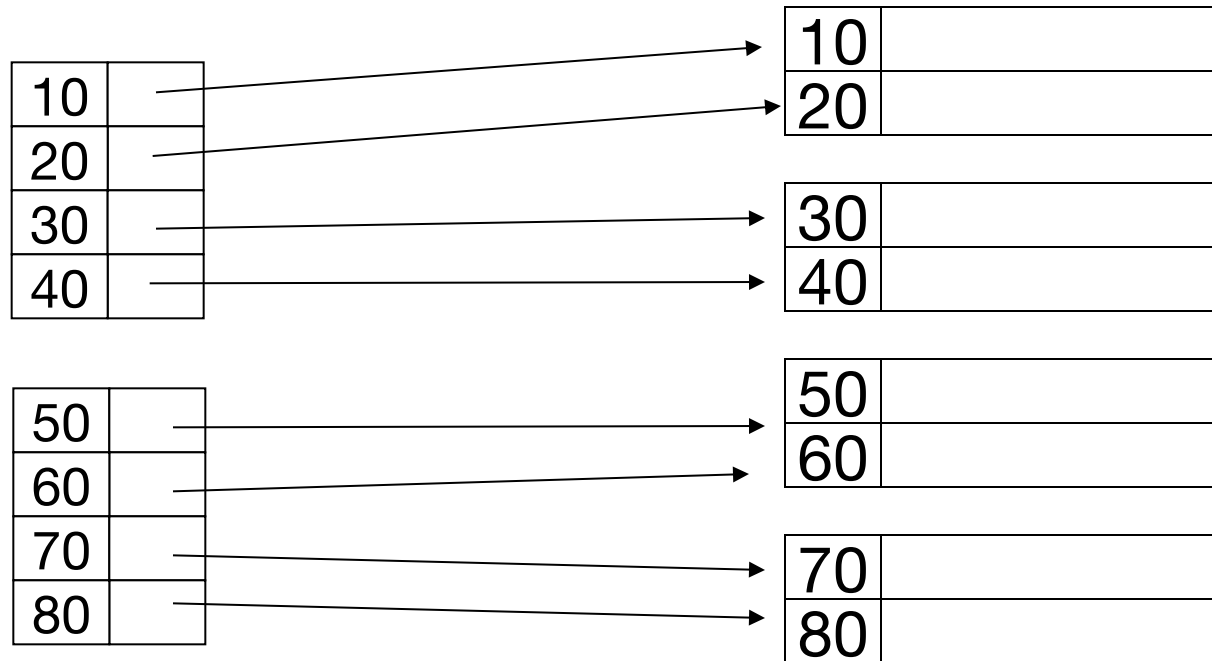


Deletion: Dense Index



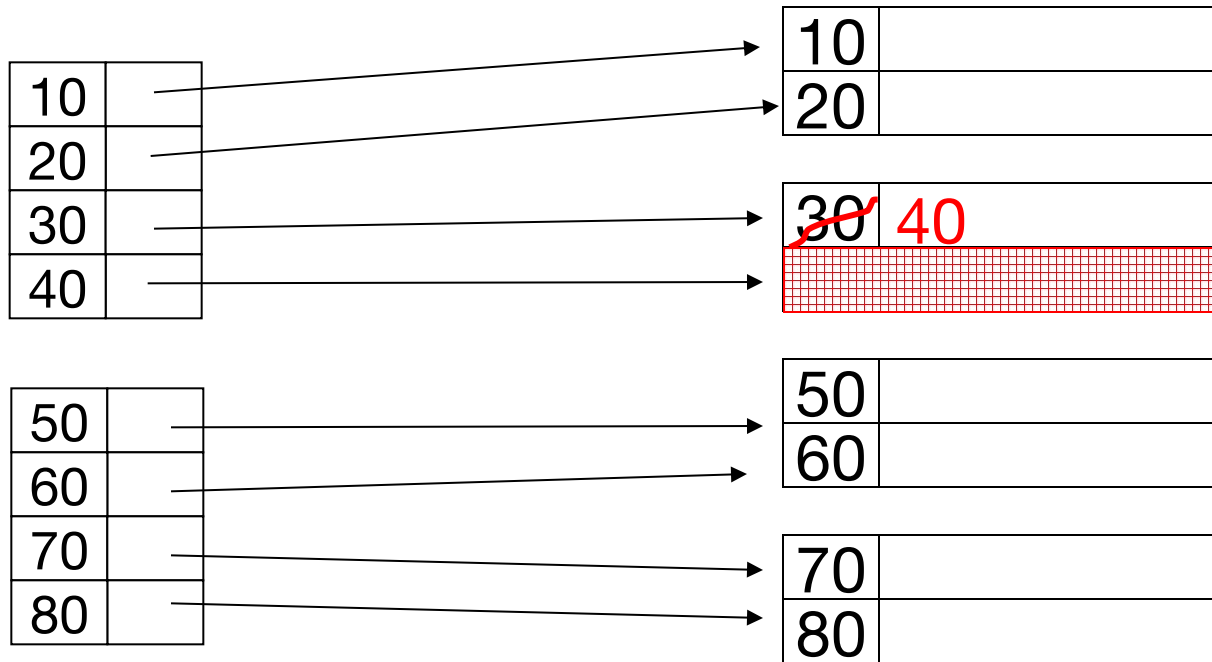
Deletion: Dense Index

– delete record 30



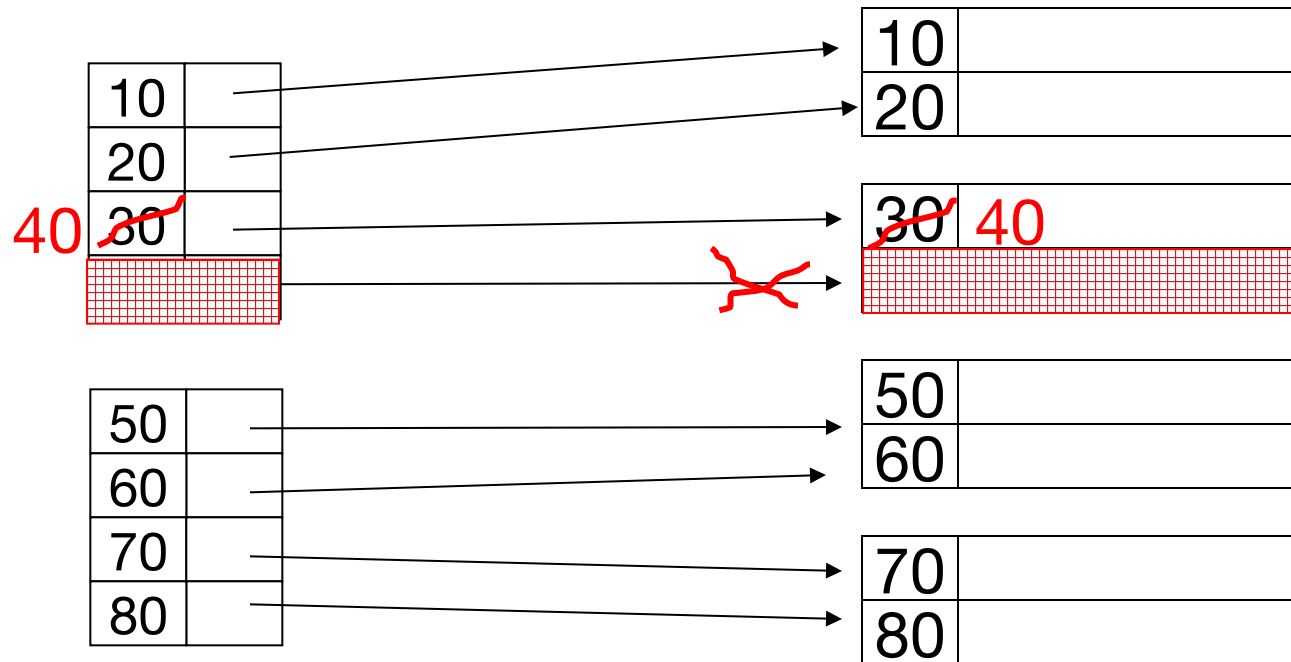
Deletion: Dense Index

– delete record 30



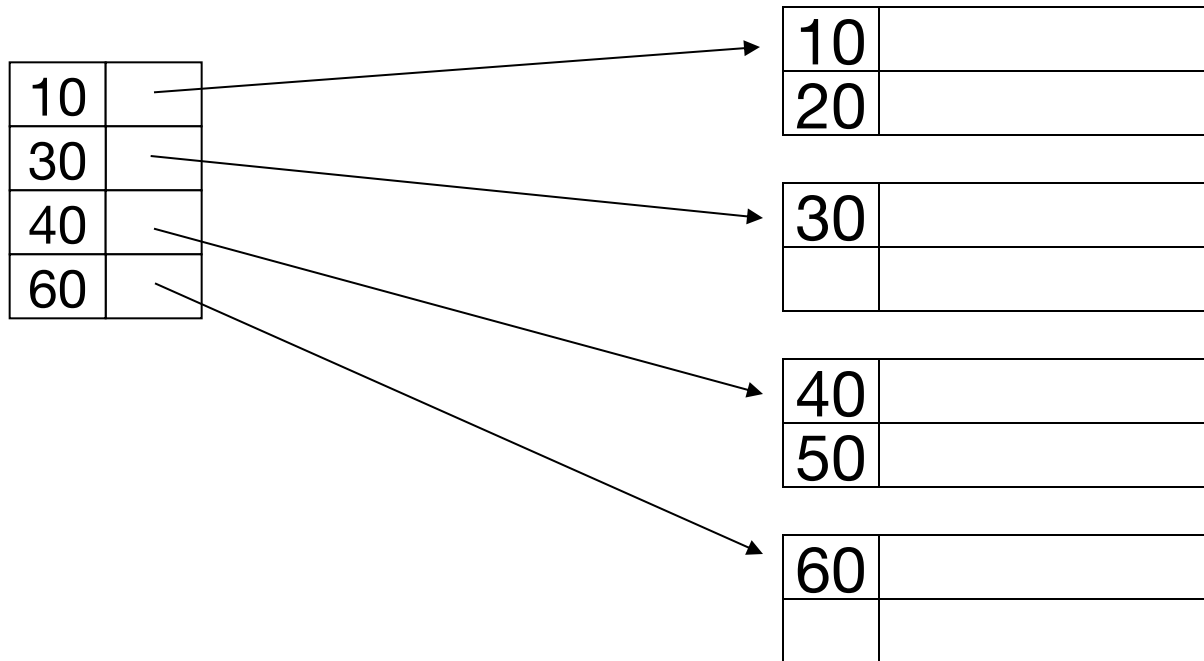
Deletion: Dense Index

– delete record 30



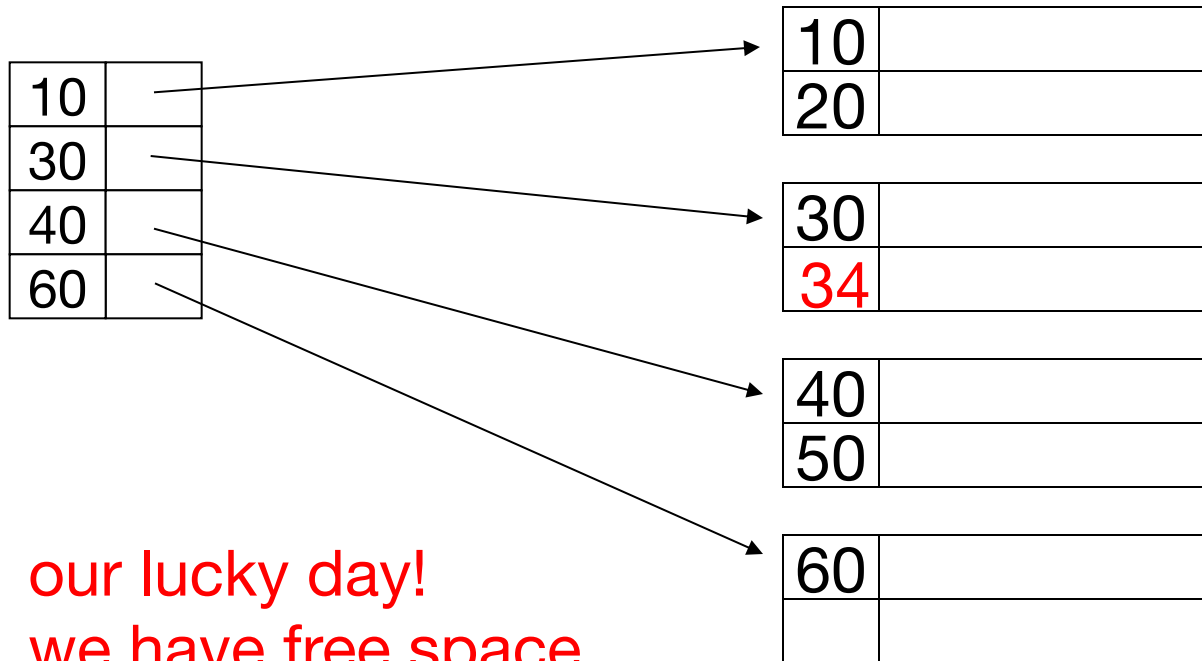
Insertion: Sparse Index

– insert record 34



Insertion: Sparse Index

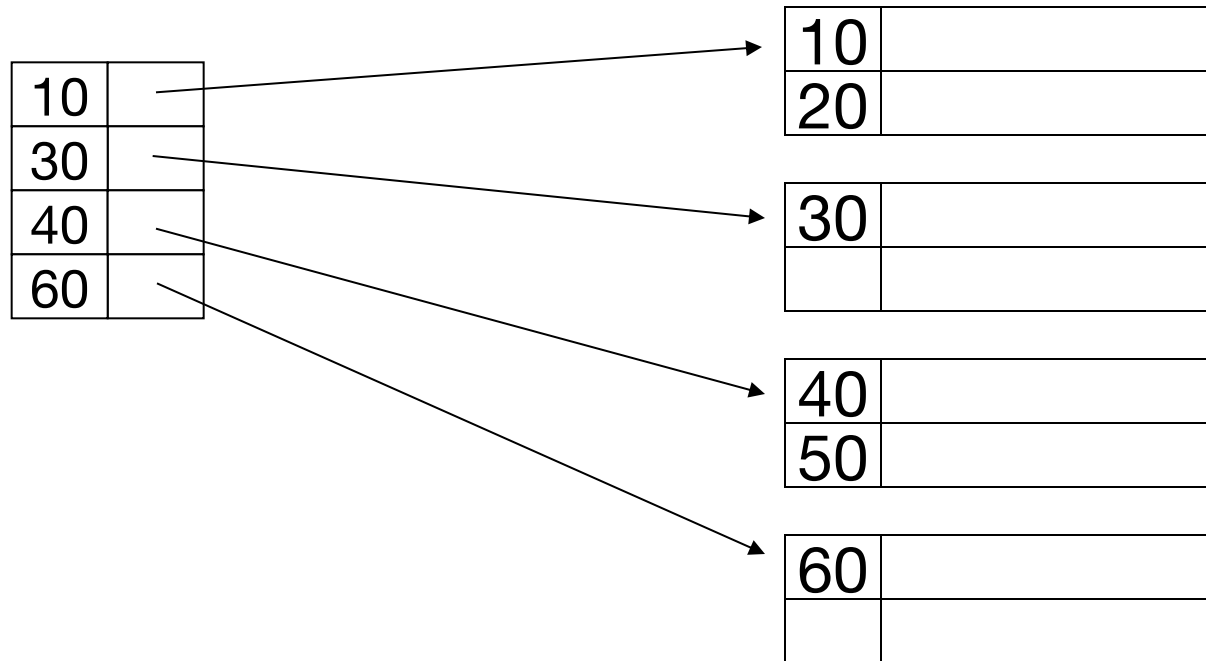
– insert record 34



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

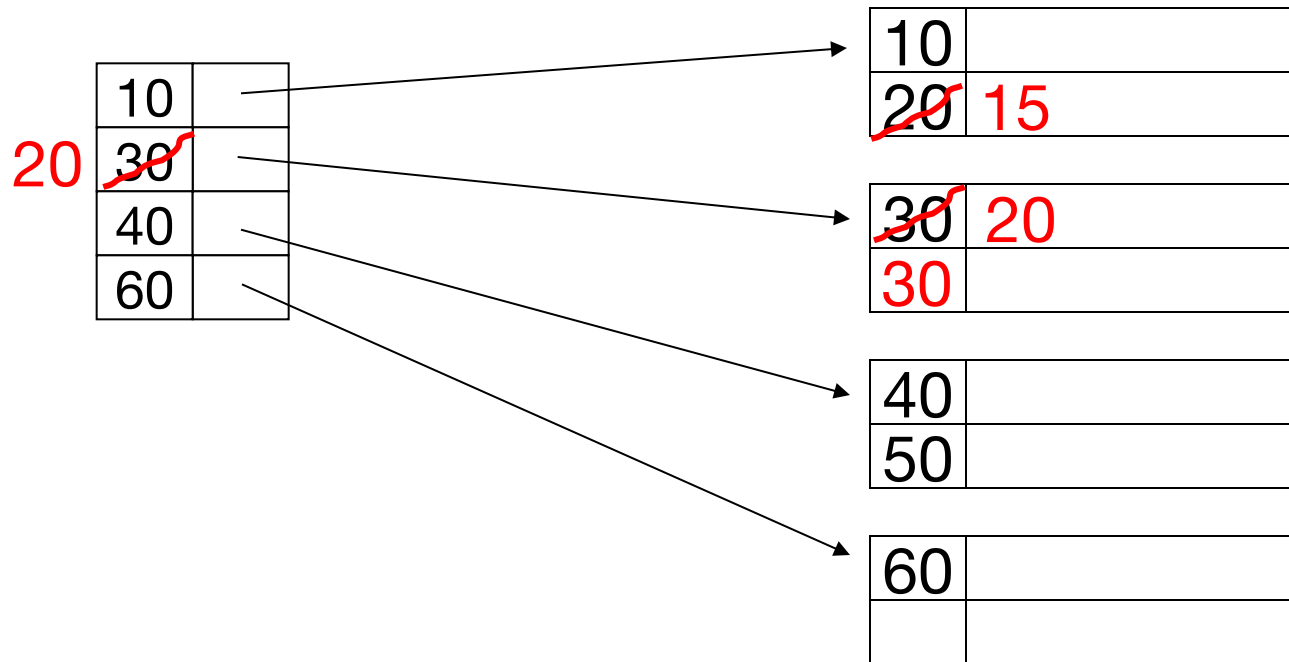
Insertion: Sparse Index

– insert record 15



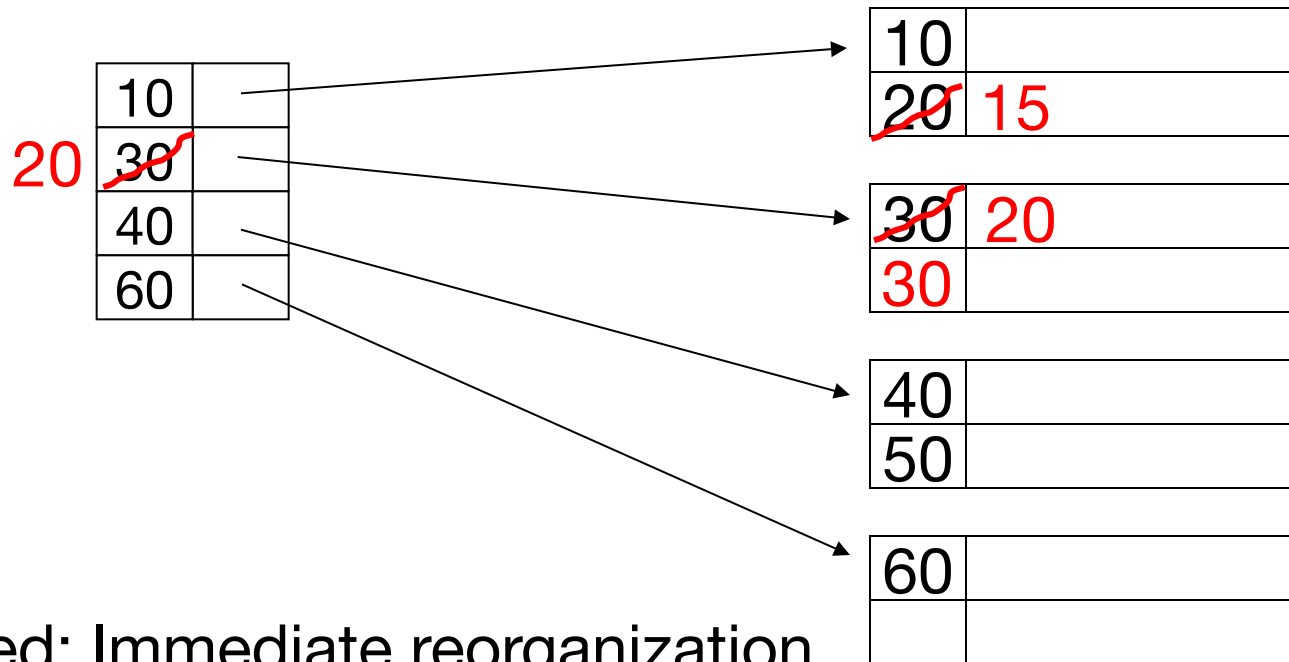
Insertion: Sparse Index

– insert record 15



Insertion: Sparse Index

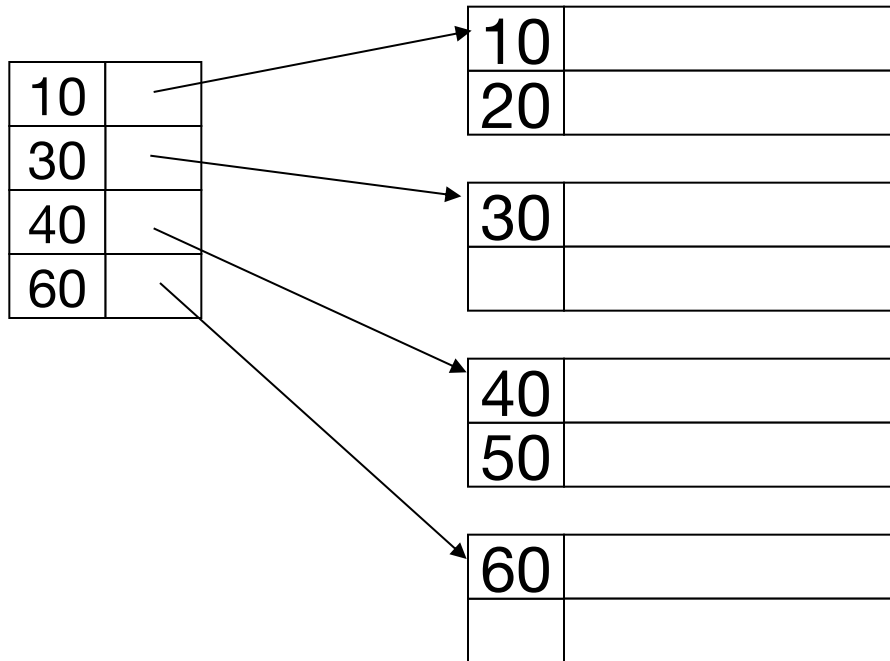
– insert record 15



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

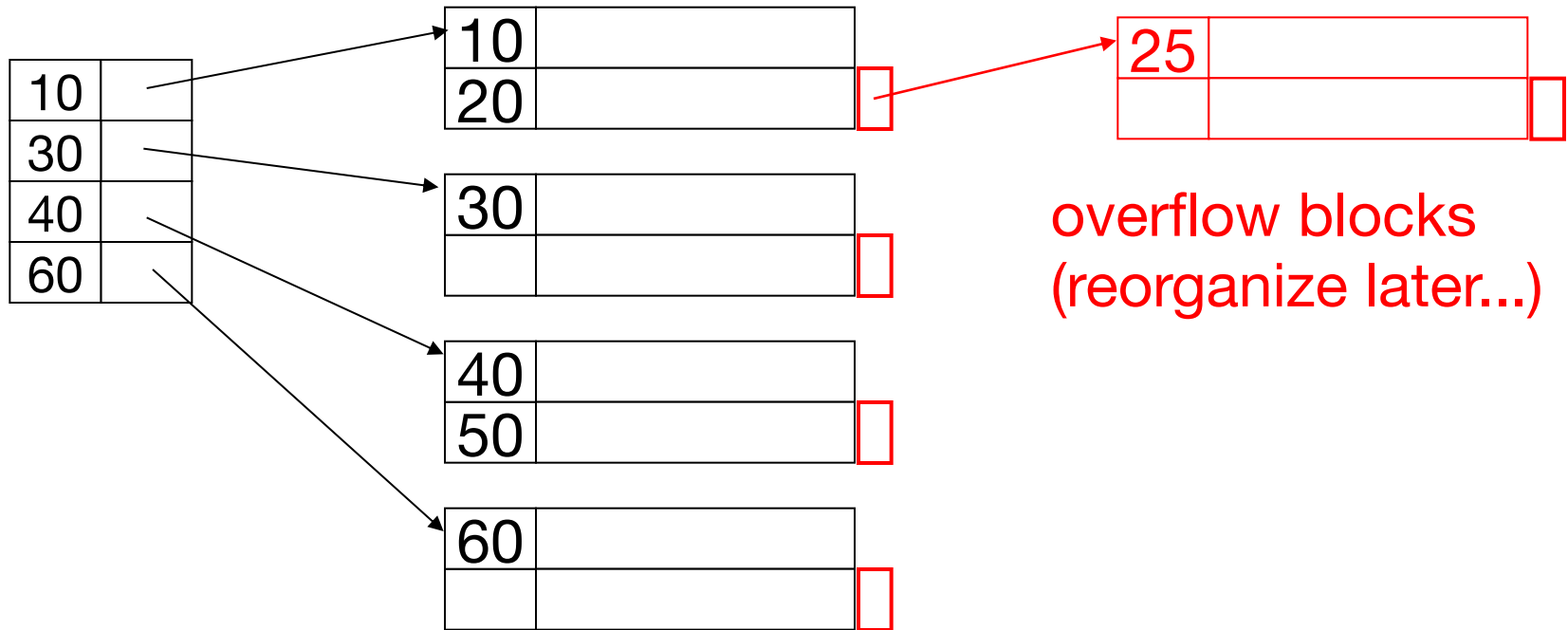
Insertion: Sparse Index

– insert record 25




Insertion: Sparse Index

– insert record 25



Secondary Indexes

Ordering
field



30	
50	

20	
70	

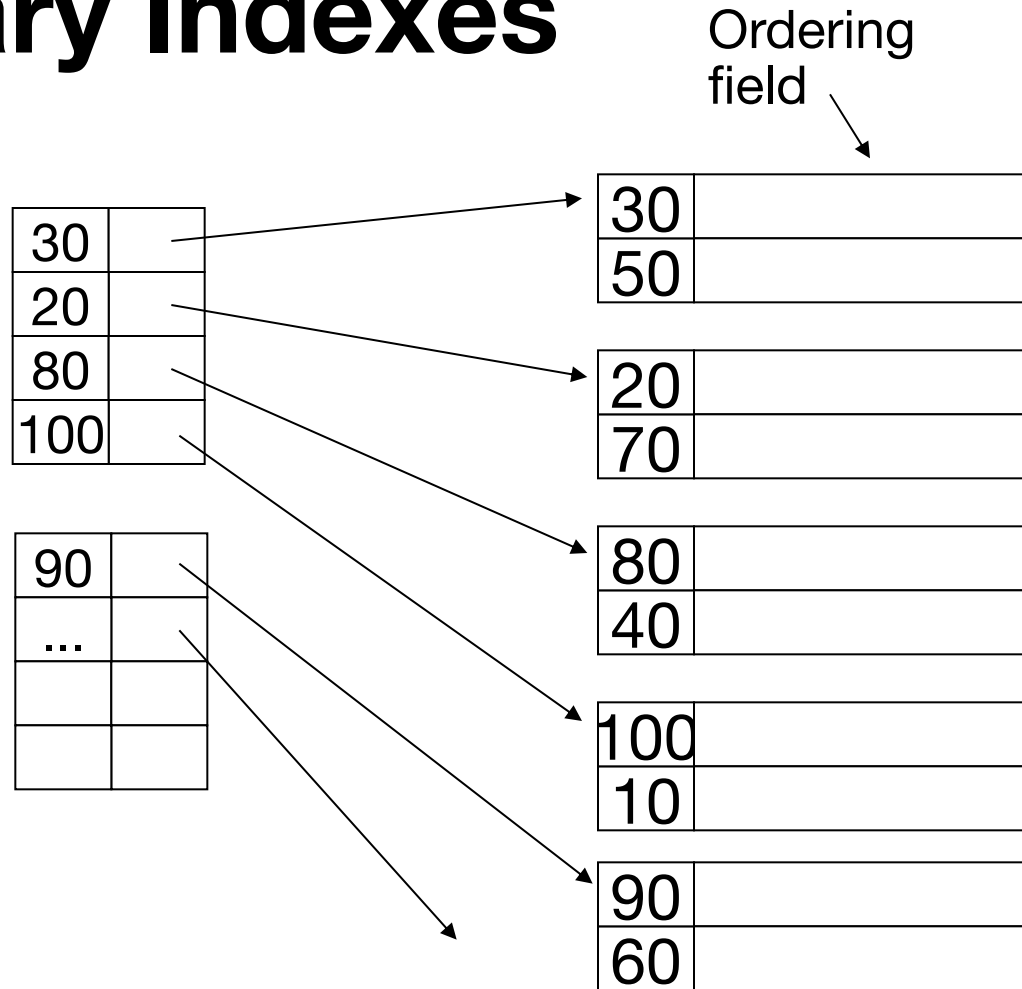
80	
40	

100	
10	

90	
60	

Secondary Indexes

Sparse index:



Secondary Indexes

Sparse Index:

30	
20	
80	
100	

90	
...	

Ordering
field

30	
50	

20	
70	

80	
40	

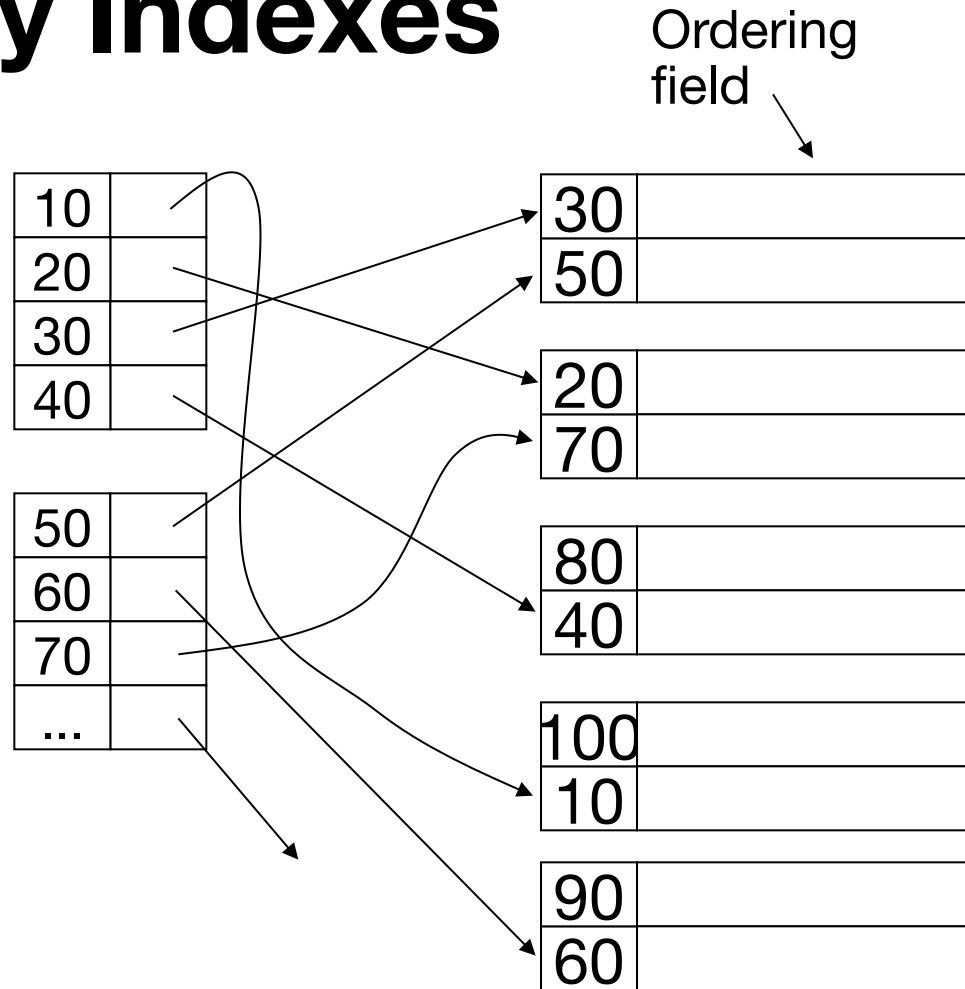
100	
10	

90	
60	

does not make sense!

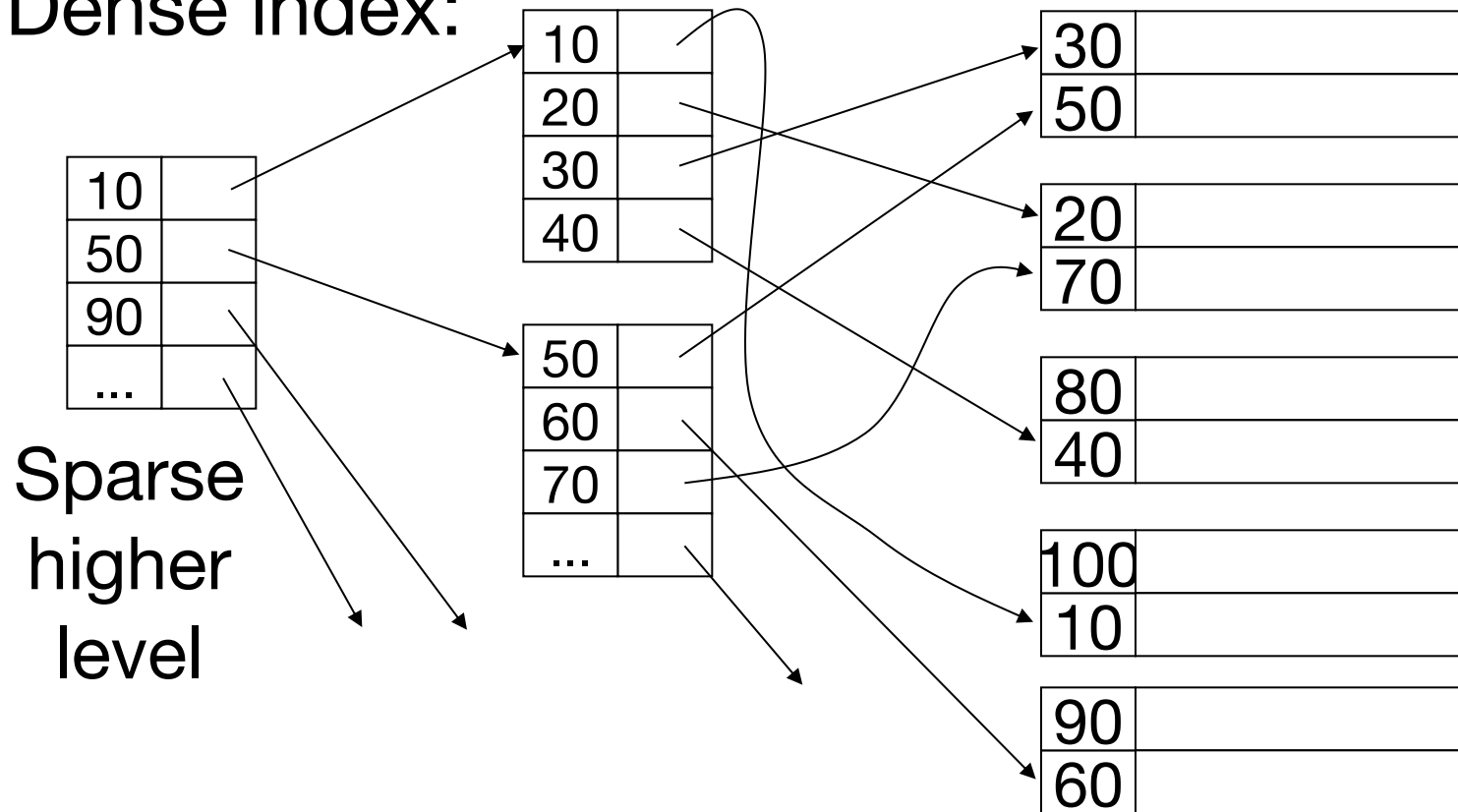
Secondary Indexes

Dense index:

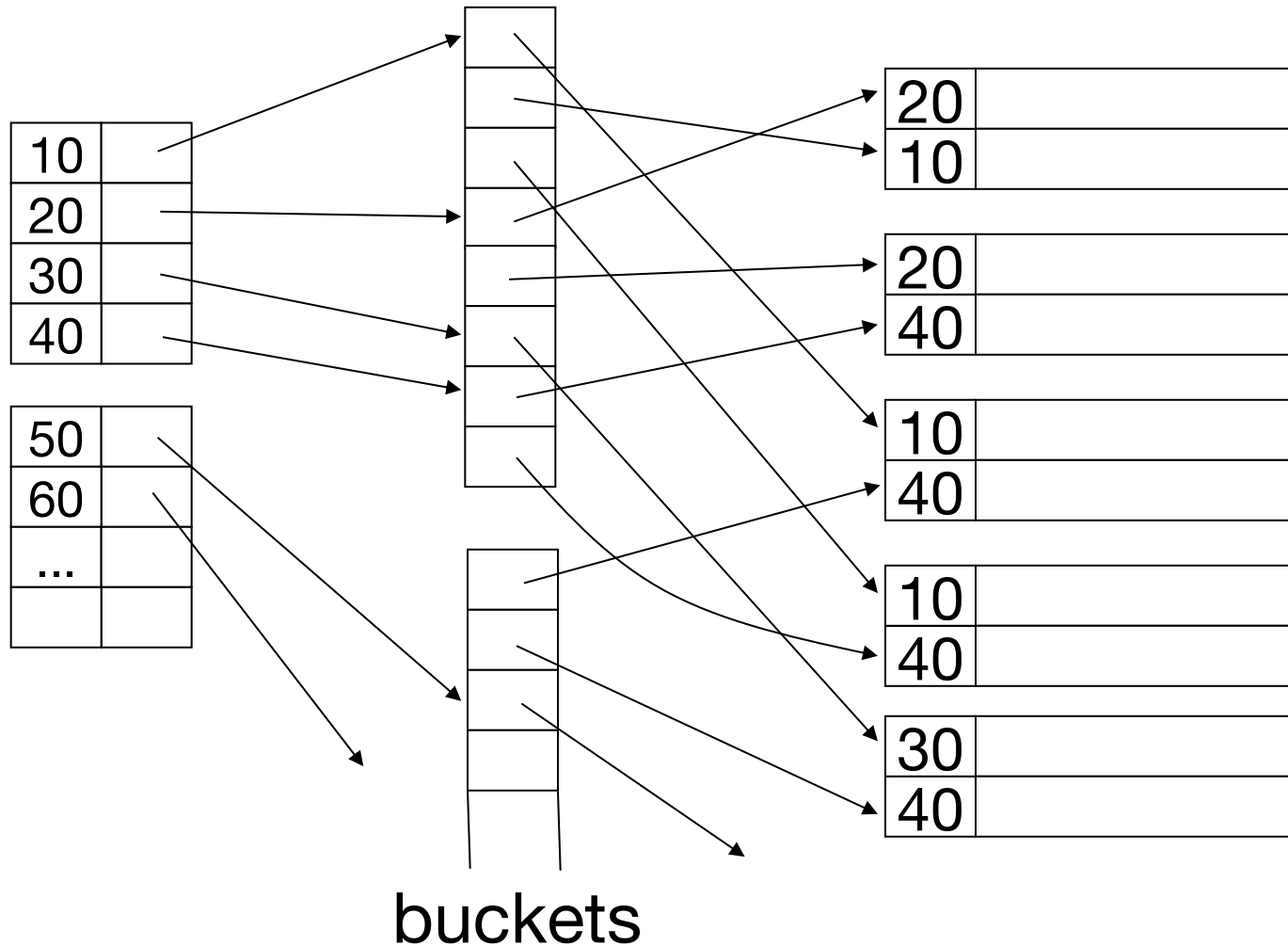


Secondary Indexes

Dense index:



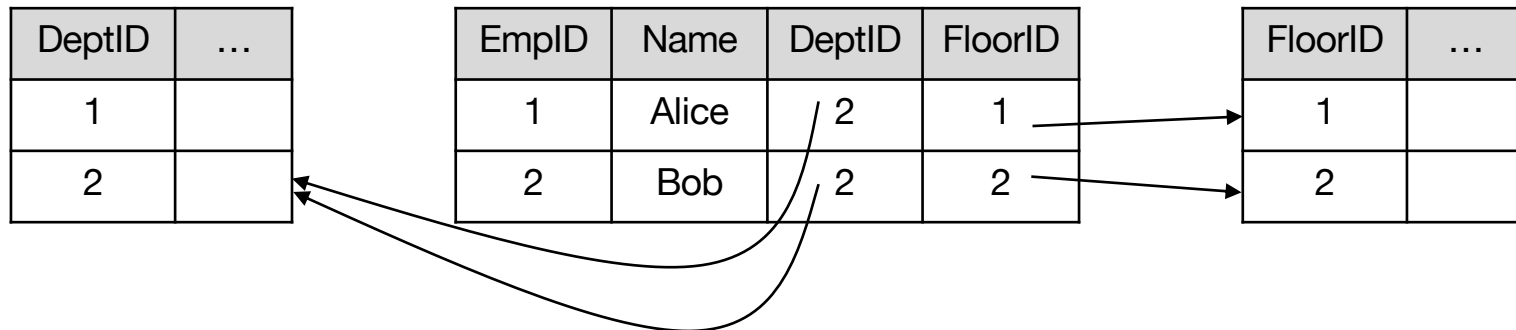
Duplicate Values in Secondary Indexes



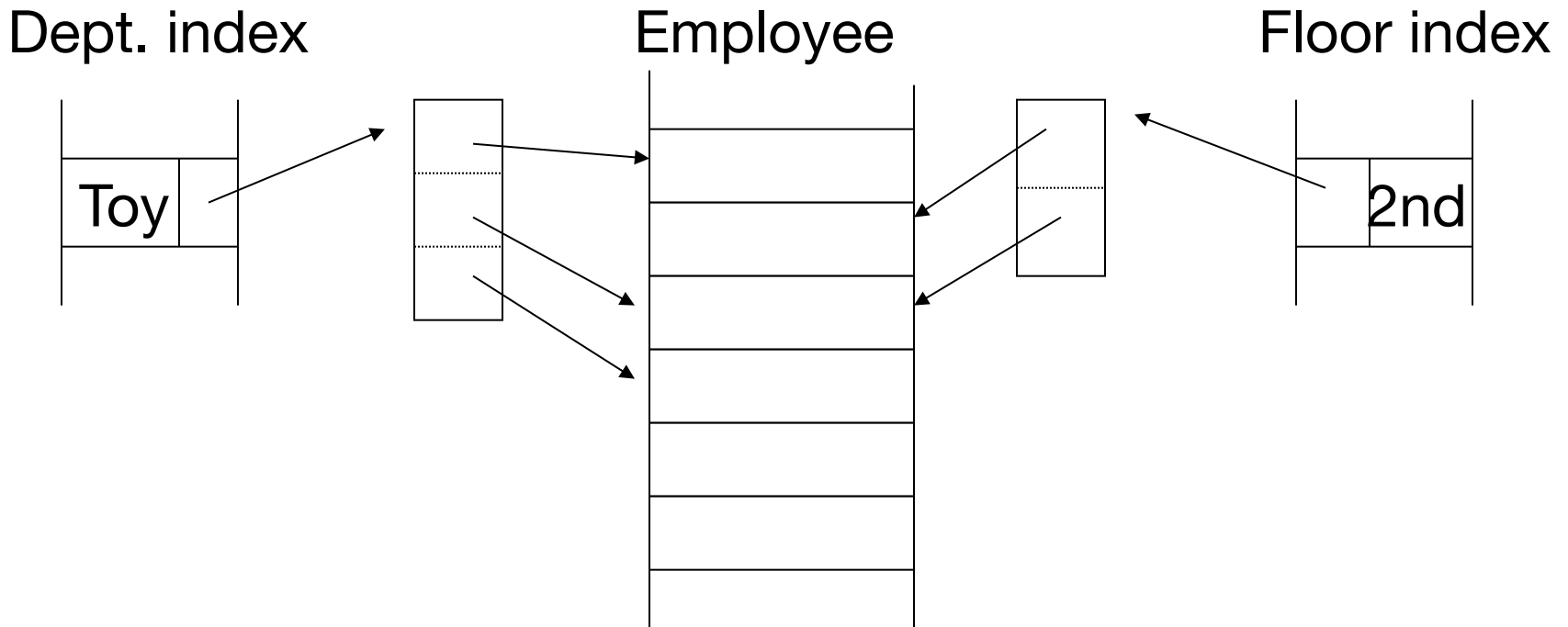
Another Benefit of Buckets

Can compute complex queries through Boolean operations on record pointer lists

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



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



Intersect “Toy” bucket and “2nd floor” buckets
to get list of matching employees

Often Used in Text Search

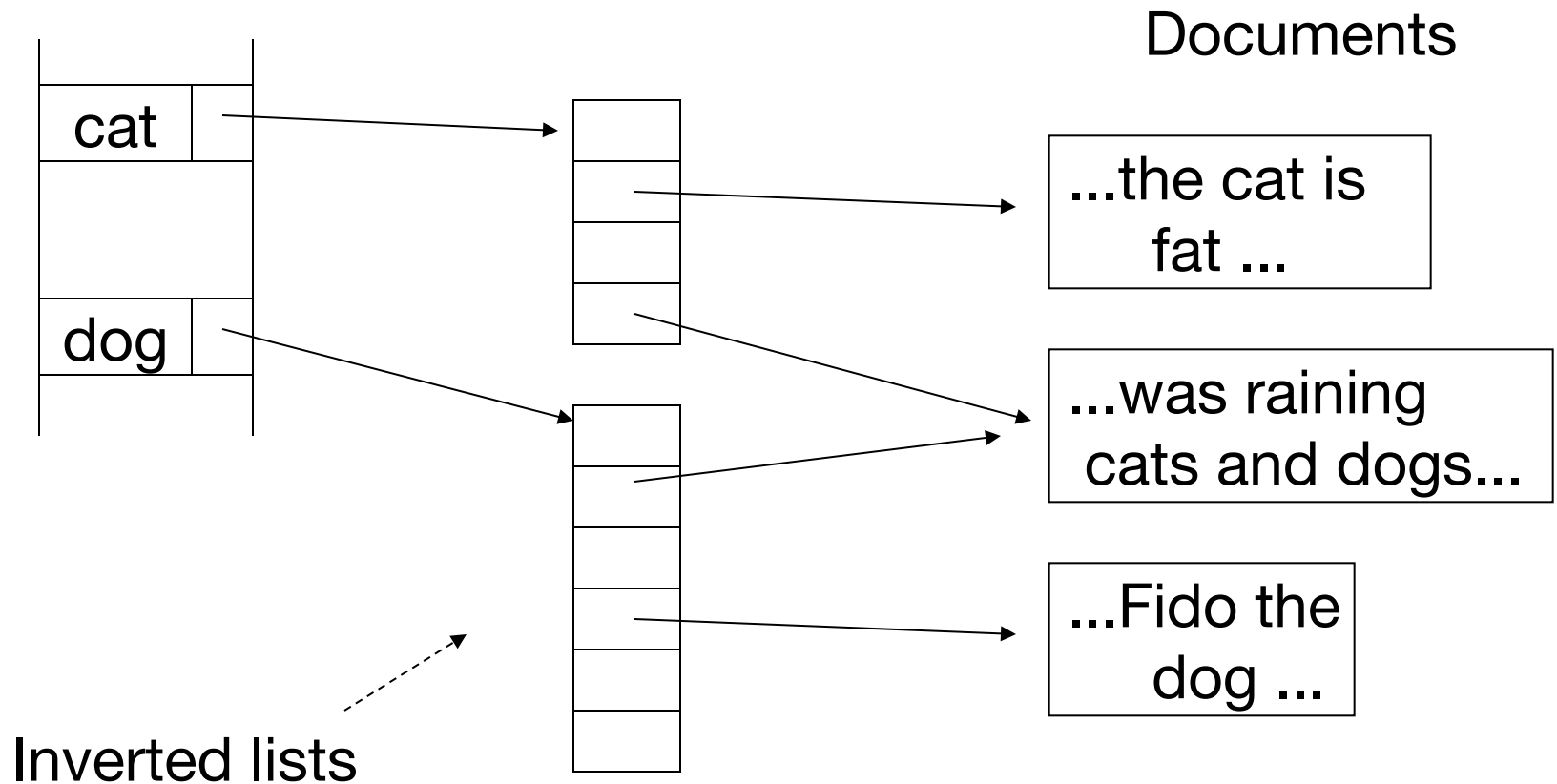
Documents

...the cat is
fat ...

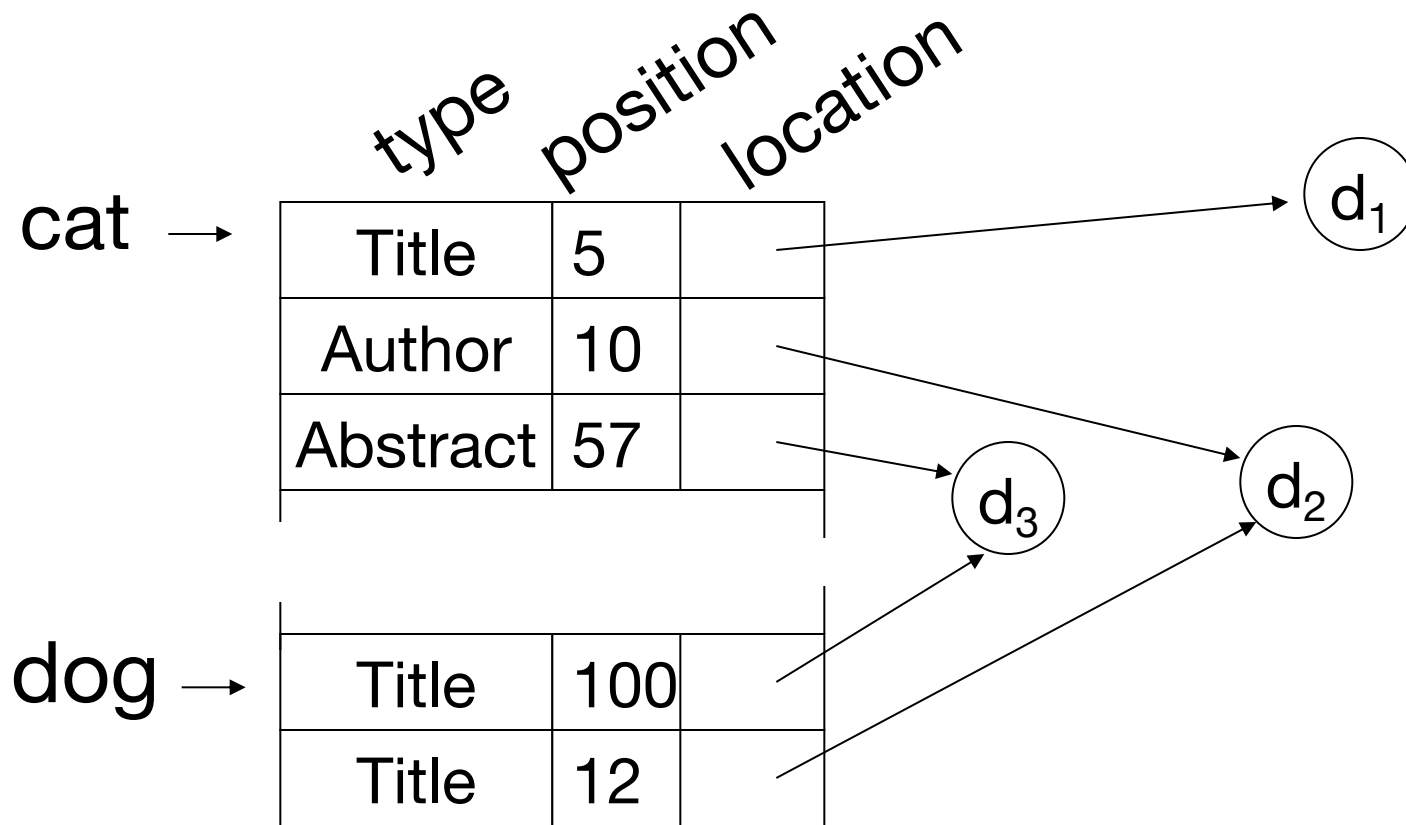
...was raining
cats and dogs...

...Fido the
dog ...

Often Used in Text Search



Extension: 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

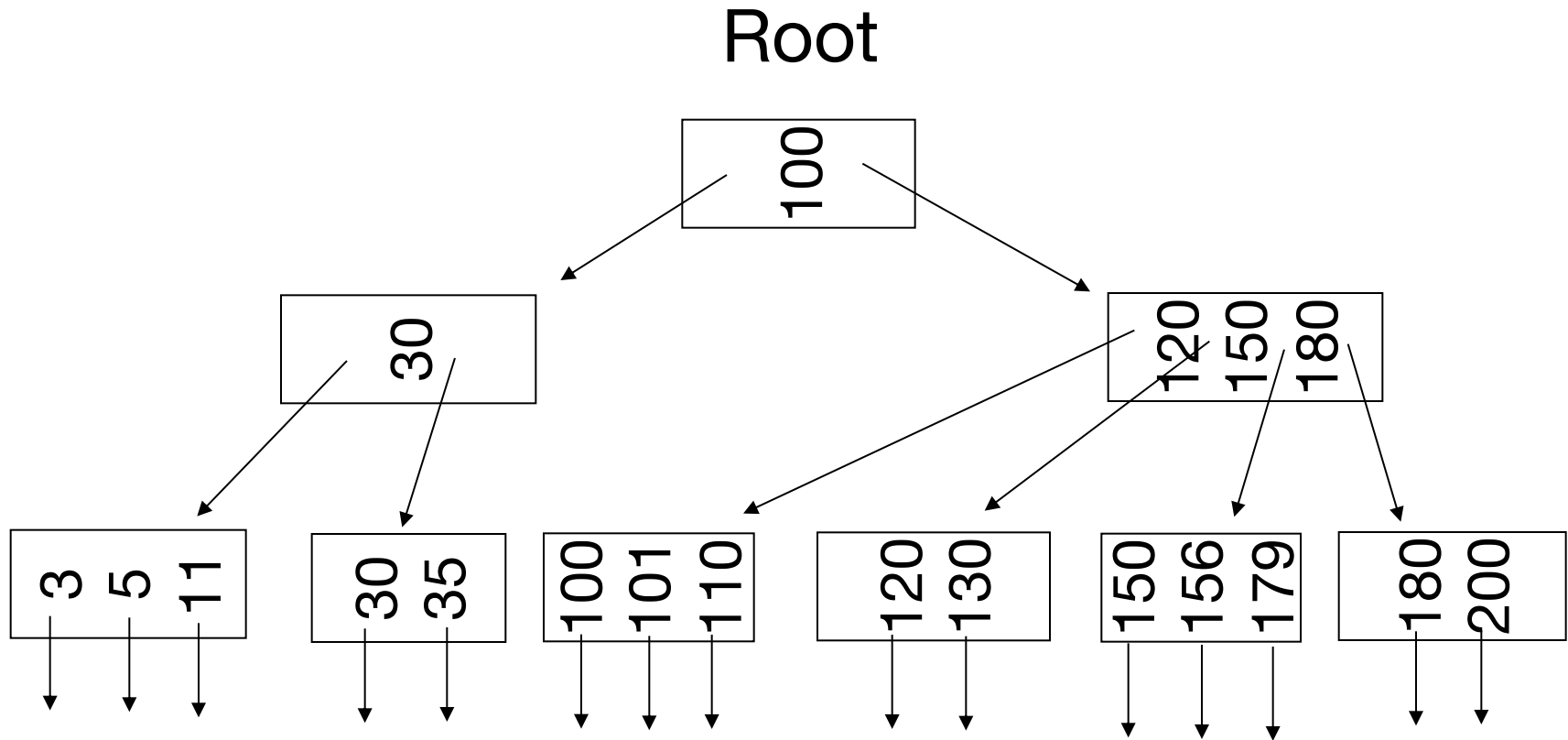
Another type of index

- » Give up on sequentiality of index
- » Try to get “balance”

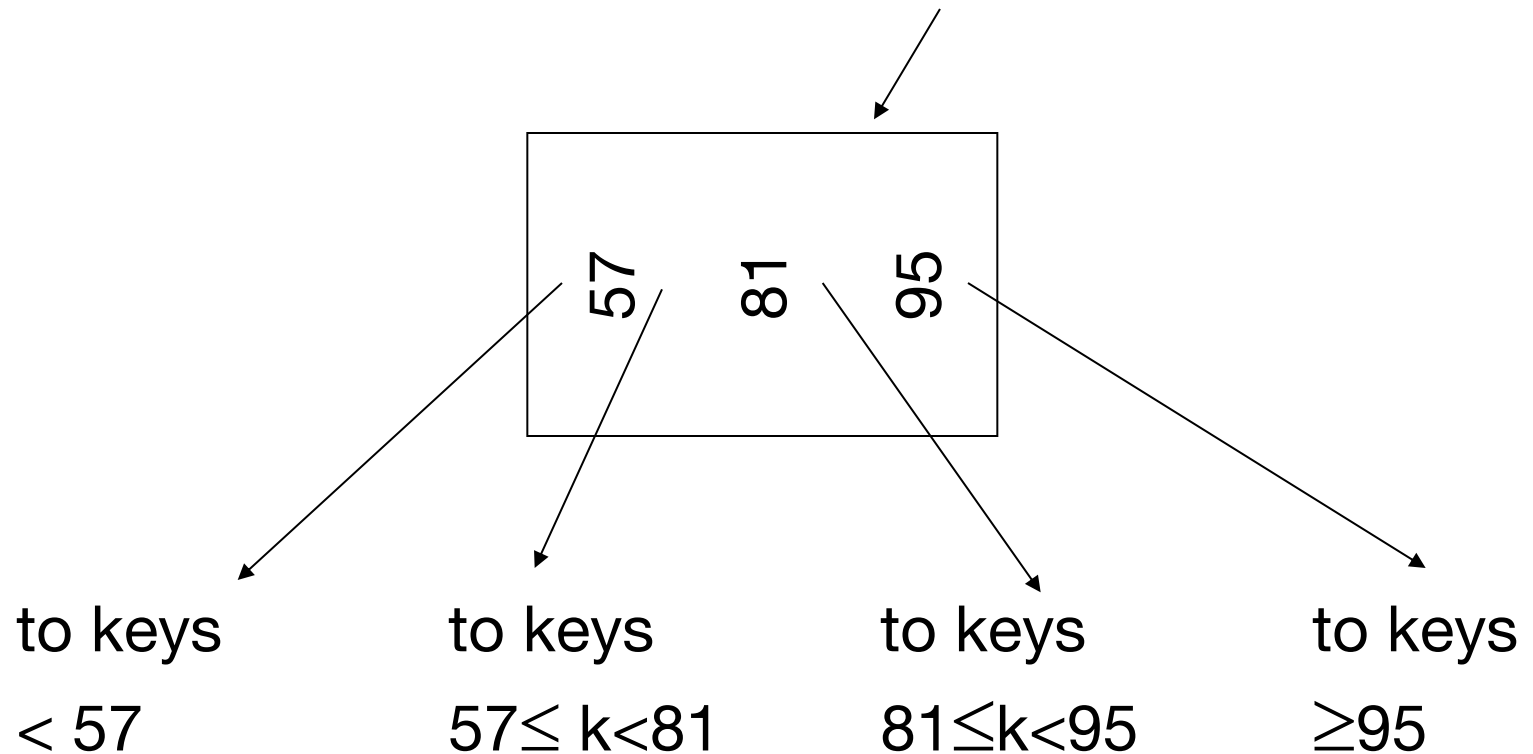
Note: the exact data structure we'll look at is a **B+ tree**, but plain old “B-trees” are similar

B+ Tree Example

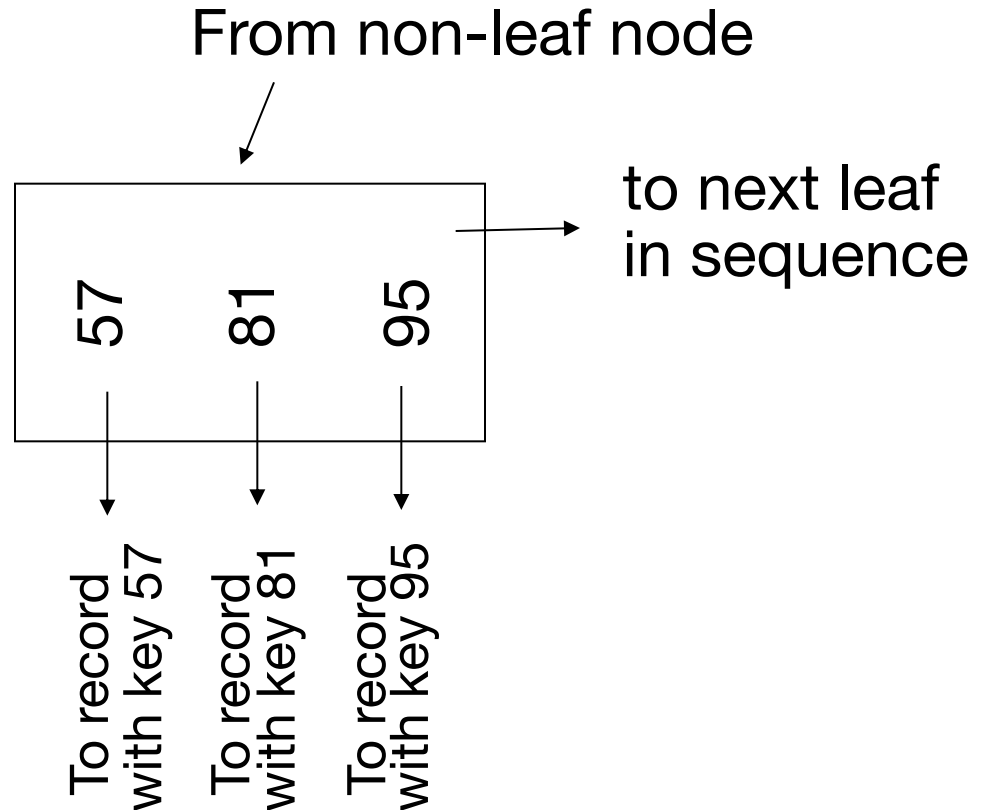
(n = 3)



Sample Non-Leaf



Sample Leaf Node



Size of Nodes on Disk

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

(Fixed size nodes)

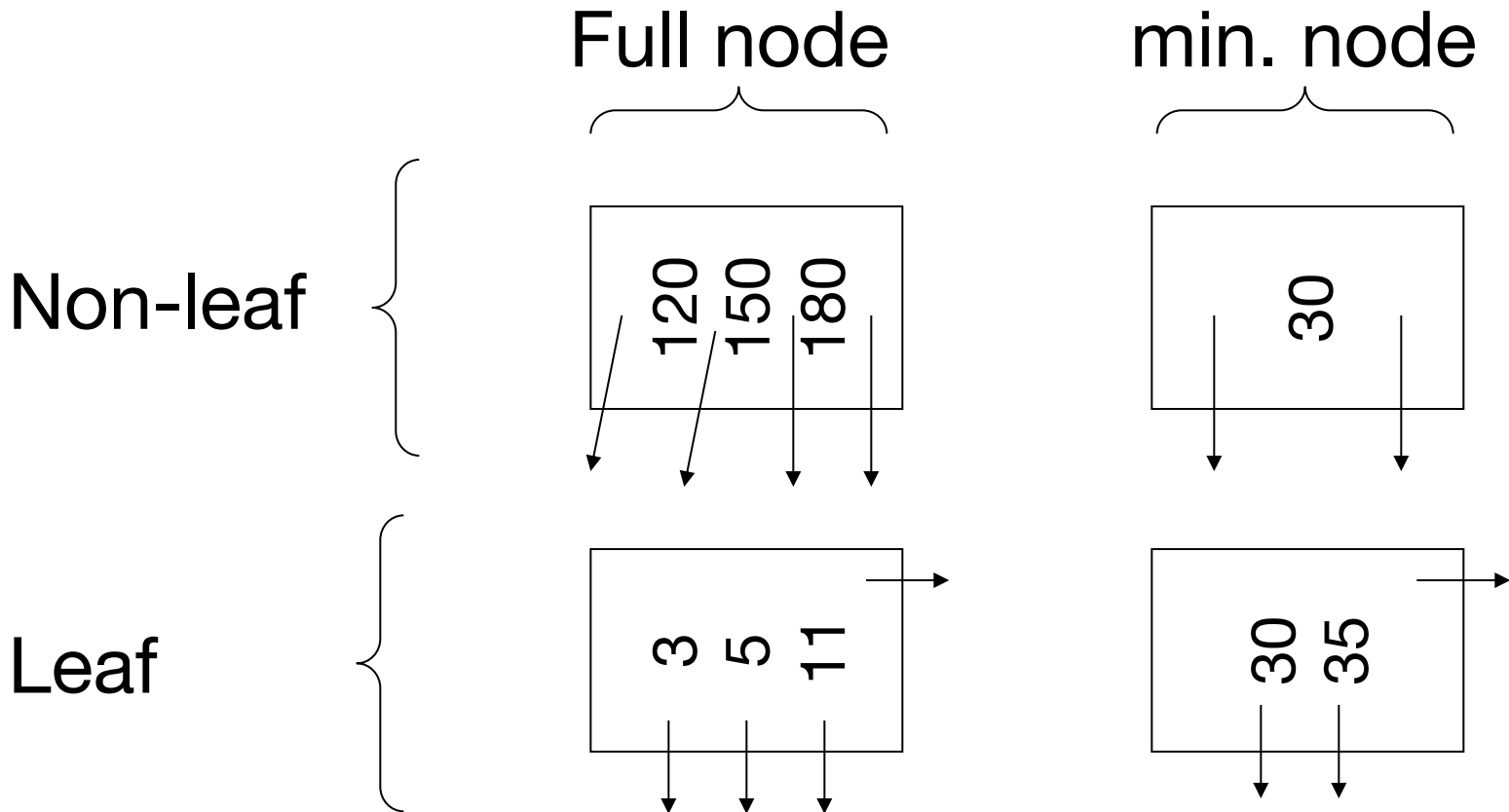
Don't Want Nodes to be Too Empty

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

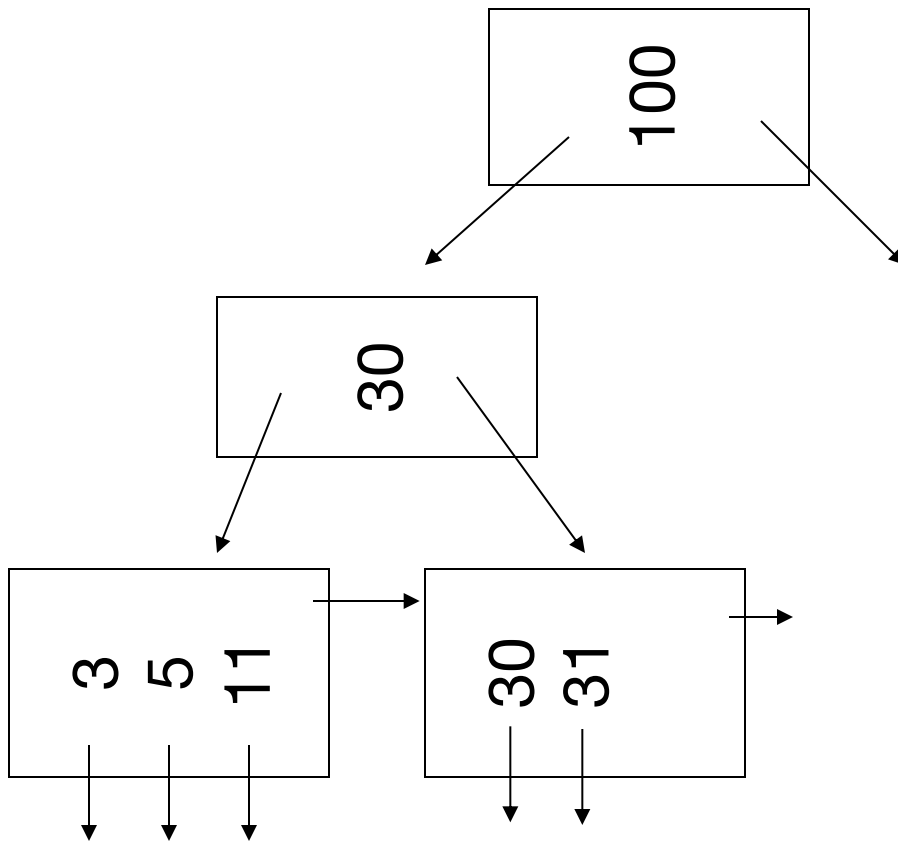
* When there is only one record in the B+ tree, min pointers in the root is 1 (the other pointers are null)

Insert Into B+ Tree

- (a) simple case
 - » space available in leaf
- (b) leaf overflow
- (c) non-leaf overflow
- (d) new root

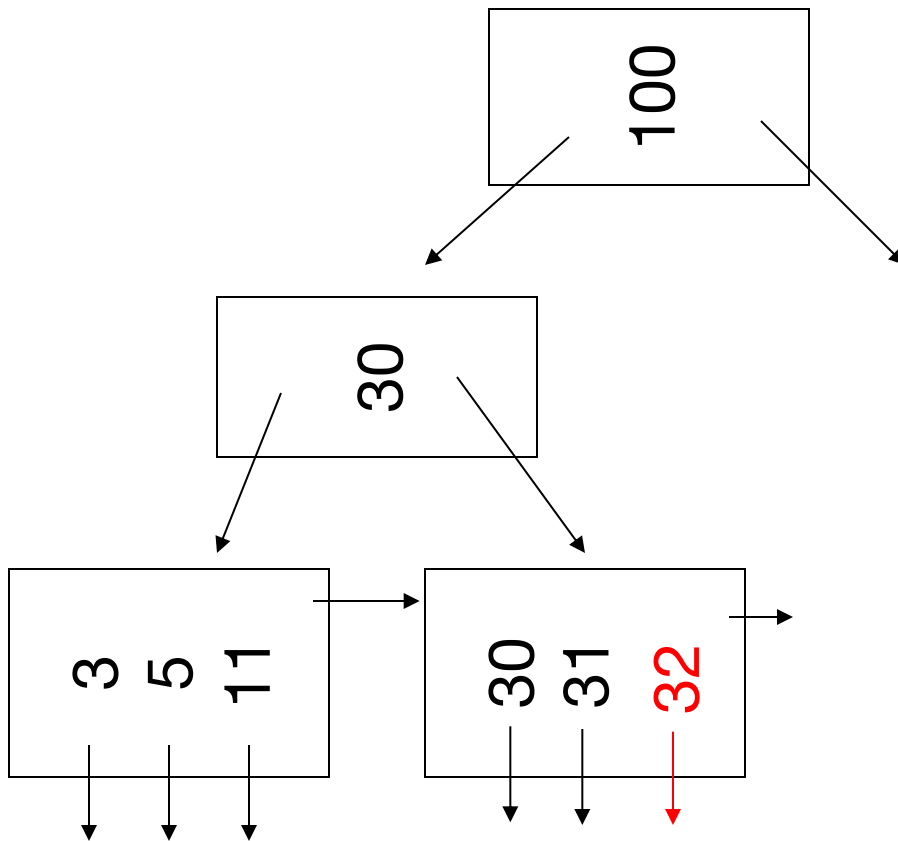
(a) Insert key = 32

$n=3$



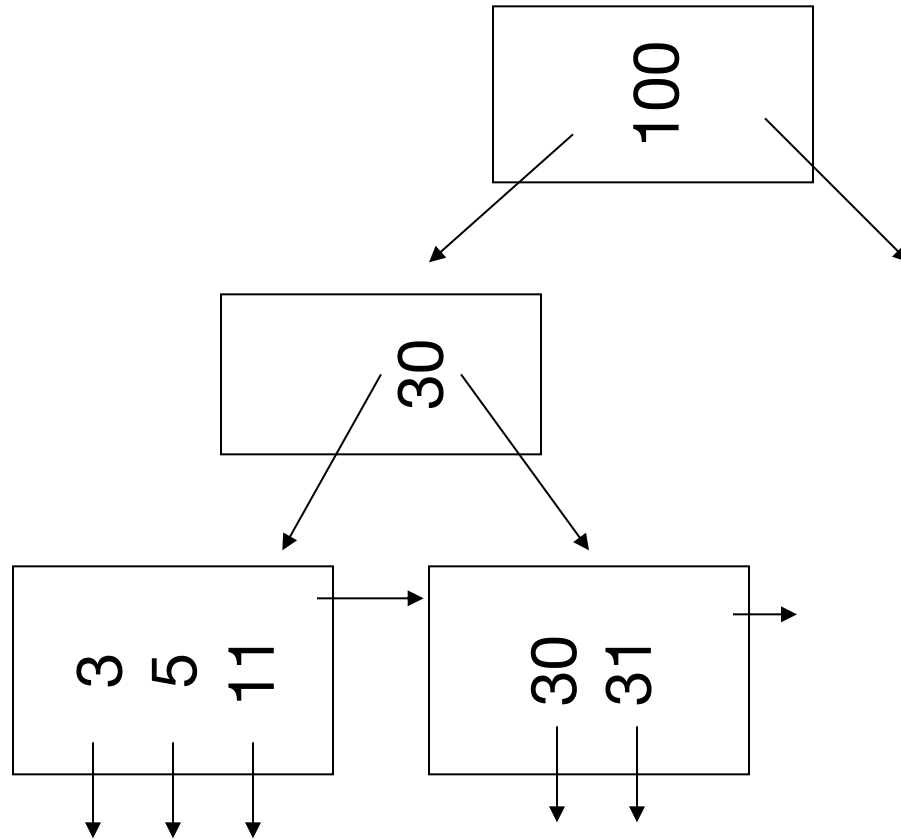
(a) Insert key = 32

n=3



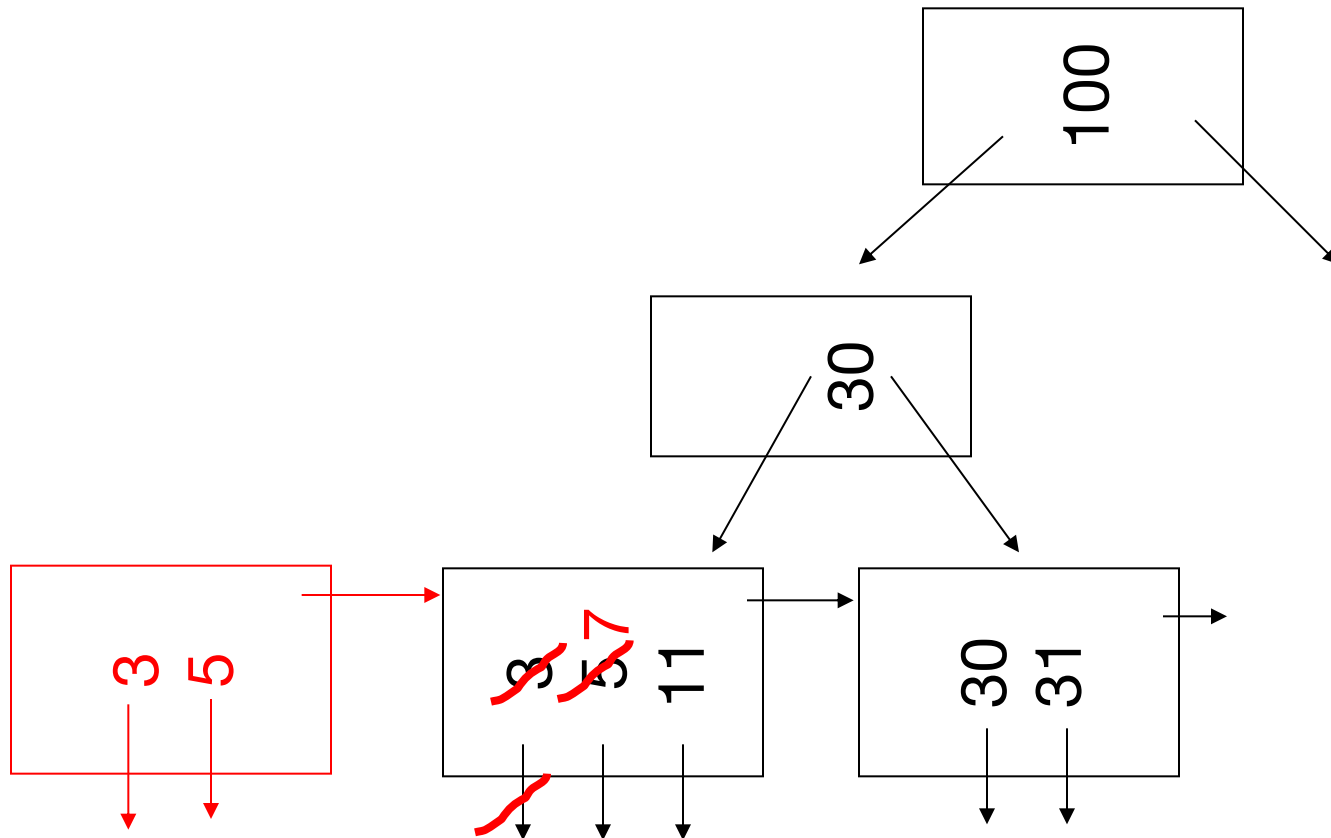
(a) Insert key = 7

n=3



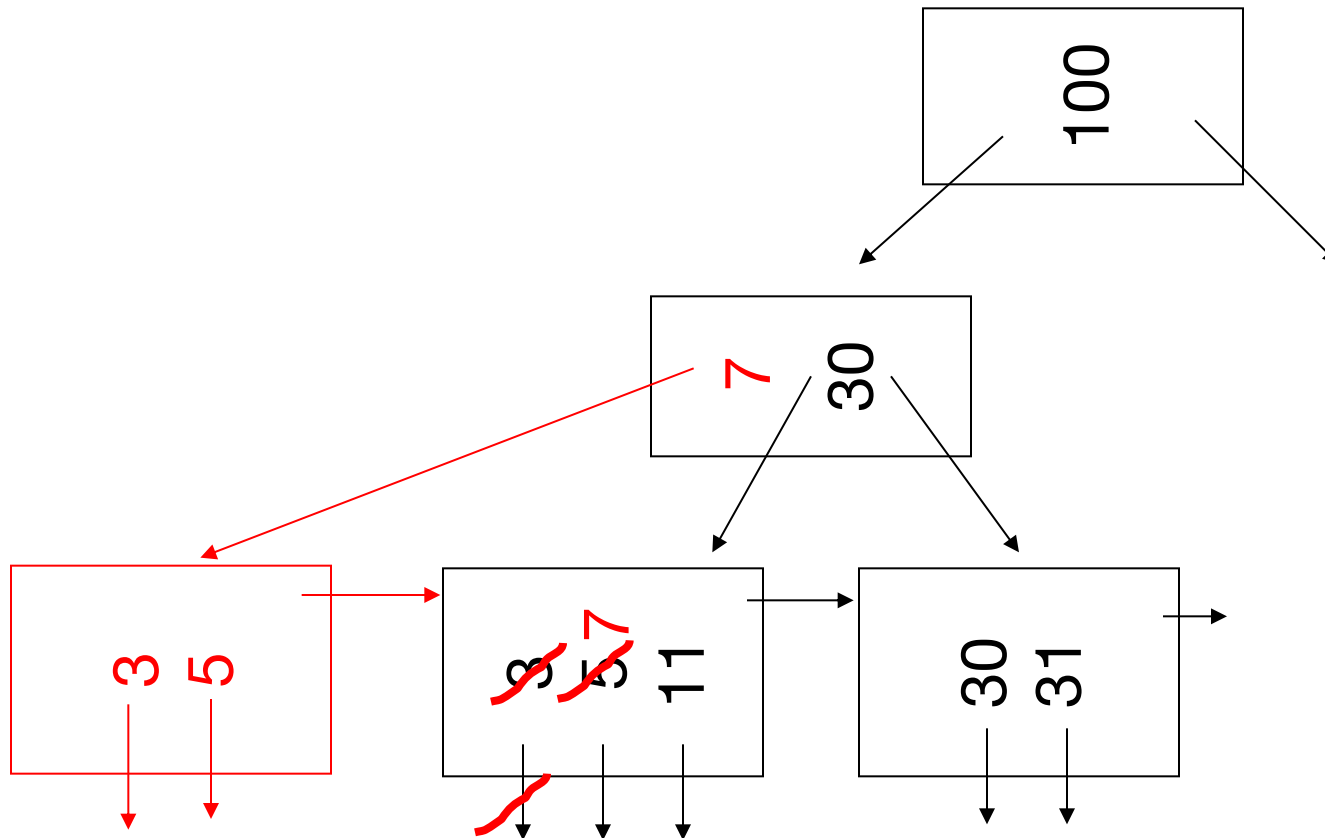
(a) Insert key = 7

n=3



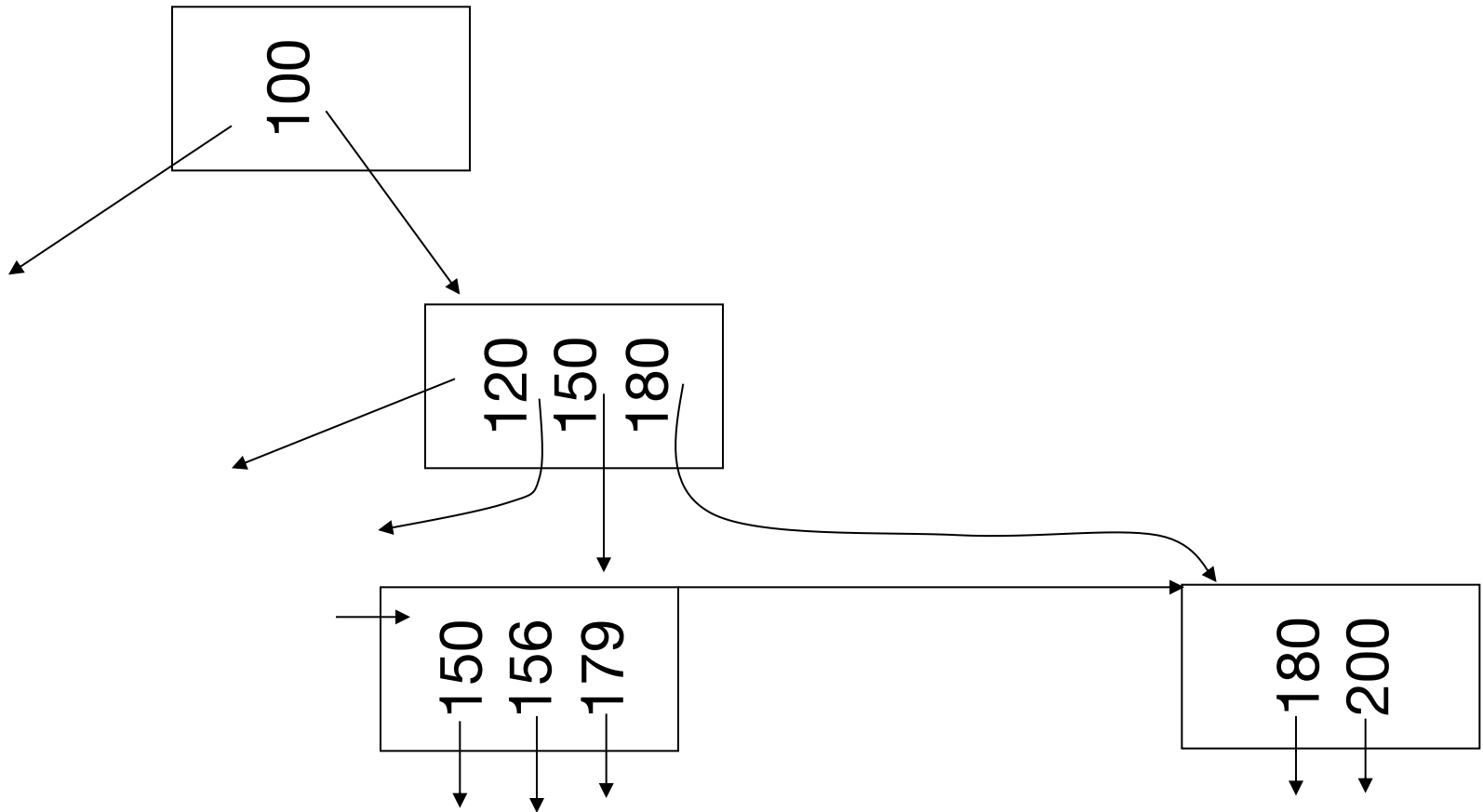
(a) Insert key = 7

n=3



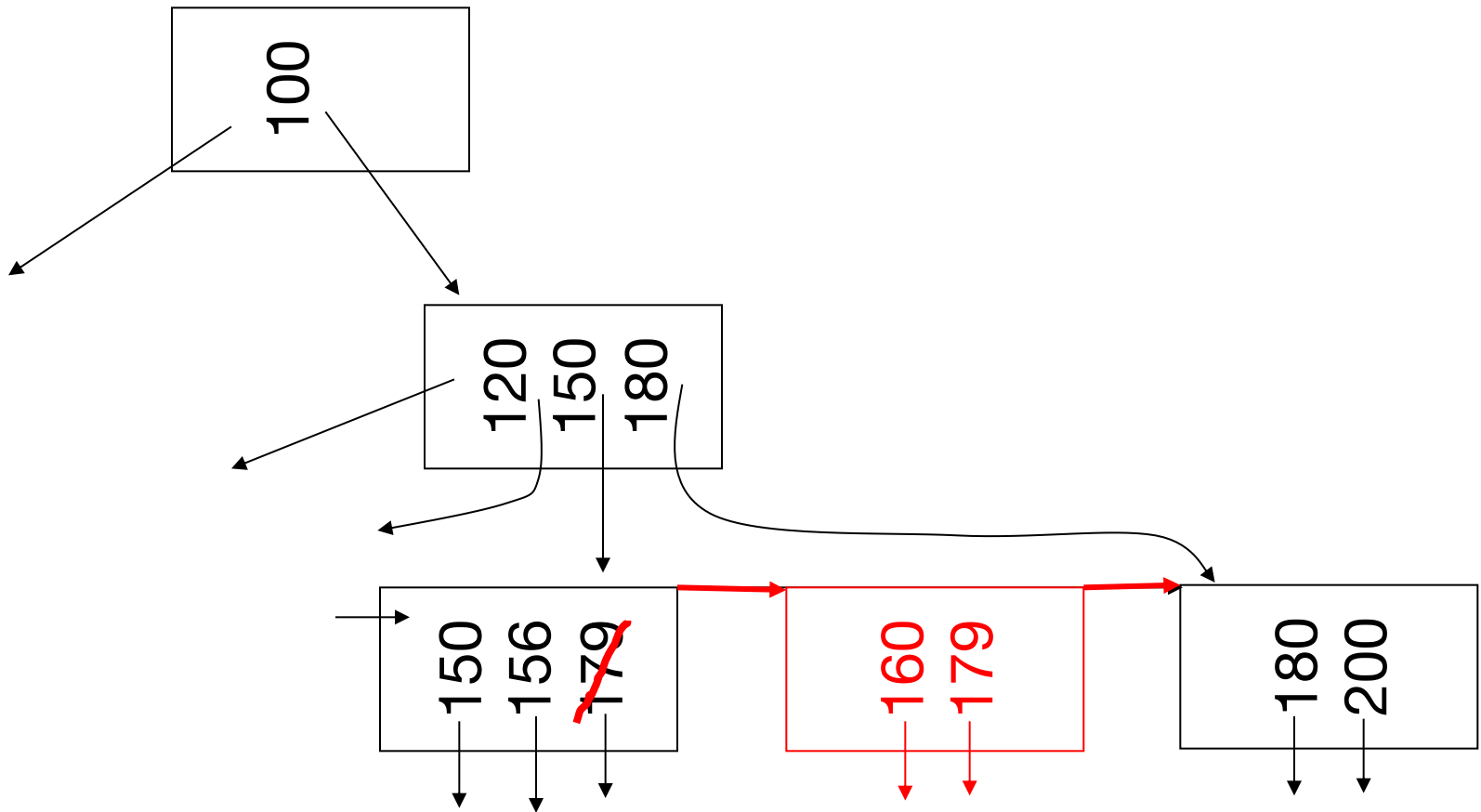
(c) Insert key = 160

n=3



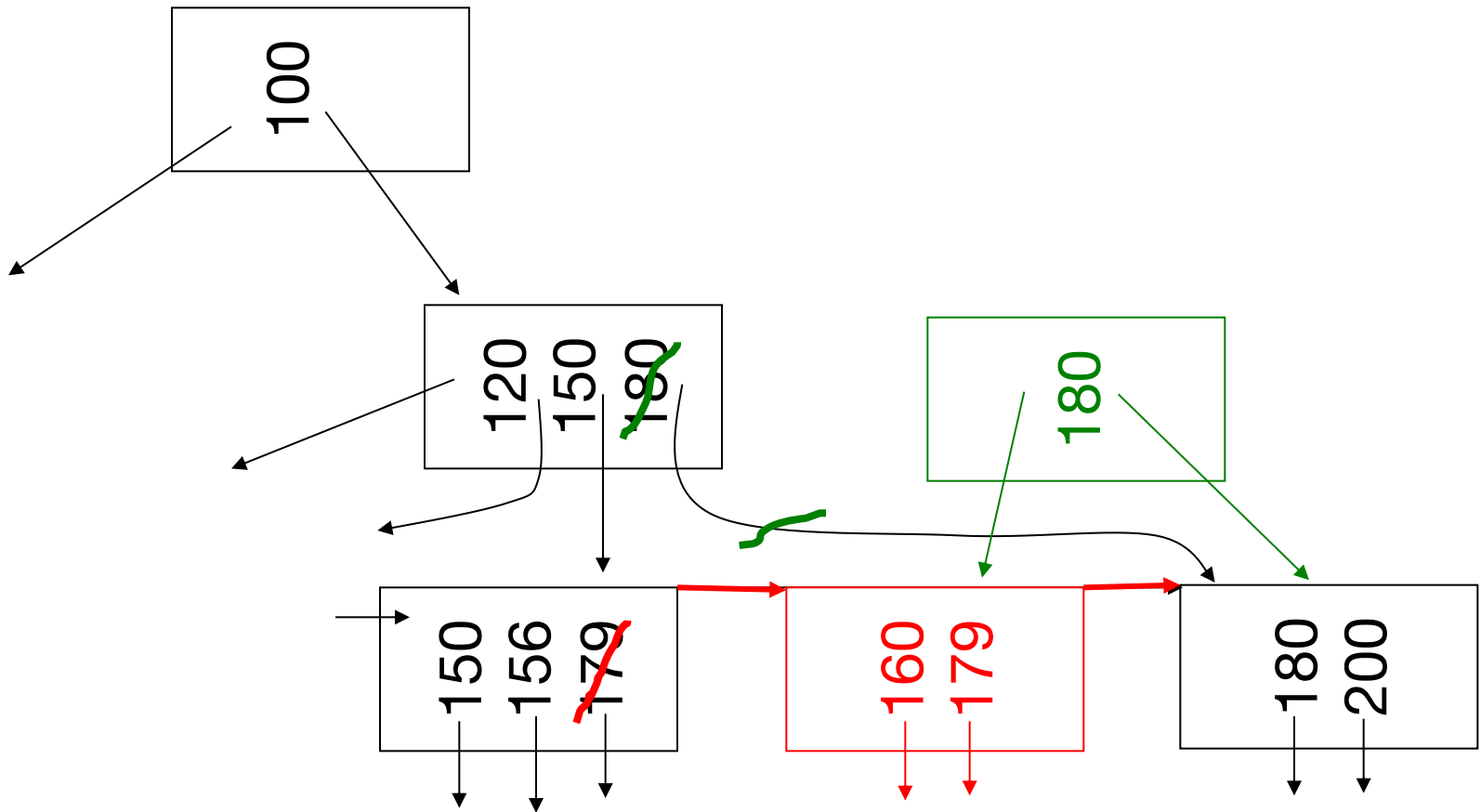
(c) Insert key = 160

n=3



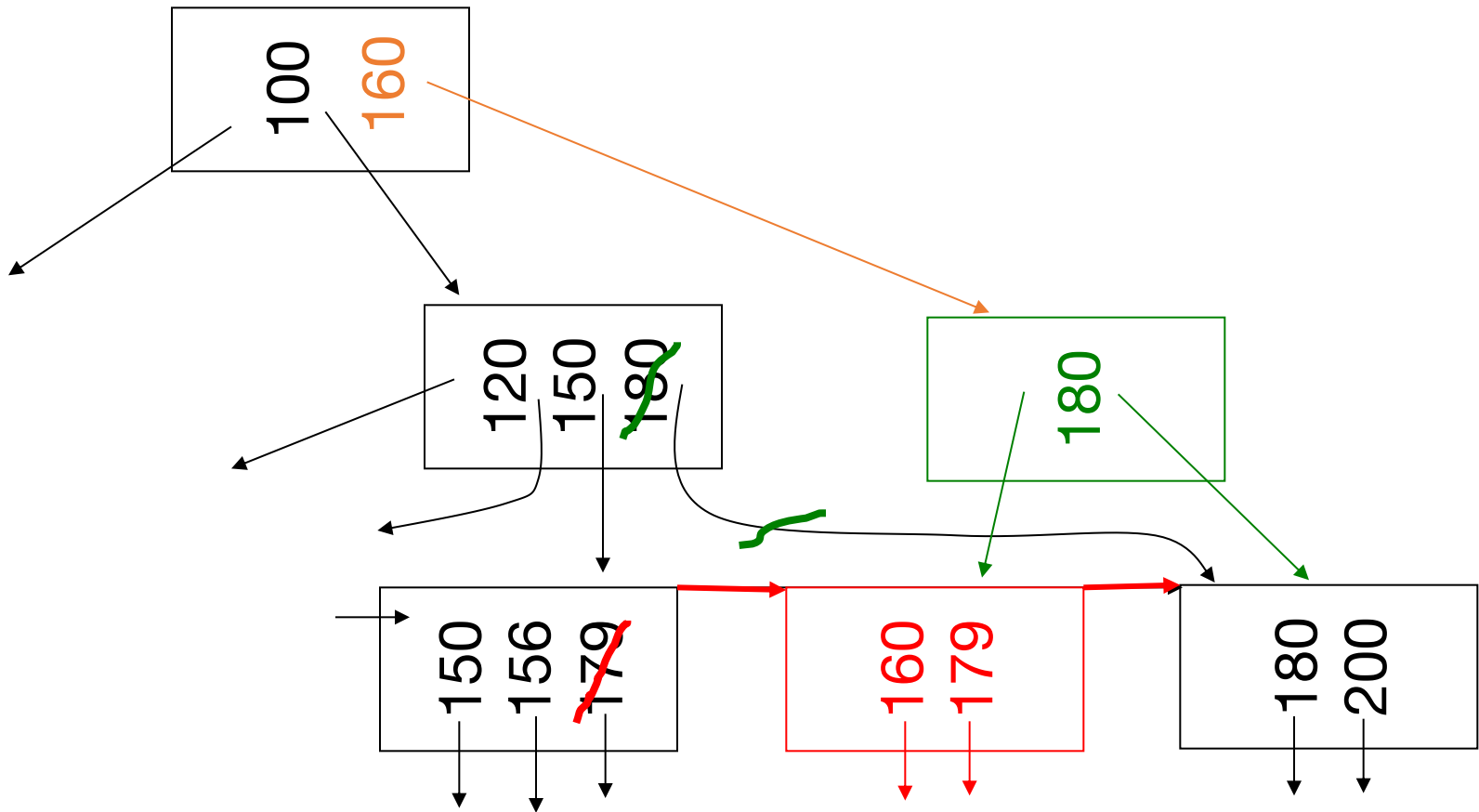
(c) Insert key = 160

n=3



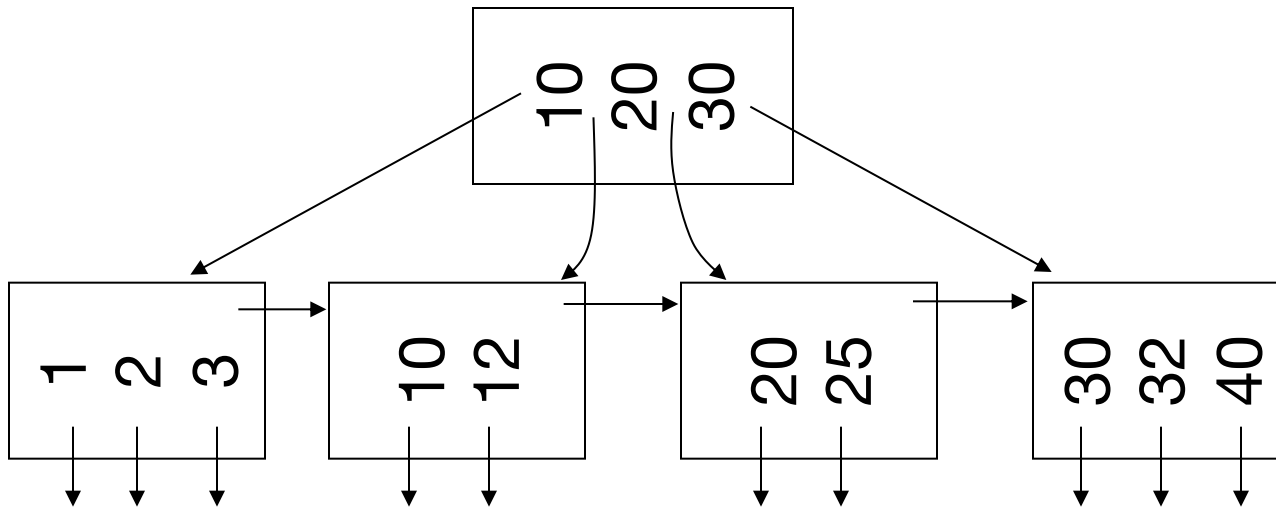
(c) Insert key = 160

n=3



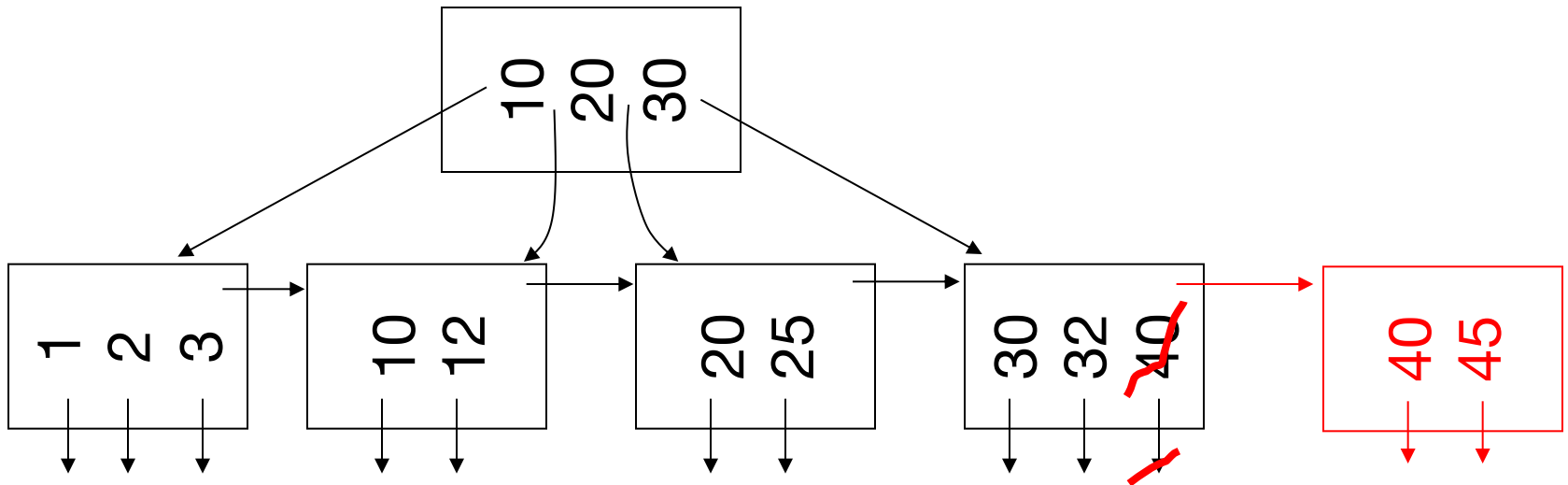
(d) New root, insert 45

n=3



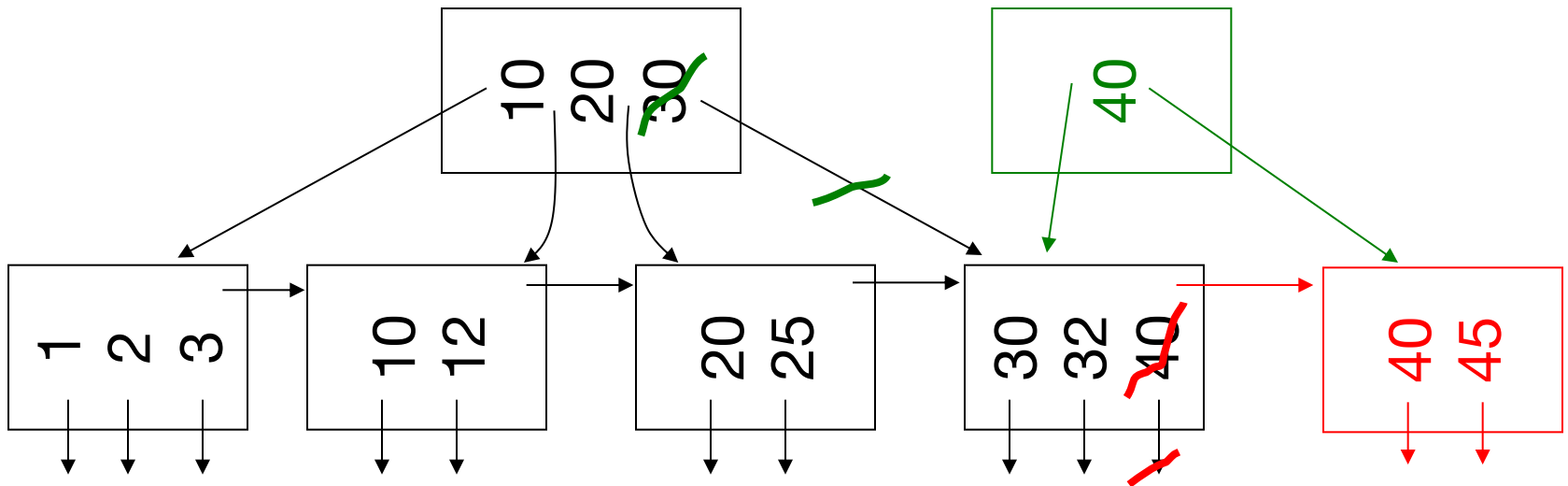
(d) New root, insert 45

n=3



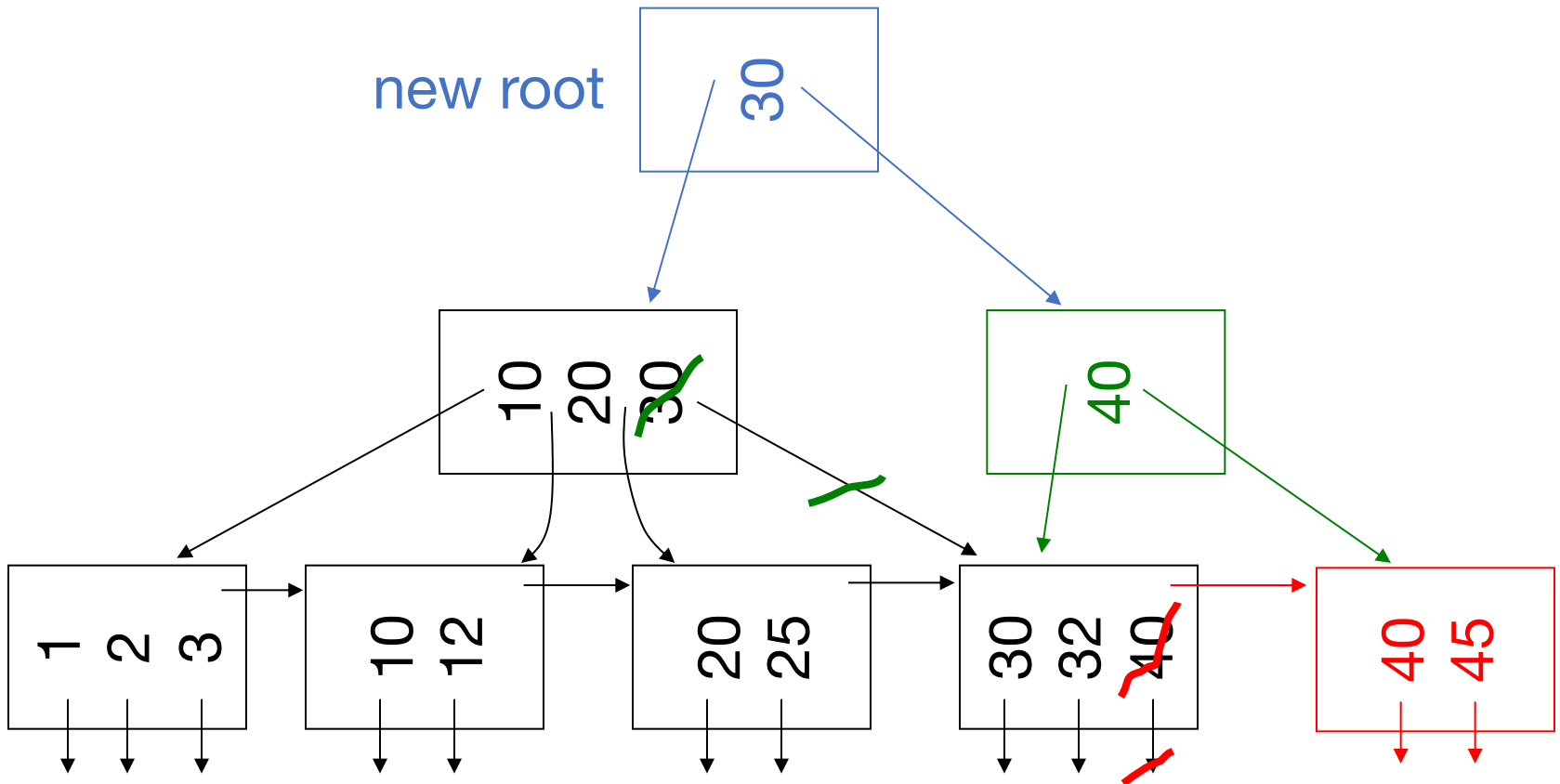
(d) New root, insert 45

n=3



(d) New root, insert 45

n=3

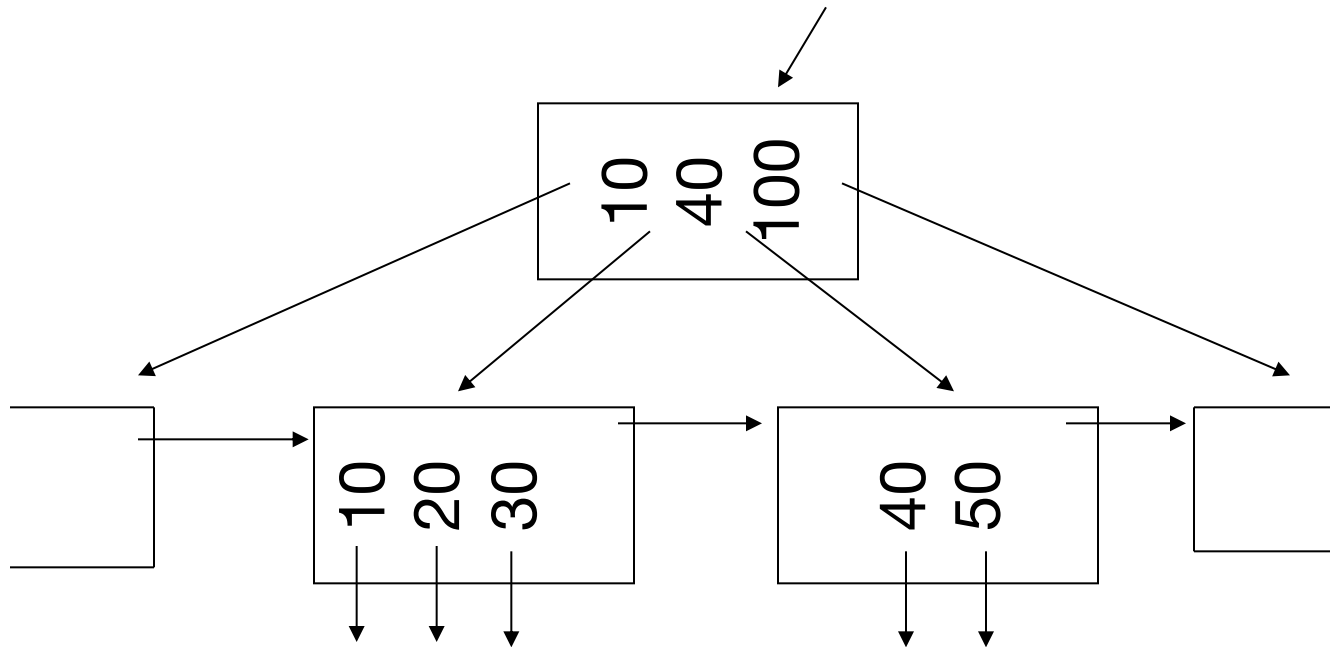


Deletion from B+tree

- (a) Simple case: no example
- (b) Coalesce with neighbor (sibling)
- (c) Re-distribute keys
- (d) Cases (b) or (c) at non-leaf

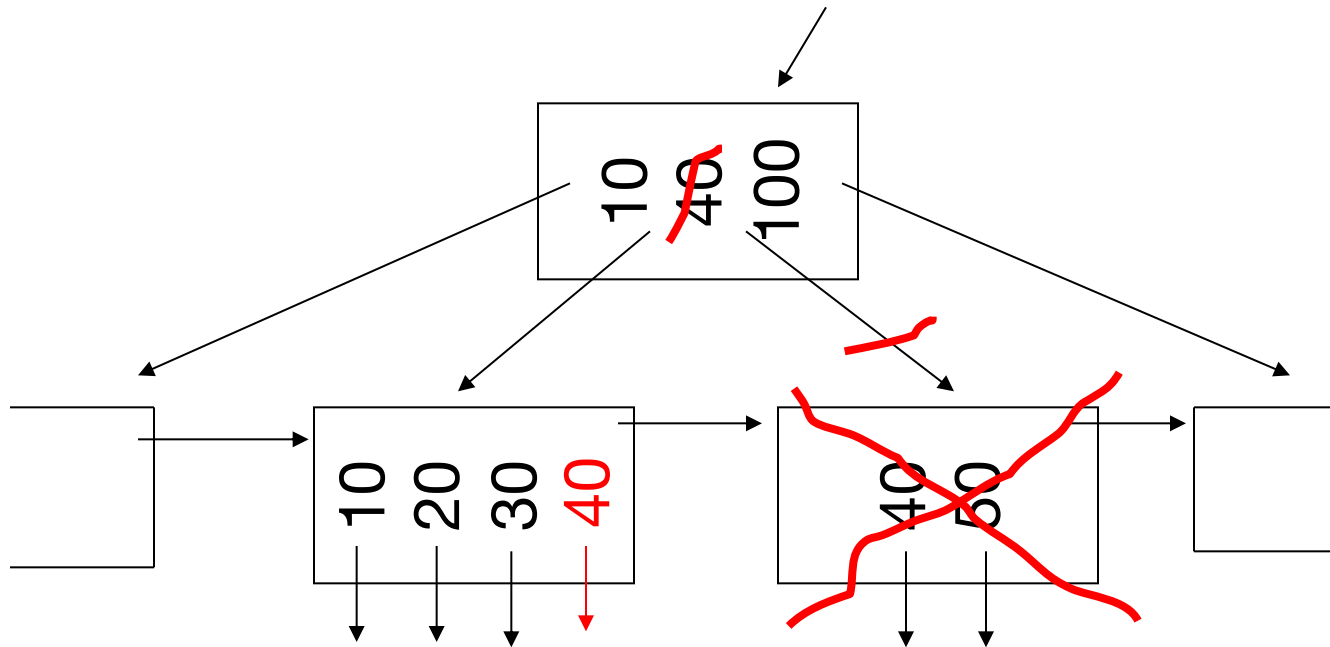
(b) Coalesce with sibling
» Delete 50

n=4



(b) Coalesce with sibling
» Delete 50

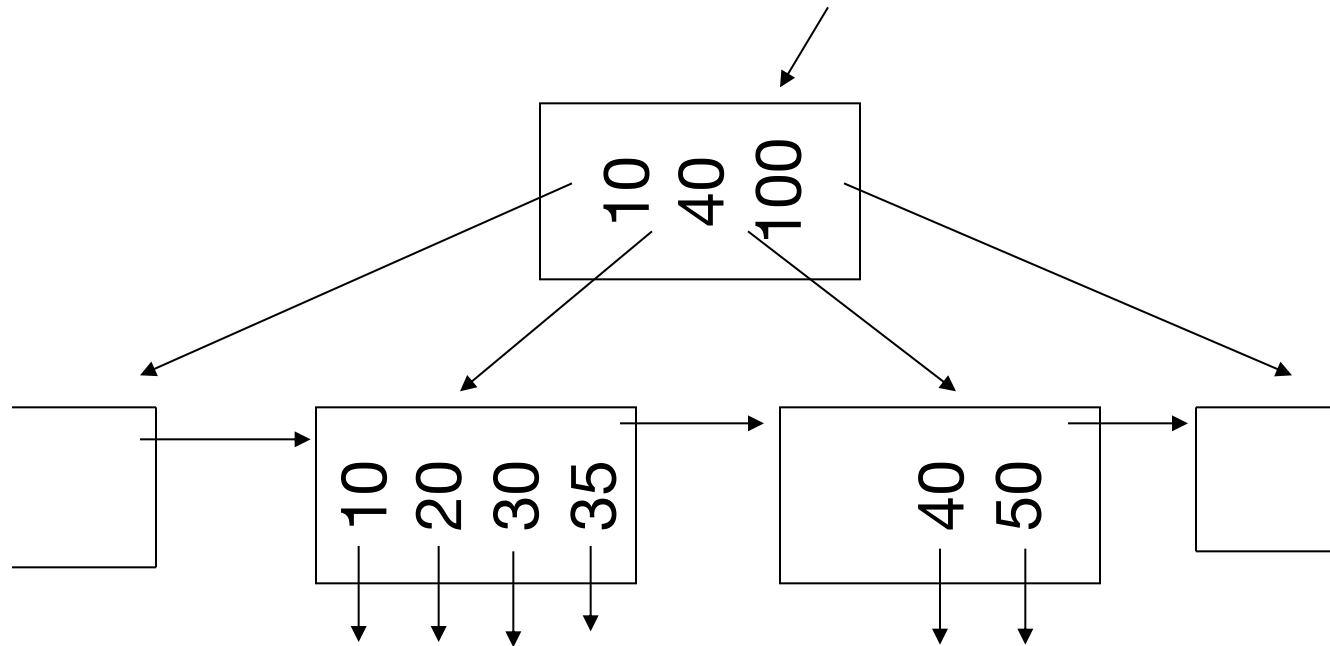
n=4



(c) Redistribute keys

» Delete 50

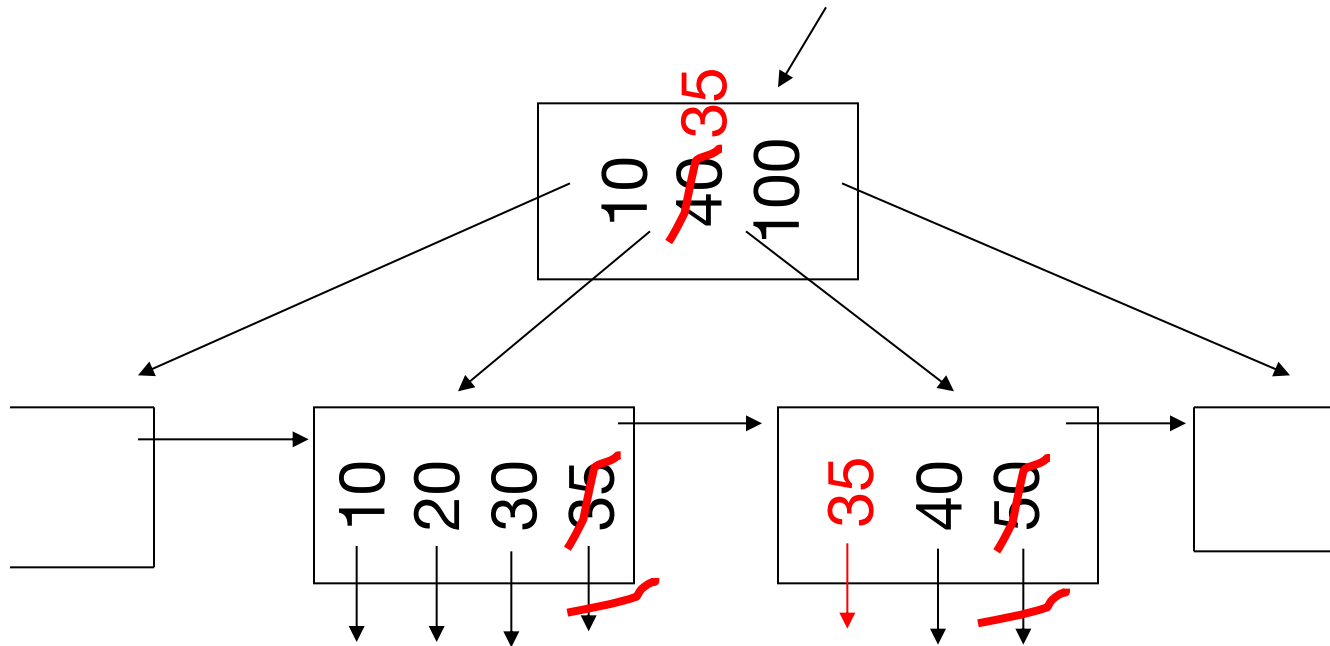
n=4



(c) Redistribute keys

» Delete 50

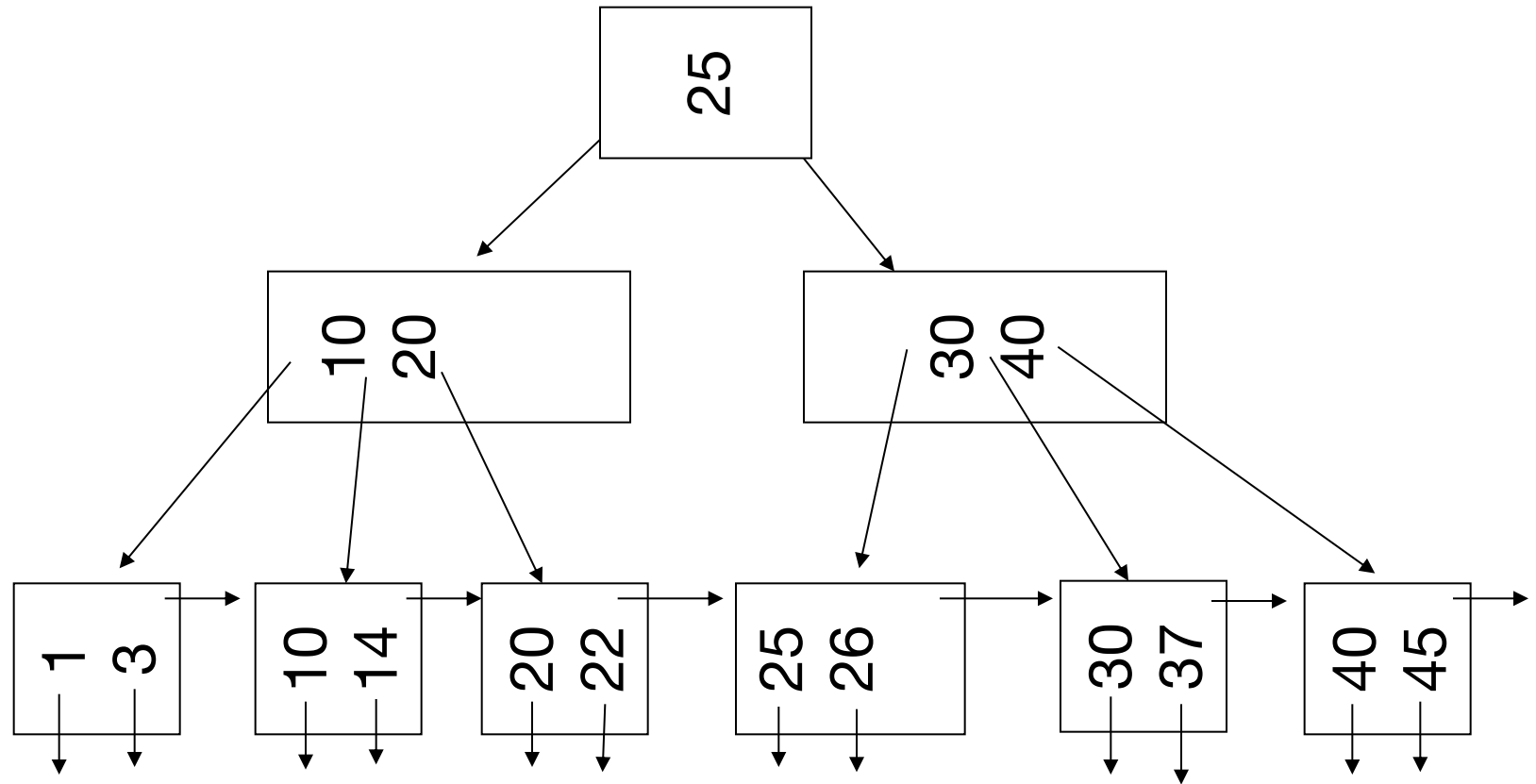
n=4



(d) Non-leaf coalesce

– Delete 37

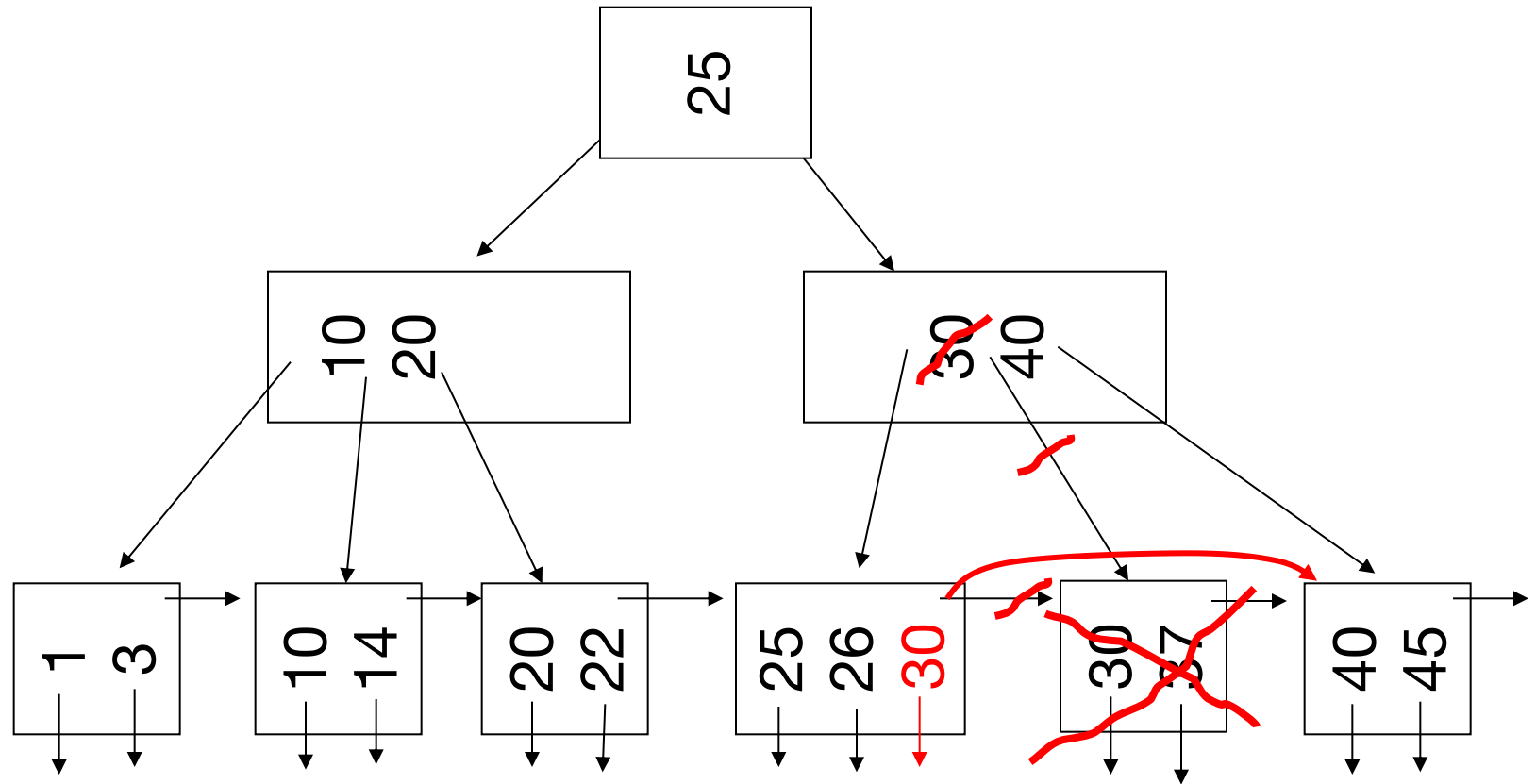
n=4



(d) Non-leaf coalesce

– Delete 37

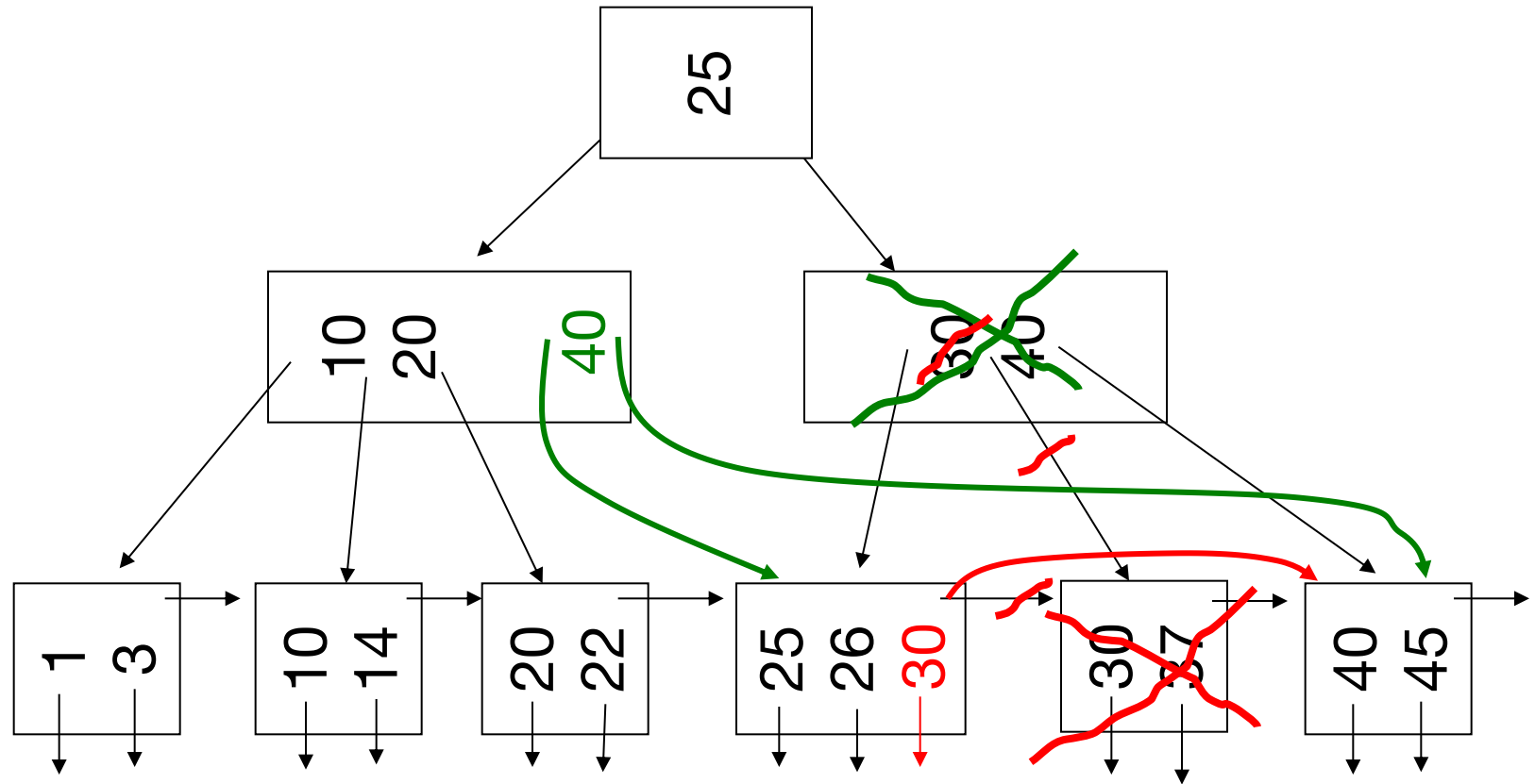
n=4



(d) Non-leaf coalesce

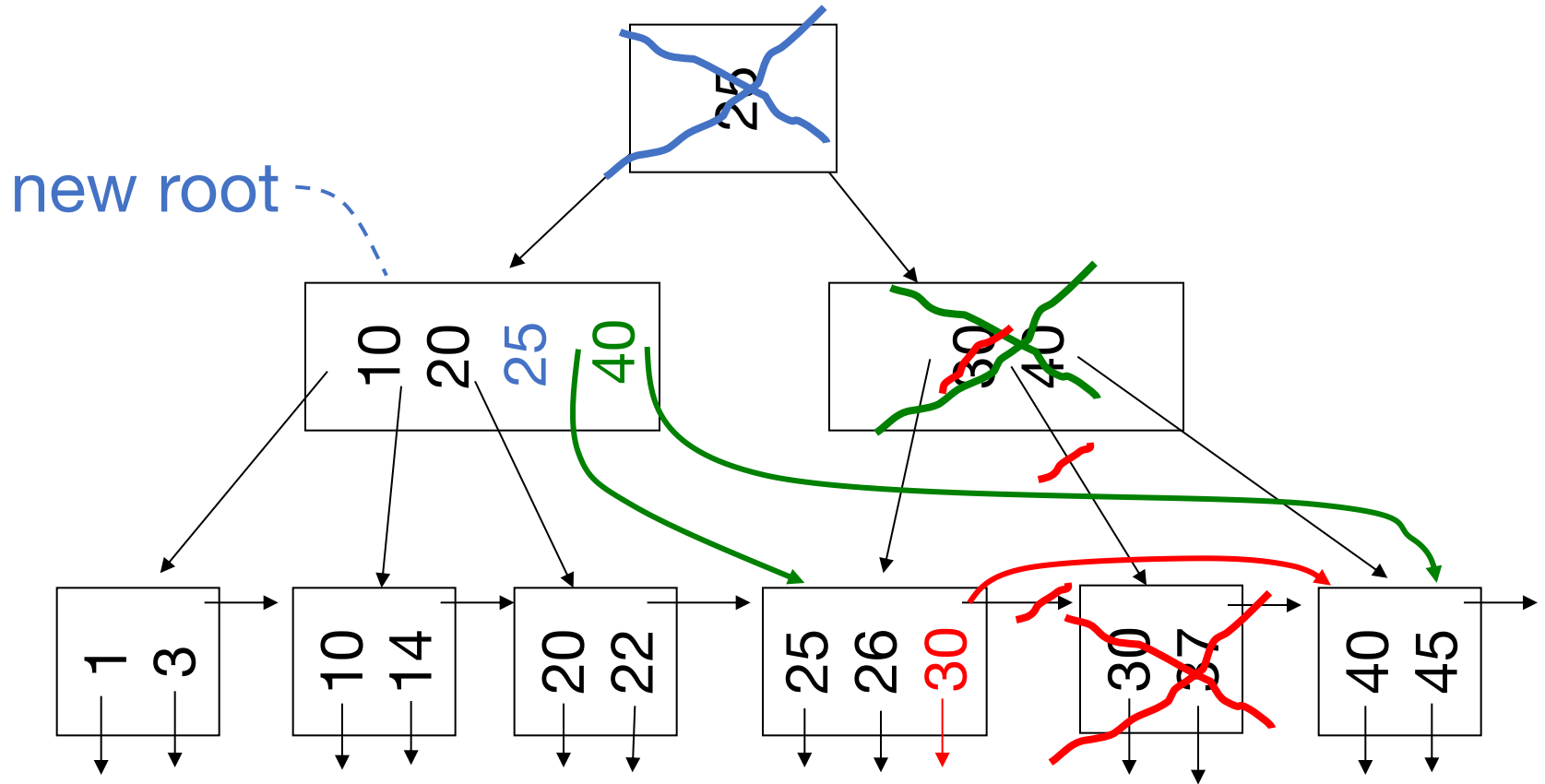
– Delete 37

n=4



n=4

n=4



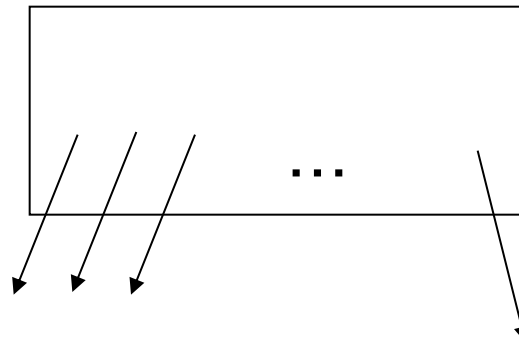
B+ Tree Deletion in Practice

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

Some Types of Indexes

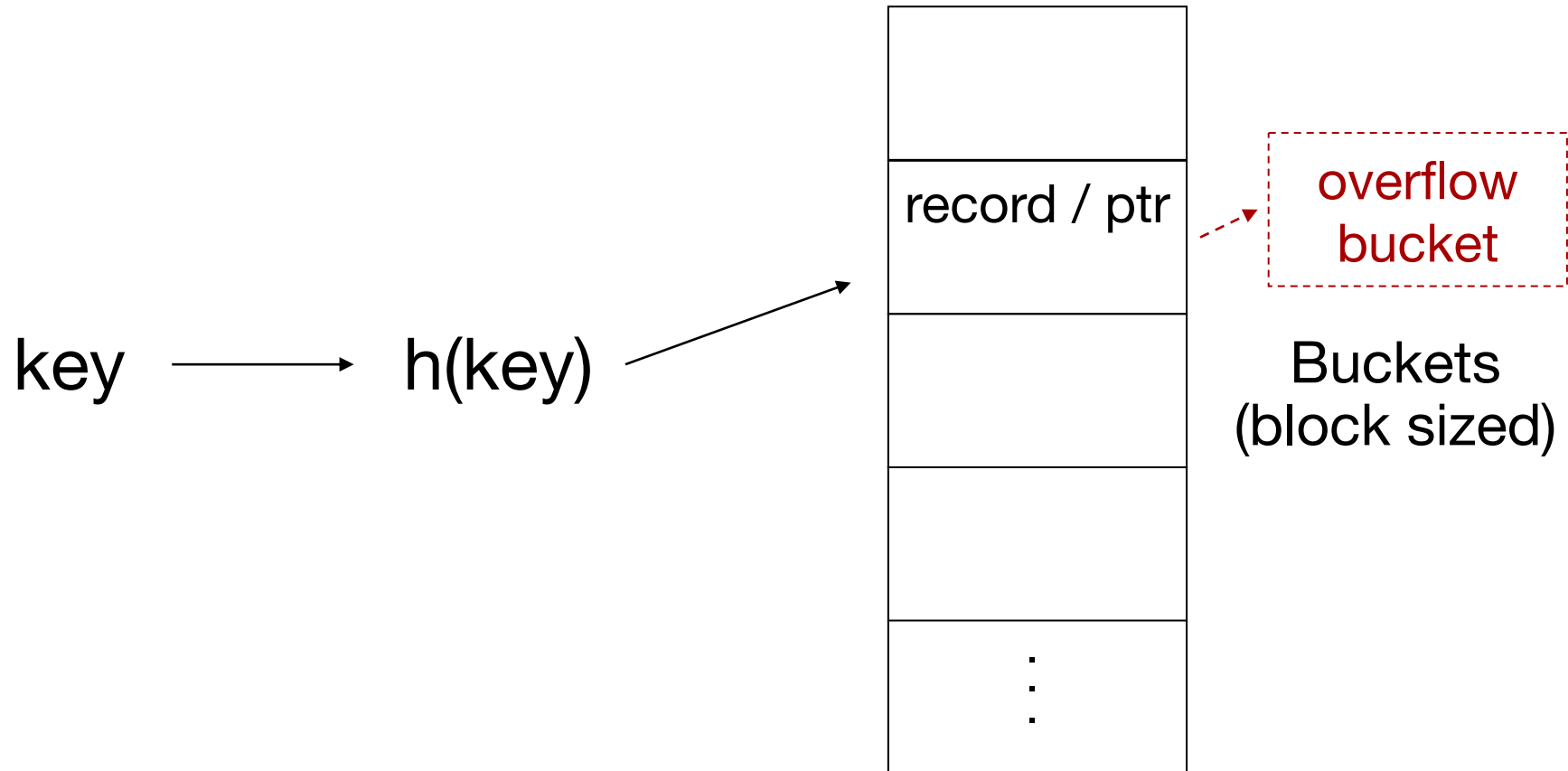
Conventional indexes

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

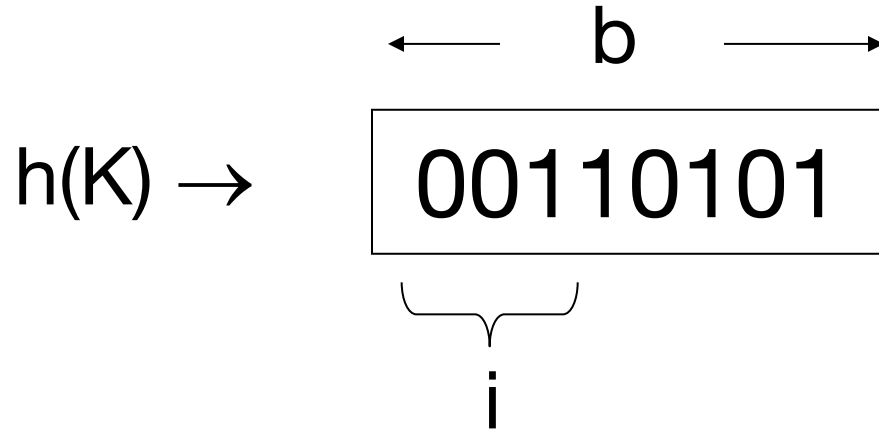
- » **On disk:** moving everything is expensive!

Extendible Hashing

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

Extendible Hashing: 2 Ideas

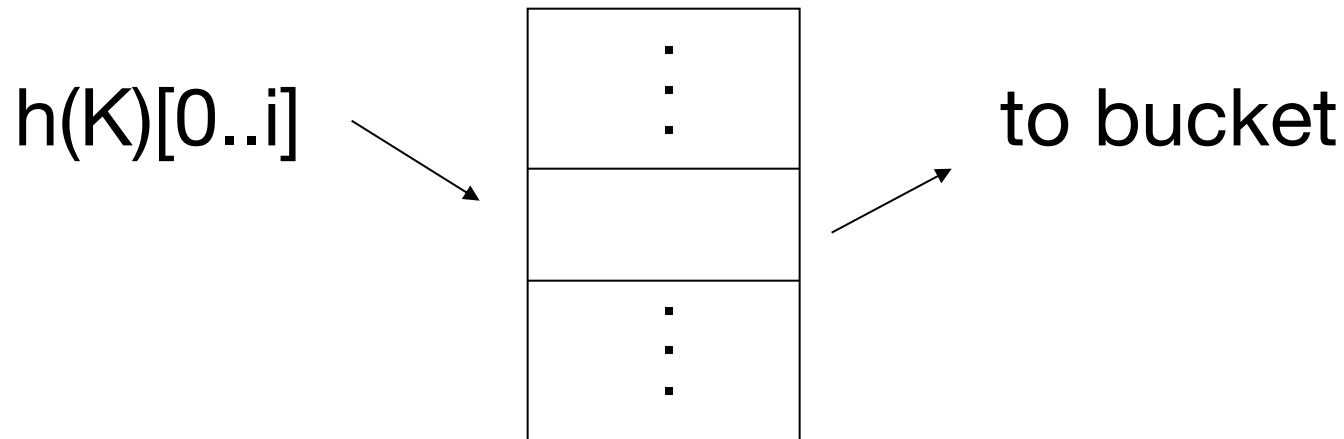
(a) Use i of b bits output by hash function



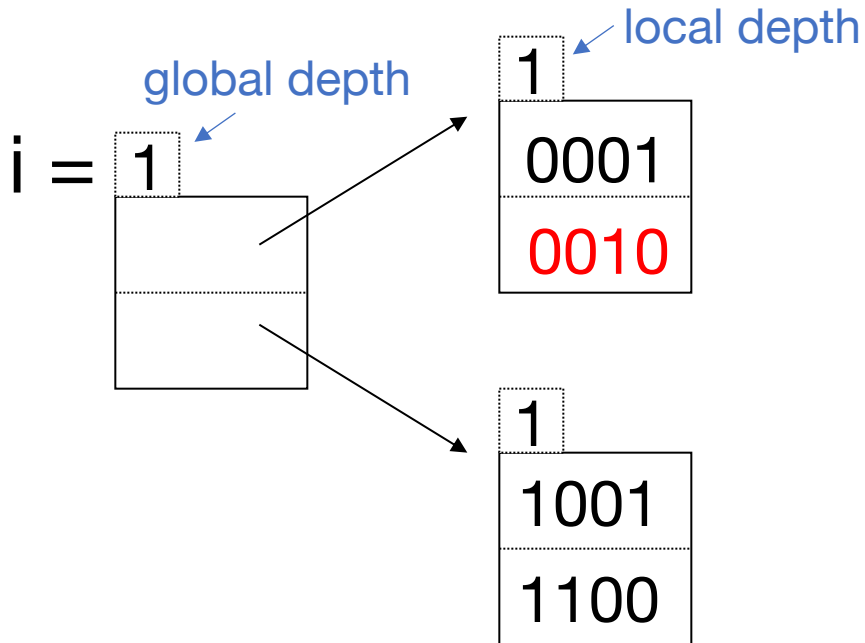
i will grow over time; the first i bits of each key's hash are used to map it to a bucket

Extendible Hashing: 2 Ideas

(b) Use a directory with pointers to buckets

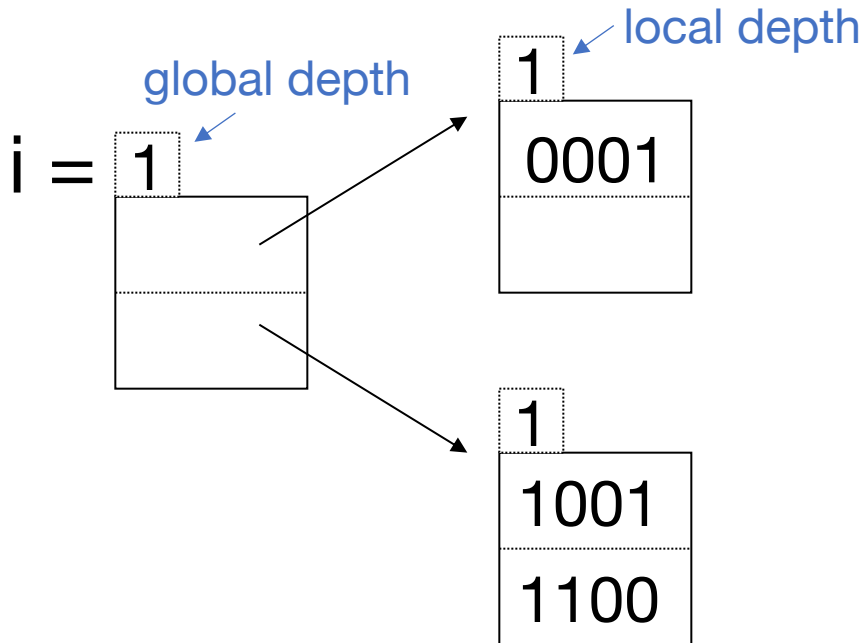


Example: 4-bit $h(K)$, 2 keys/bucket



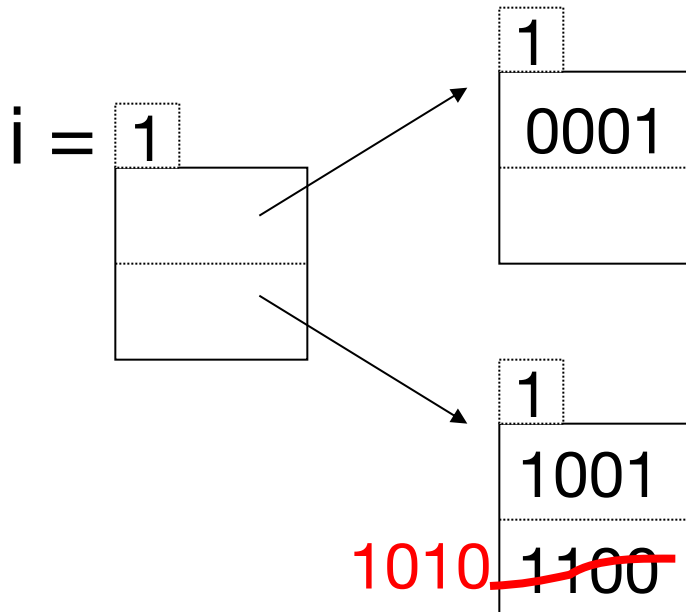
Insert 0010

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

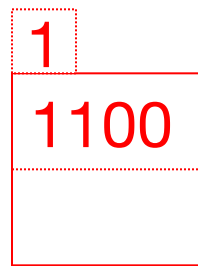


Insert 1010

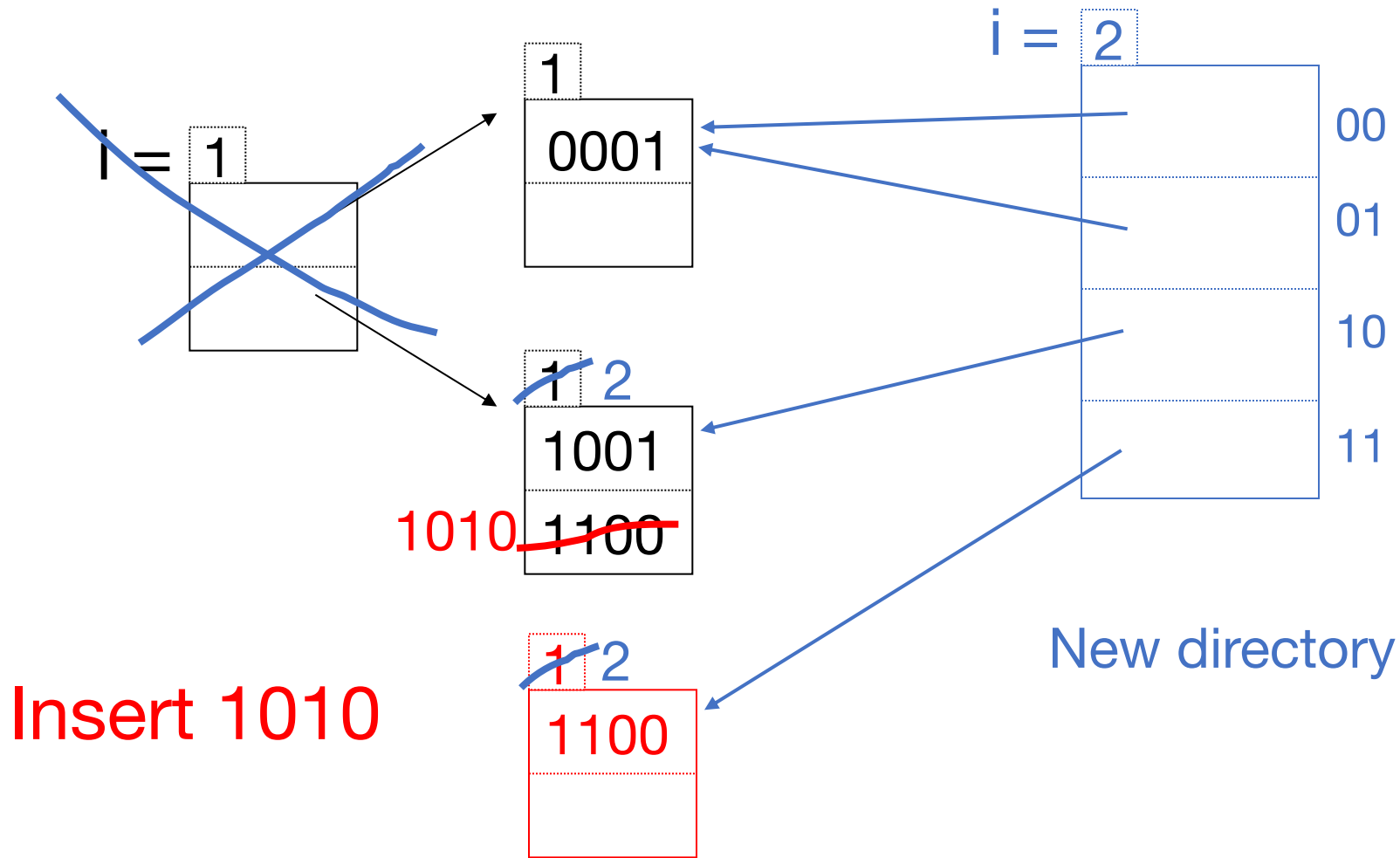
Example: 4-bit $h(K)$, 2 keys/bucket



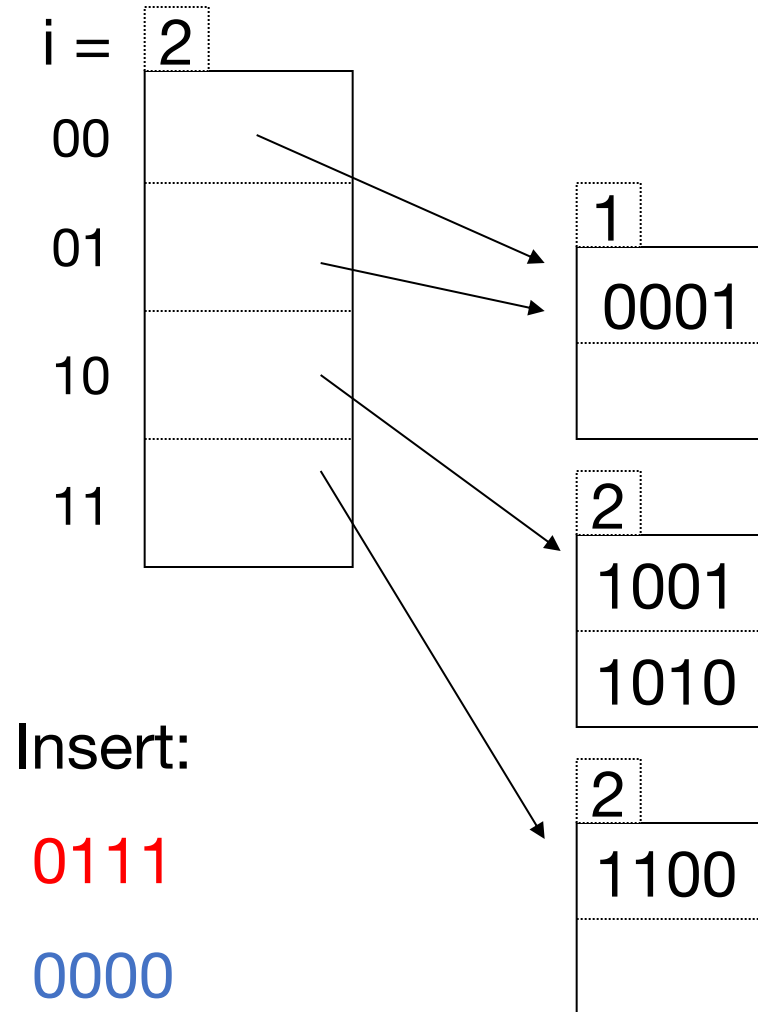
Insert 1010



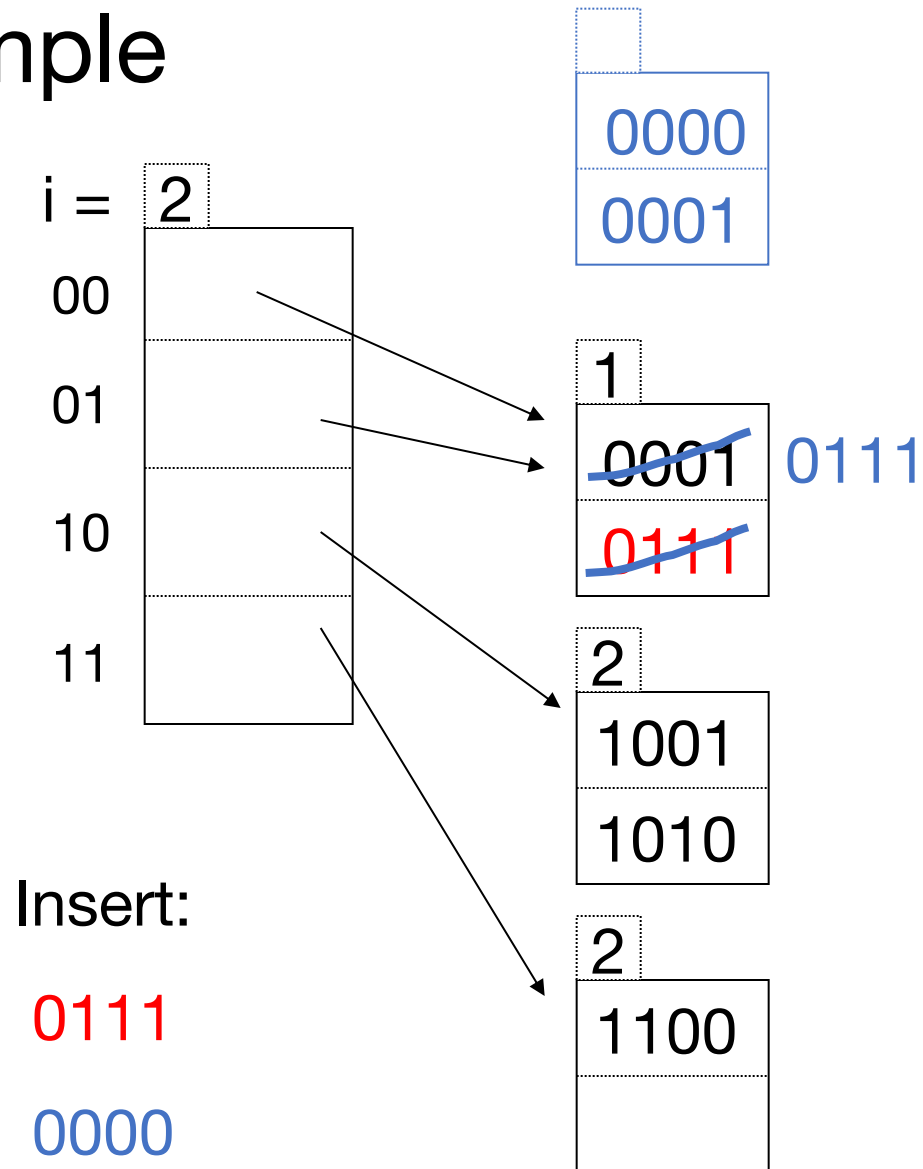
Example: 4-bit $h(K)$, 2 keys/bucket



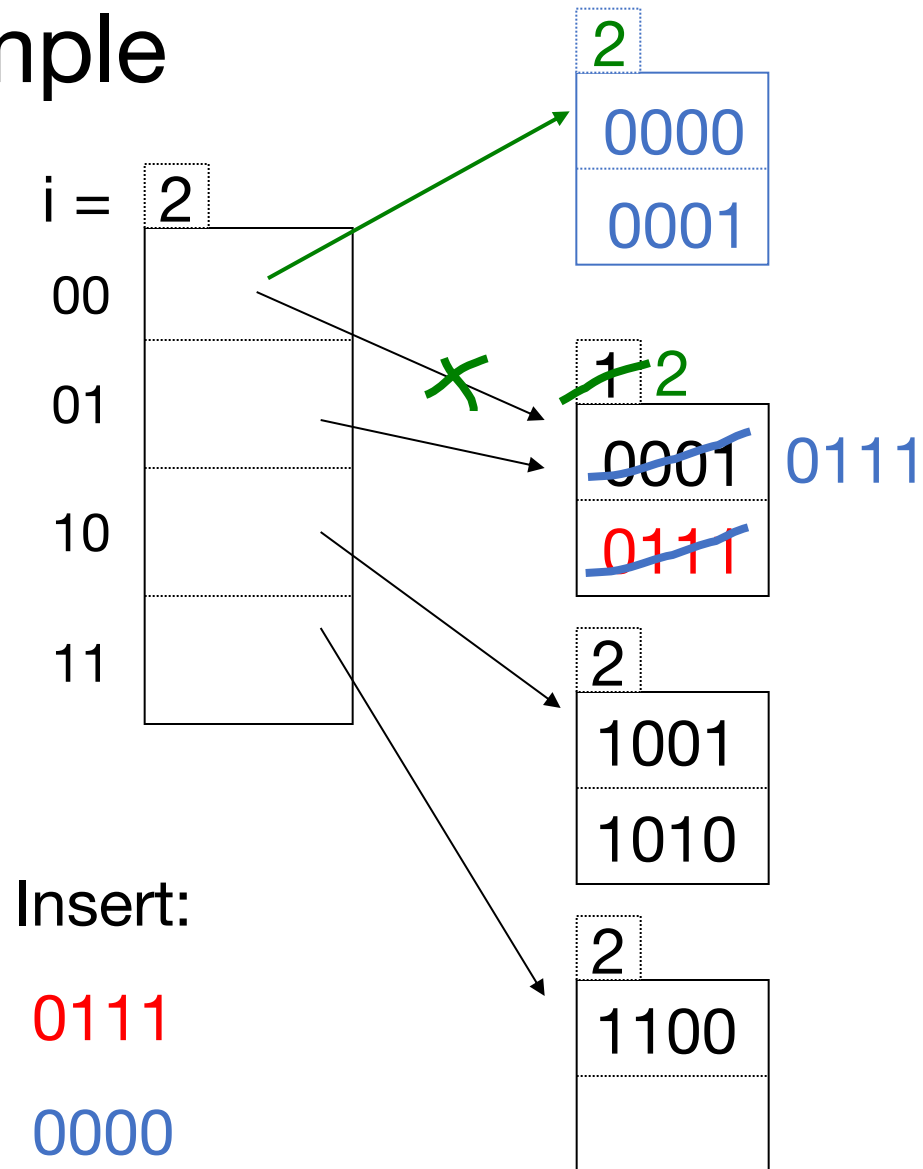
Example



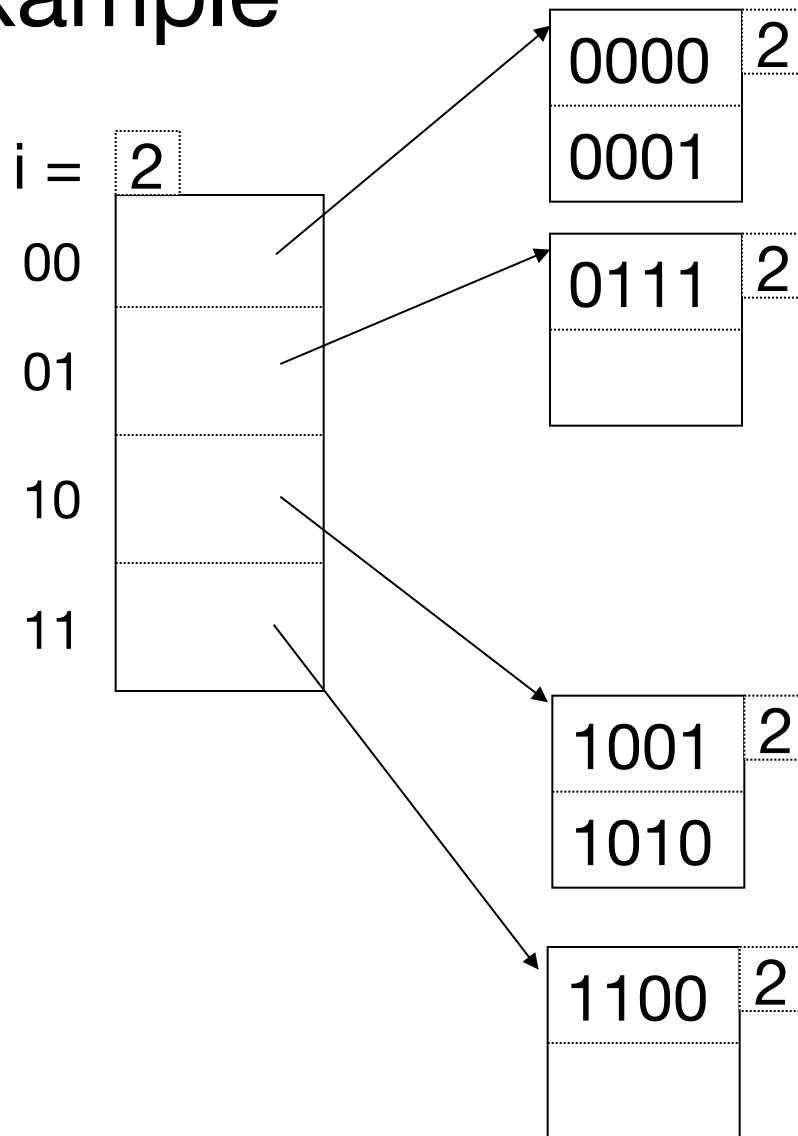
Example



Example



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

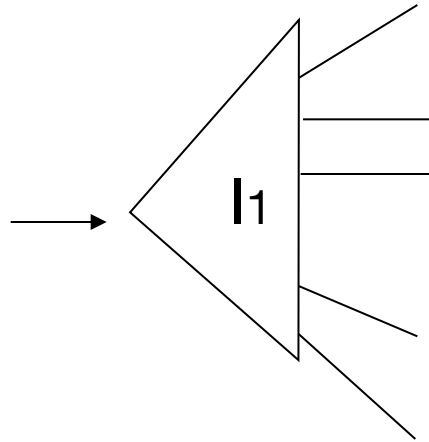
Example: find records where

DEPT = “Toy” AND SALARY > 50k

Strategy I:

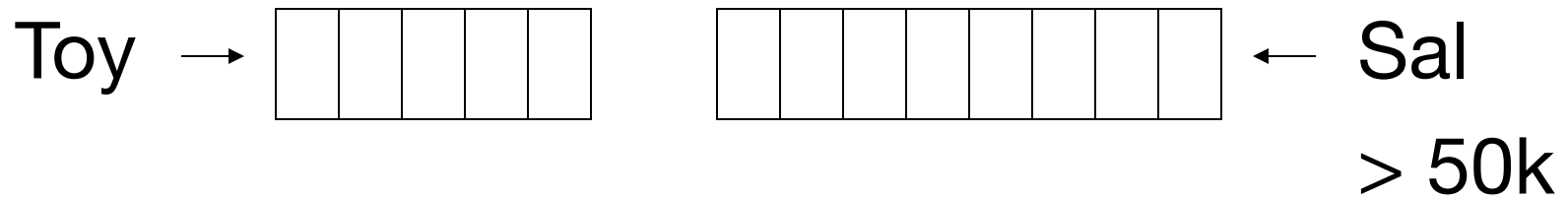
Use one index, say Dept.

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



Strategy II:

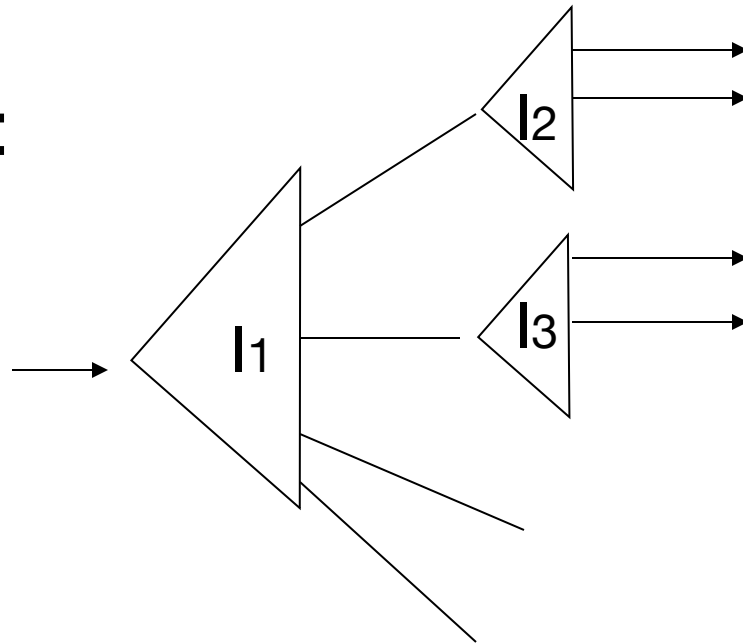
Use 2 indexes; manipulate pointers



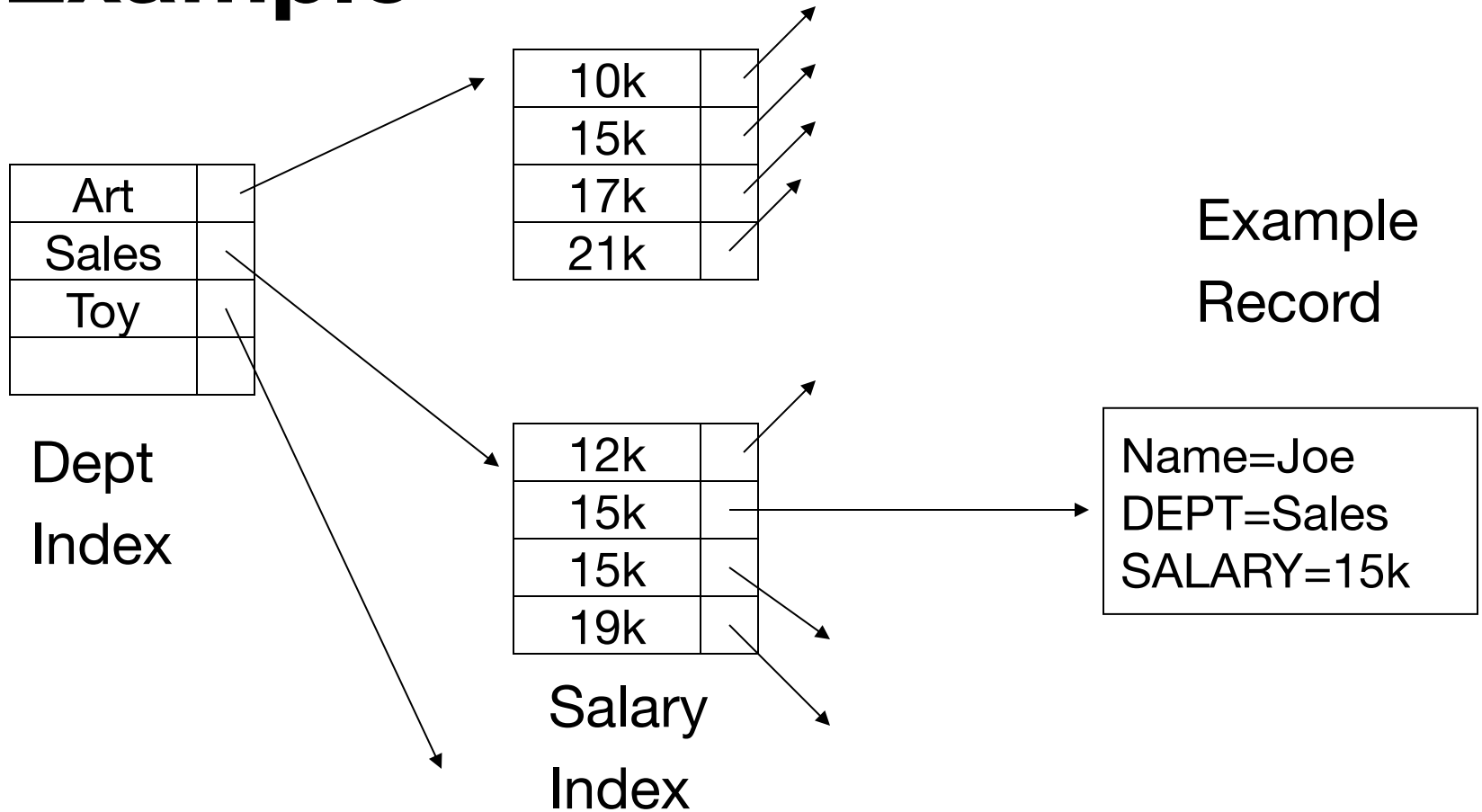
Strategy III:

Multi-key index

One idea:

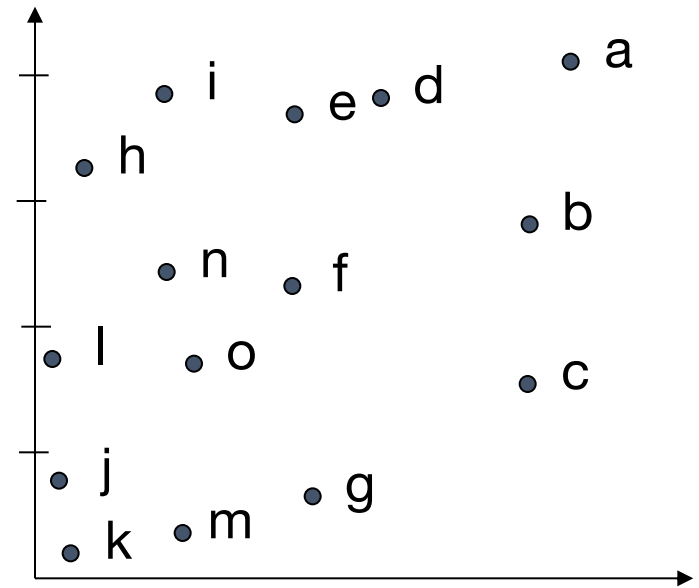


Example

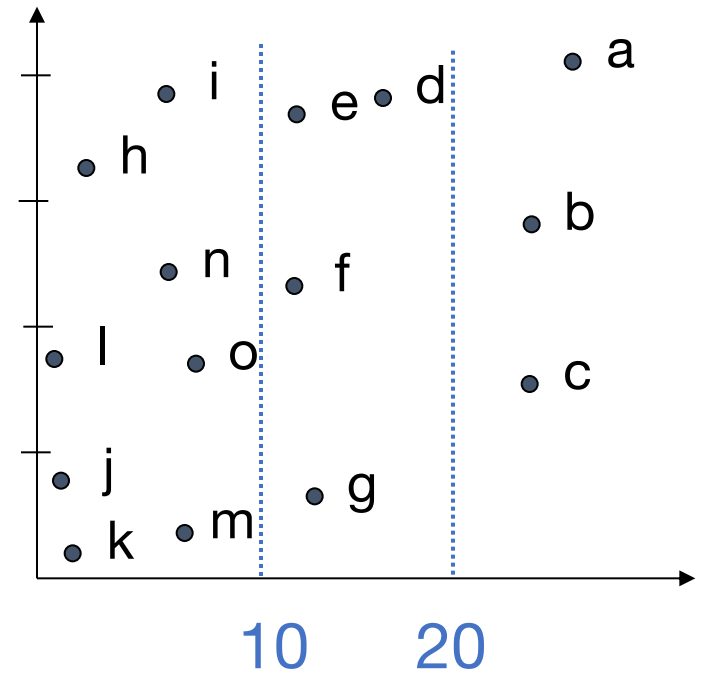
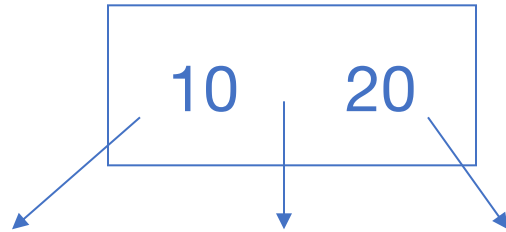


k-d Tree

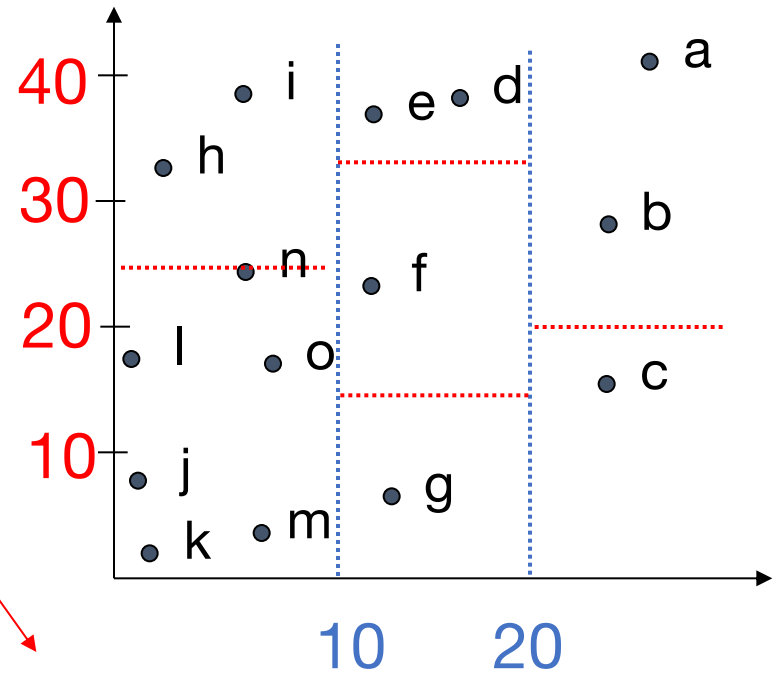
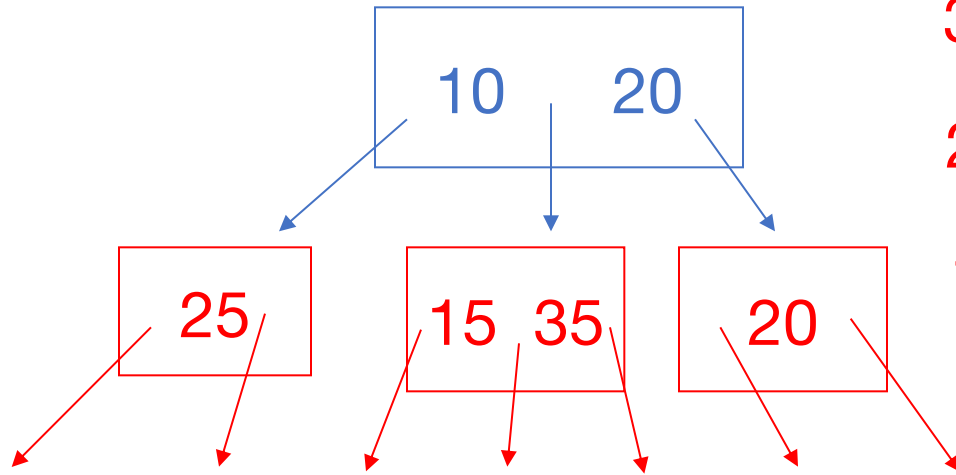
Splits dimensions in any order
to hold k-dimensional data



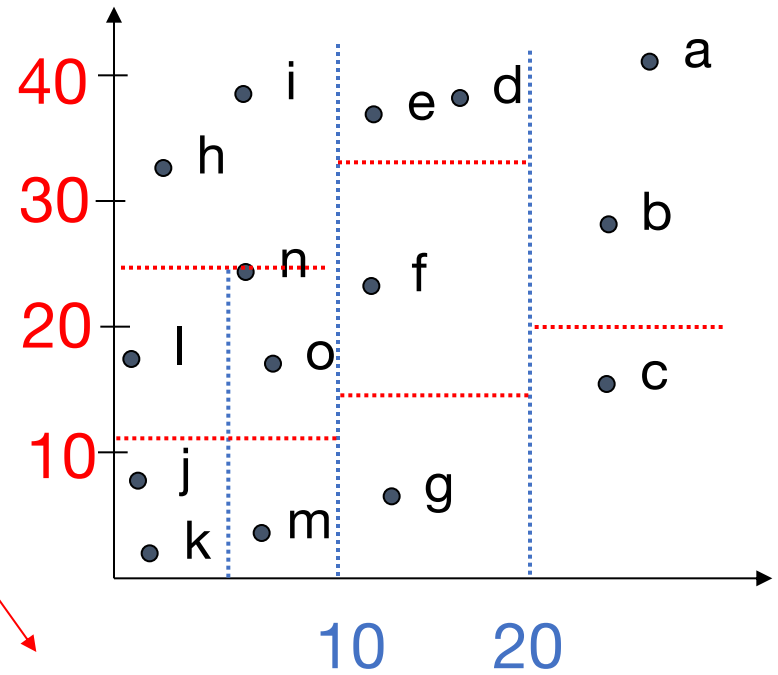
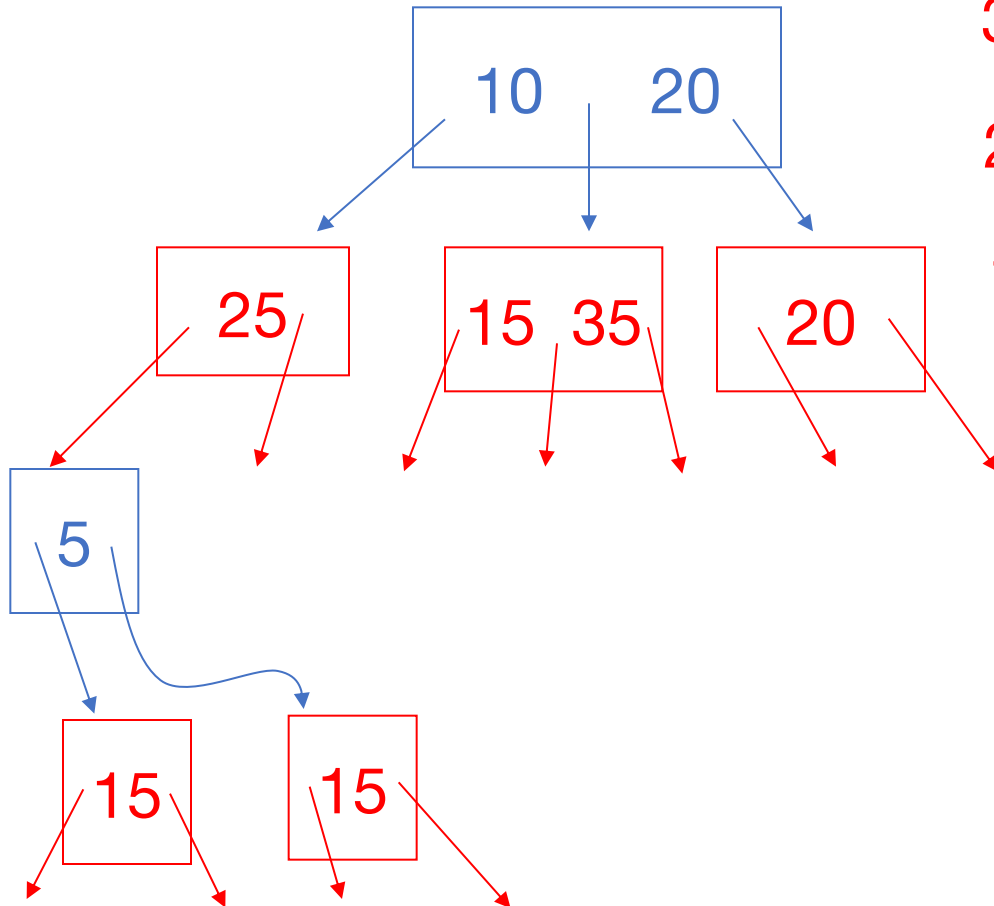
k-d Tree



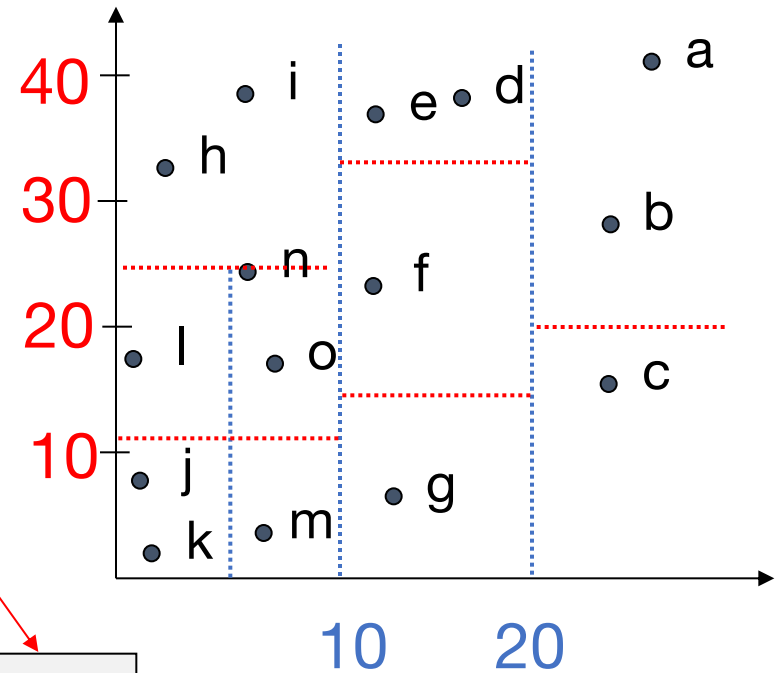
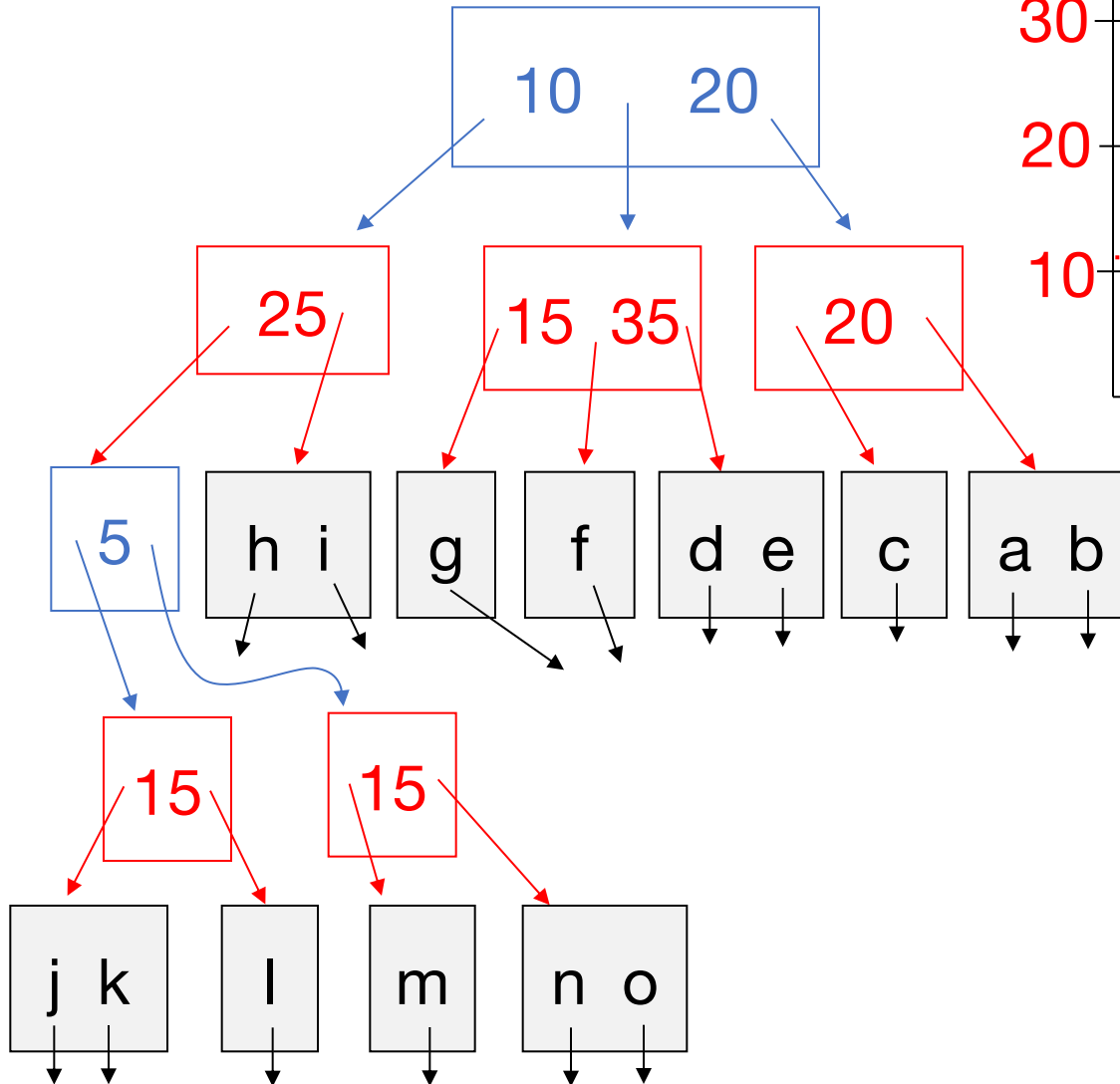
k-d Tree



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?

Storage System Examples

MySQL: transactional DBMS

- » Row-oriented storage with 16 KB pages
- » Variable length records with headers, overflow
- » Index types:
 - B-tree
 - Hash (in memory only)
 - R-tree (spatial data)
 - Inverted lists for full text search
- » Can compress pages with Lempel-Ziv

Storage System Examples

- Apache Parquet + Hive:** analytical data lake
- » Column-oriented storage as set of ~1 GB files (each file has a slice of all columns)
 - » Various compression and encoding schemes at the level of pages in a file
 - Special scheme for nested fields (Dremel)
 - » Header with statistics at the start of each file
 - Min/max of columns, nulls, Bloom filter
 - » Files partitioned into directories by one key