Query Processing

Part 2: Dense and Sparse Indexes

"Field" means "attribute"

Terminology

- A *sorted* (or *sequential*) *file* is stored (on a disk) sequentially in sorted order
  - We sort a file on a *sequence* (or *sort*) *field*
  - Can also sort lexicographically on several fields
- A *heap* is a file that is stored on a disk in no particular order
- A *search key* is a field(s) on which we can search efficiently for records with a given key value
  - Search keys are implemented by dense & sparse indexes, B+trees, hash tables

Two Types of Search Keys

- A file is organized according to a *primary search key* that
  - Determines the location of a record in the file
  - Is used for insertions, deletions and updates
  - Is usually called *primary key*, although it does not necessarily define a record uniquely
- A *secondary search key* is used only for searching (usually called *secondary key*)

In the context of query processing, "key" usually means "search key," and it is *not* a key in the sense of FDs, namely, duplicate values are possible

Dense and Sparse Indexes for Primary Keys

Sequential File

10	
20	

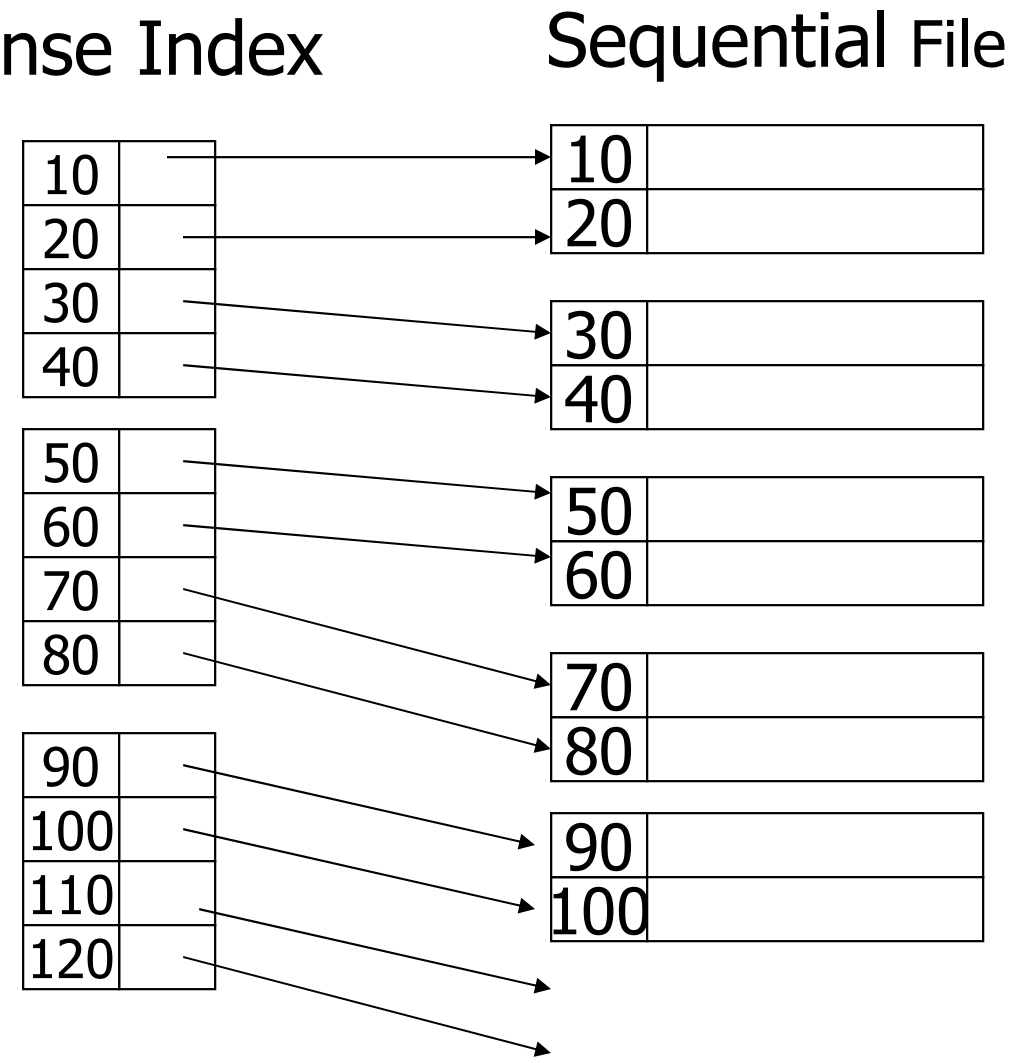
30	
40	

50	
60	

70	
80	

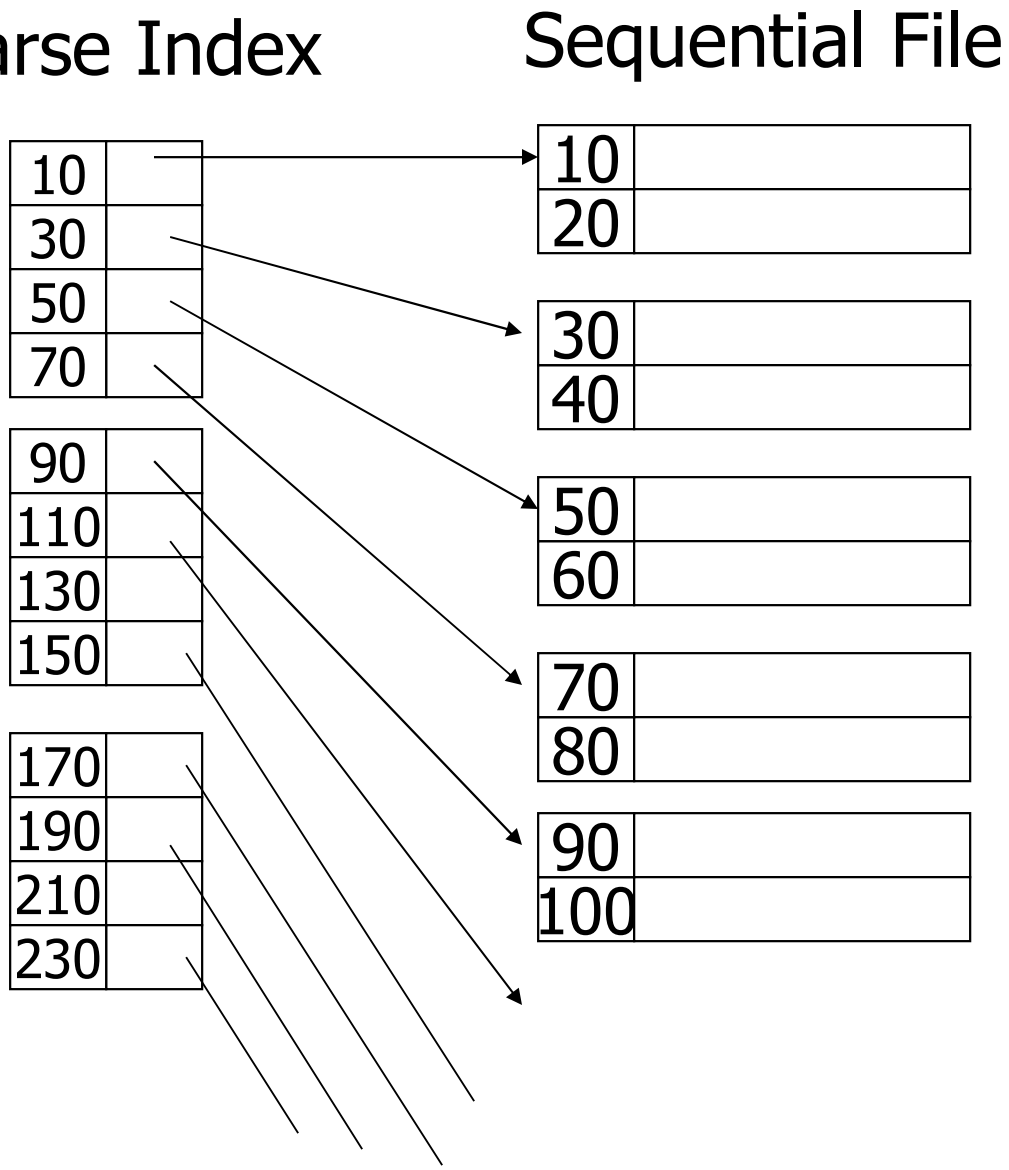
90	
100	

An index entry for each record of the file



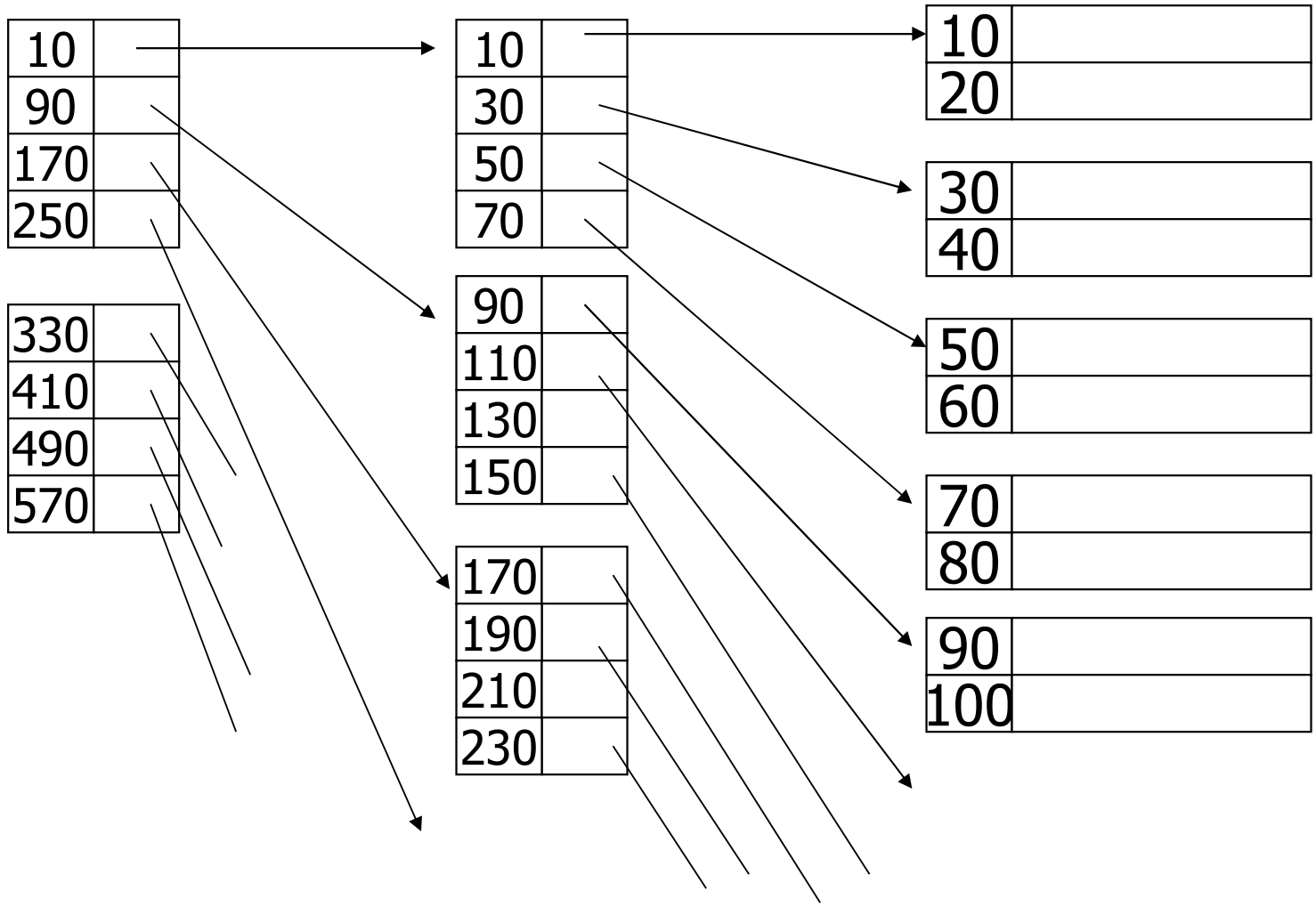
Only one index entry for each block (for the block's first value)

Given *V*, follow the pointer for the largest *K*, s.t. $K \leq V$



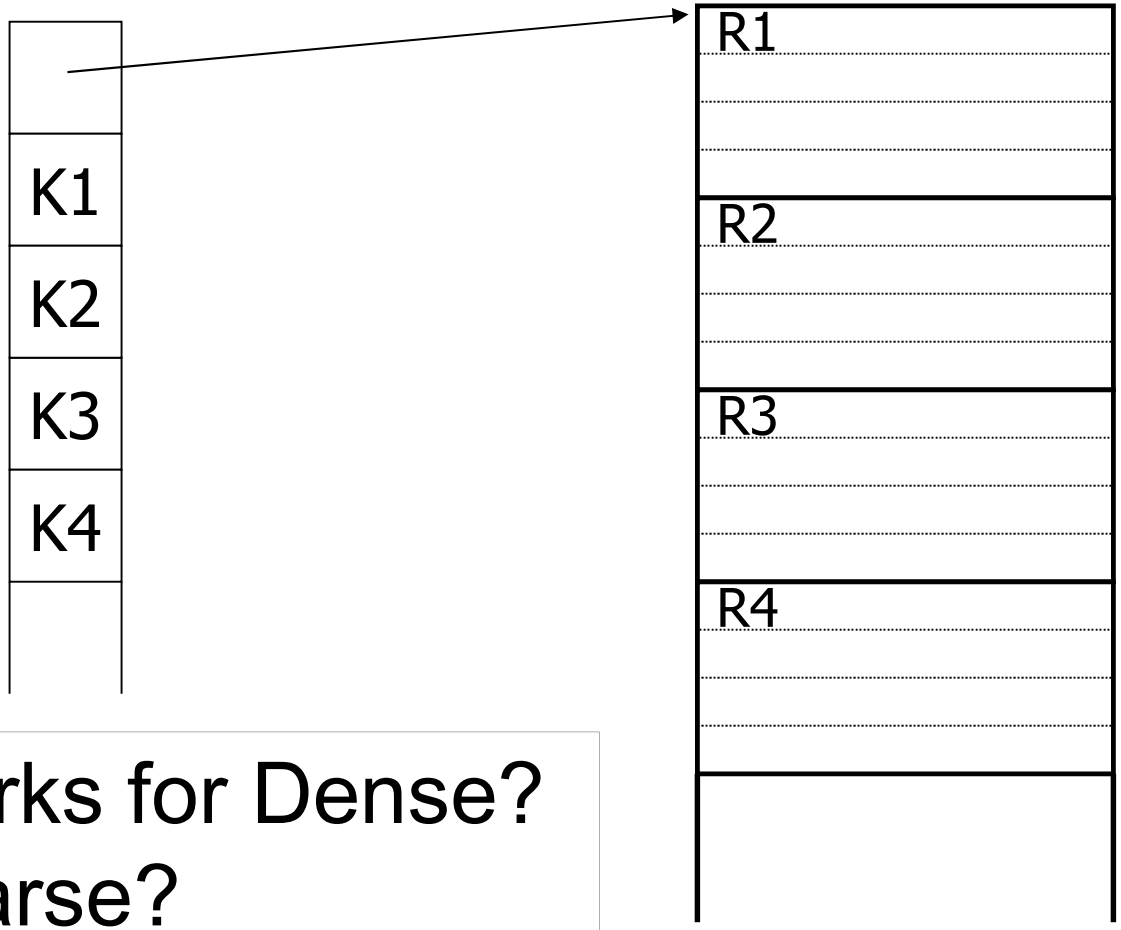
Sparse 2nd level

Sequential File



Comments & Questions

- The index blocks are not necessarily contiguous, but they are chained in both directions
- Same for the blocks of a file
- Can a heap have a sparse index?
- Should we sometimes use a dense index in the second (or higher) level?
- If the file is contiguous, we can compute the pointers, instead of storing them in the index

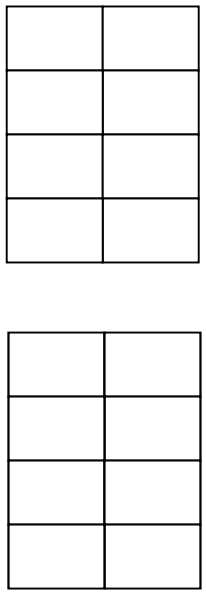


A block has 1024 Bytes

Works for Dense? Sparse?

Find the K3 block by computing its offset:
(3-1)1024 = 2048 bytes

Duplicate Keys in the File



10	
10	

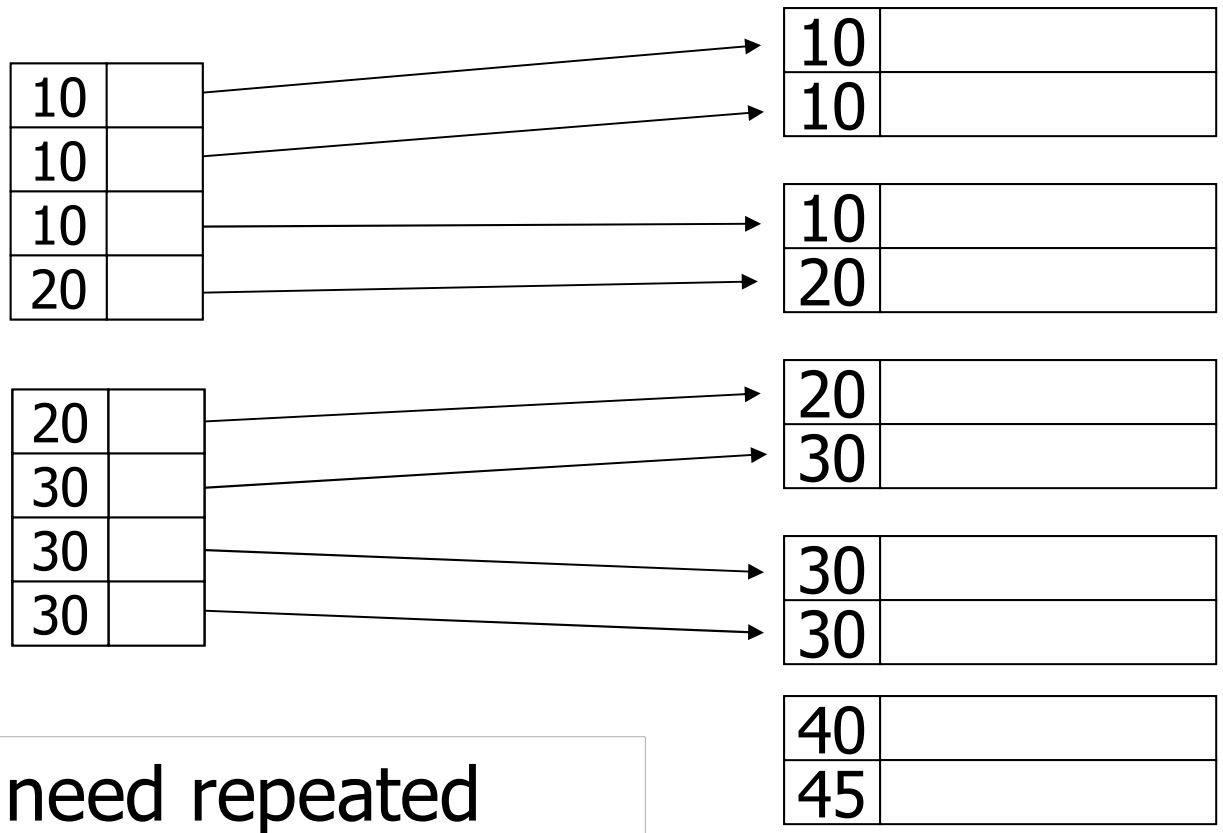
10	
20	

20	
30	

30	
30	

40	
45	

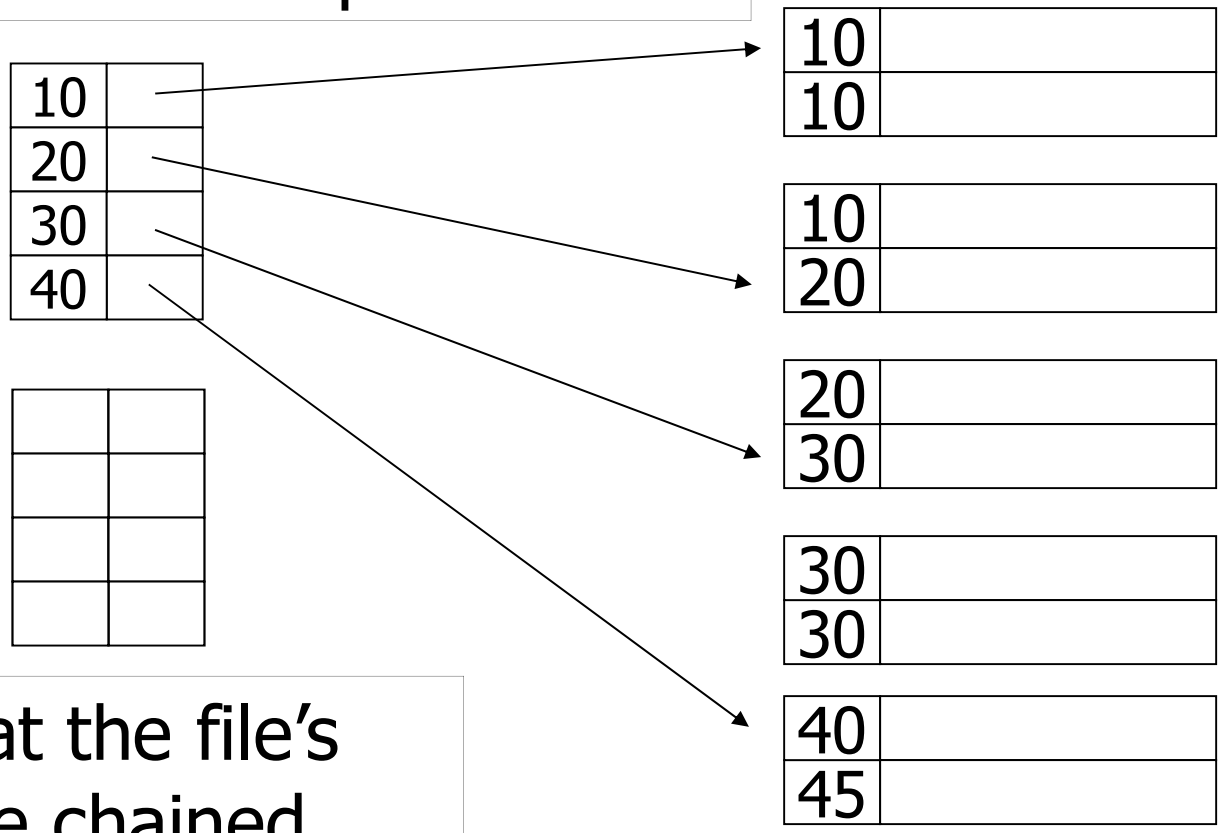
Dense Index for the File



Do we need repeated keys in the index?

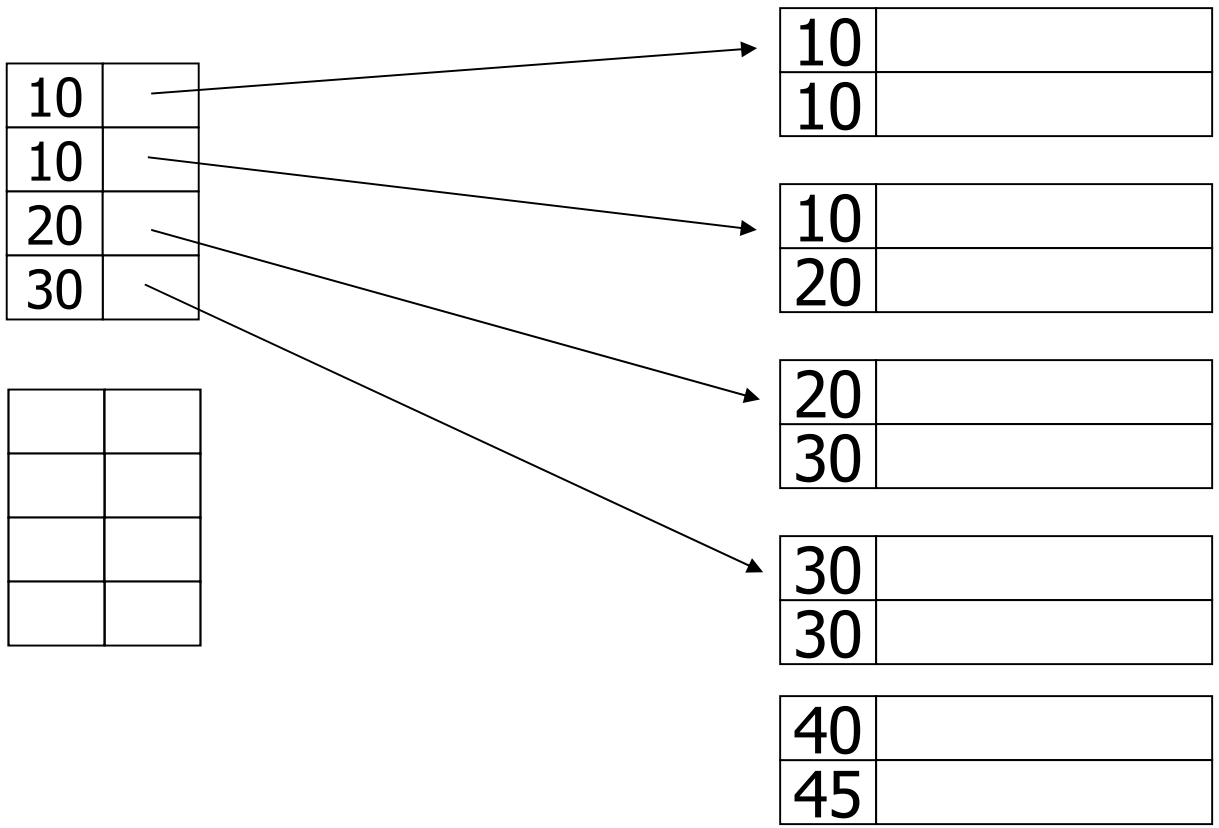
Keys Repeated in File But not in Dense Index

Now it is more like a sparse index

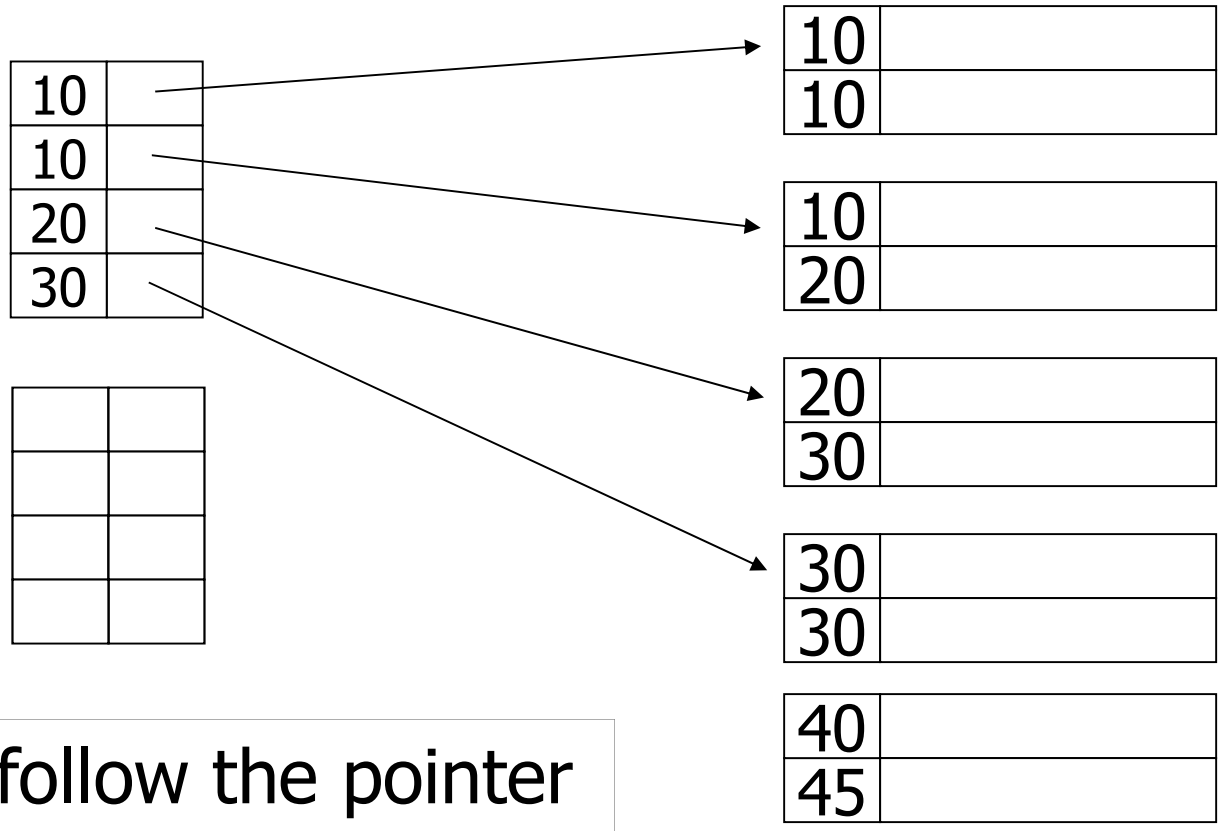


Recall that the file's blocks are chained (same for the index)

Sparse Index for a File with Duplicate Keys



How to Search?

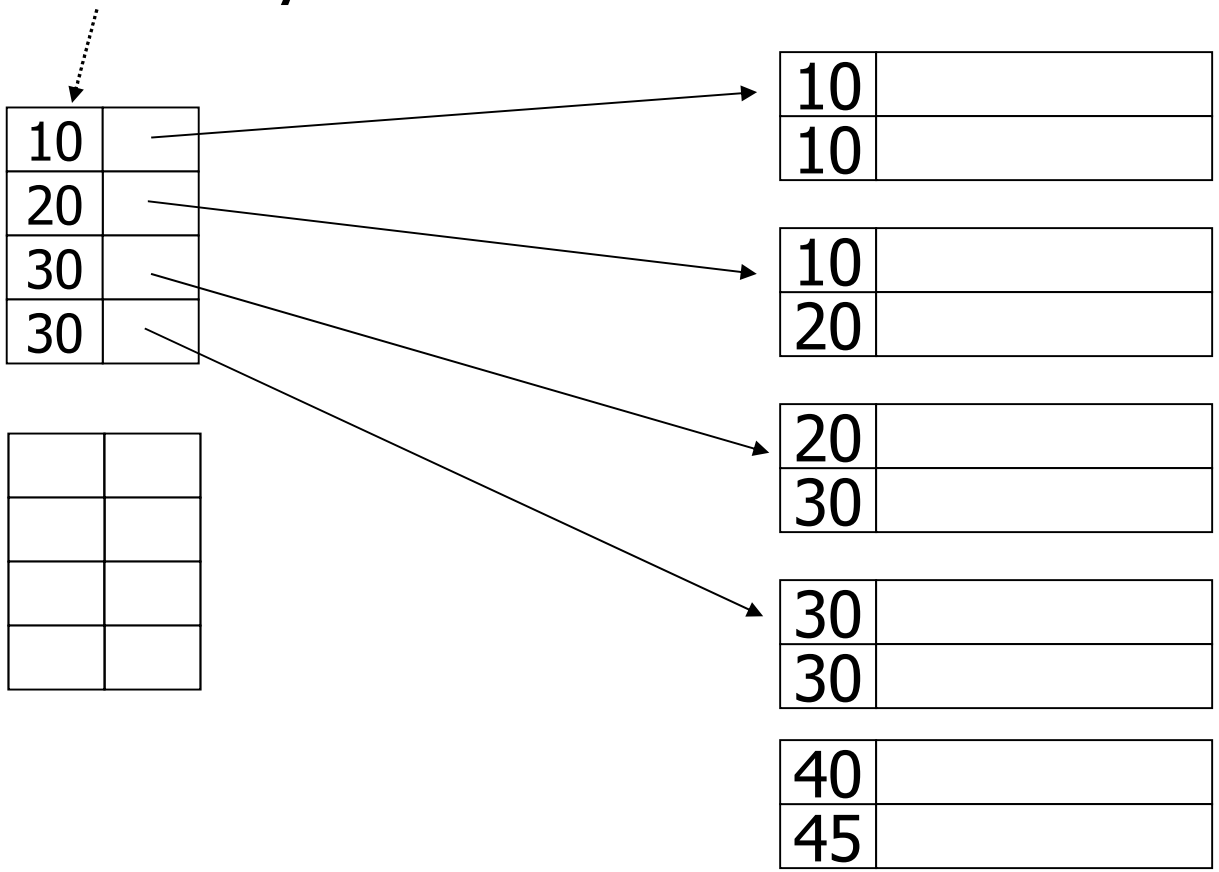


Still Valid?

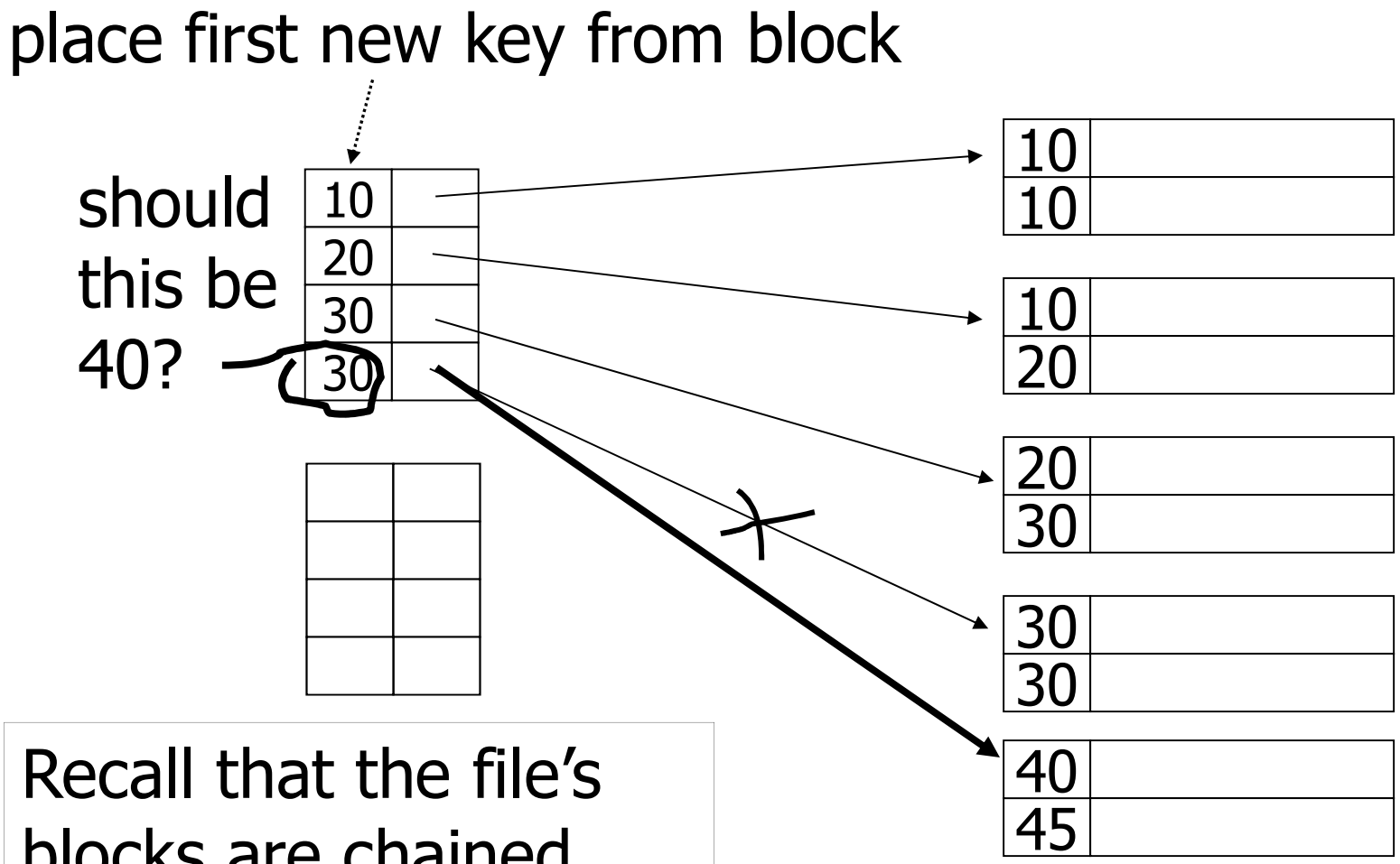
Given *V*, follow the pointer for the largest *K*, s.t. $K \leq V$

Fixing the Problem

place first new key from block



Need Repeated Keys in Index?

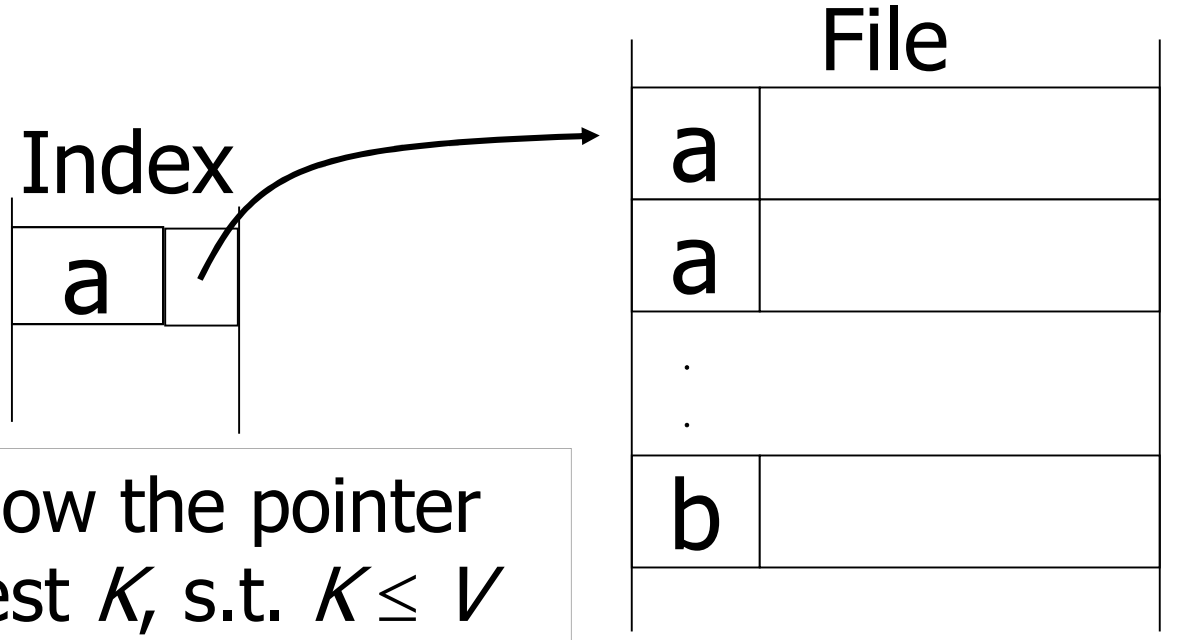


Recall that the file's blocks are chained (same for the index)

To Sum Up

- Sparse index points to a block only if it has a new value not seen before
- The smallest such value is associated with the pointer to the block

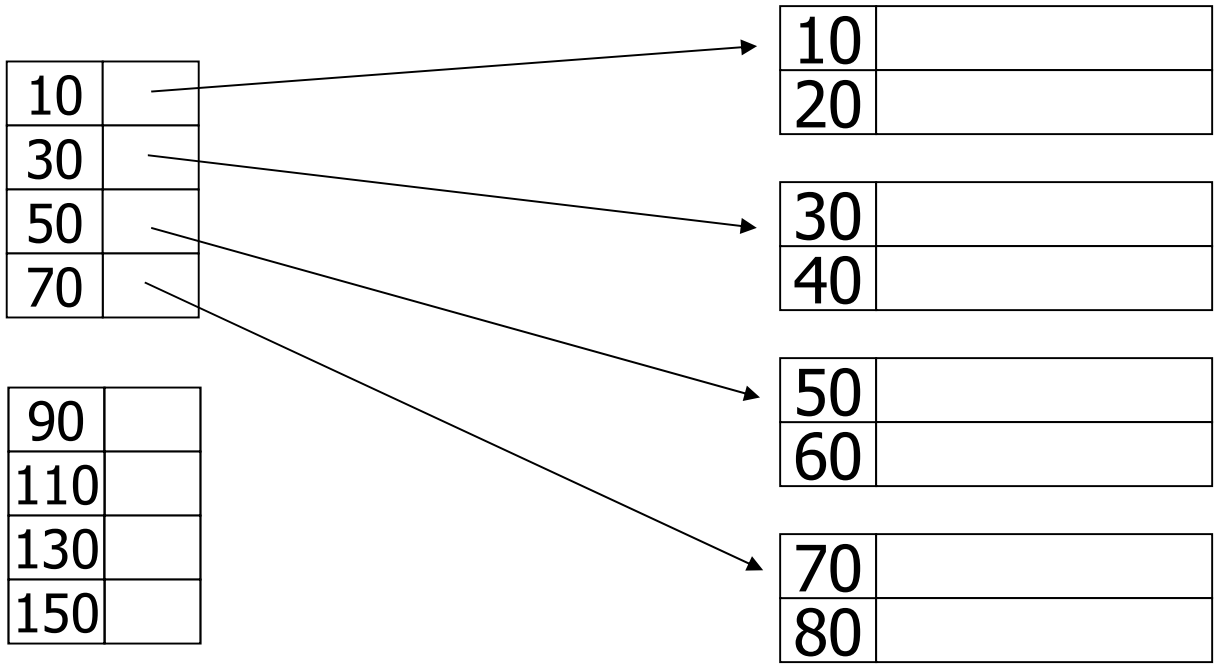
During search, when do we have to continue to the next block of the file?



✓ Given *V*, follow the pointer for the largest *K*, s.t. $K \leq V$

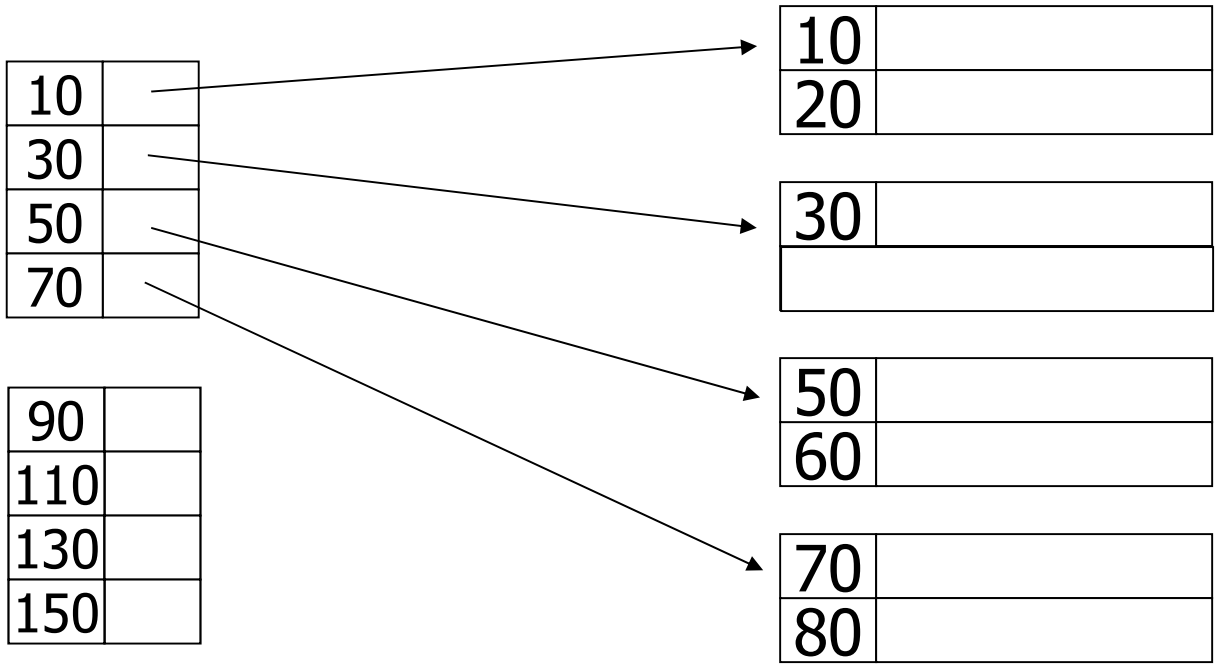
Deletion from Sparse Index

delete record 40



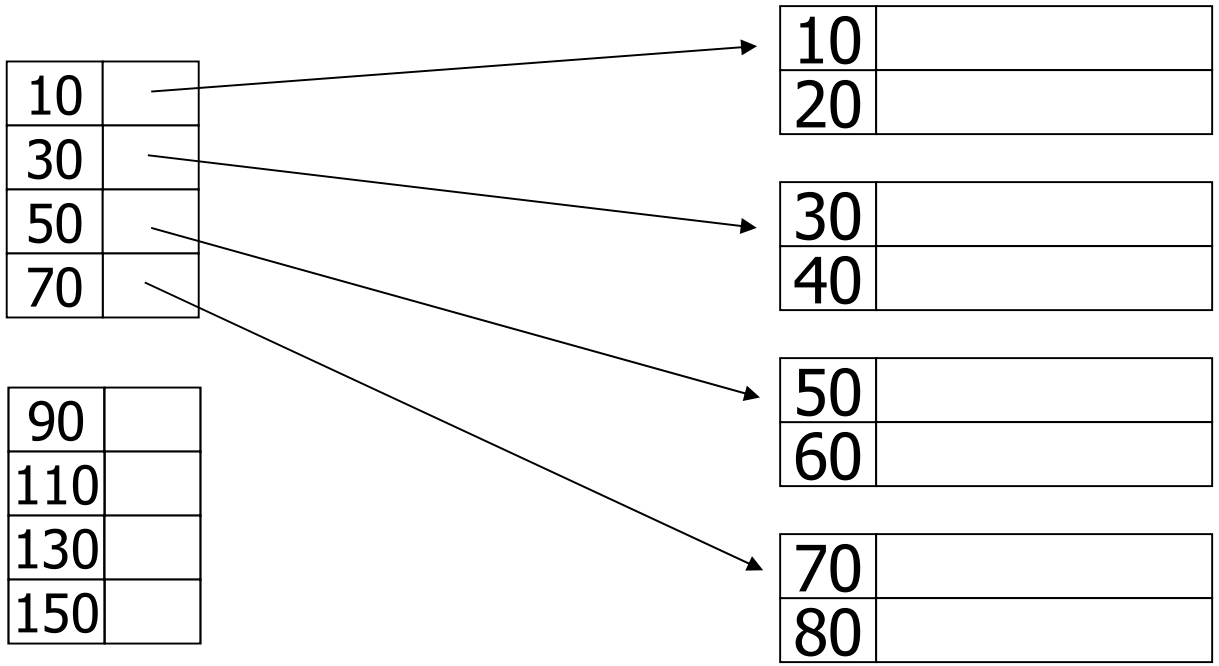
Deletion from Sparse Index

delete record 40



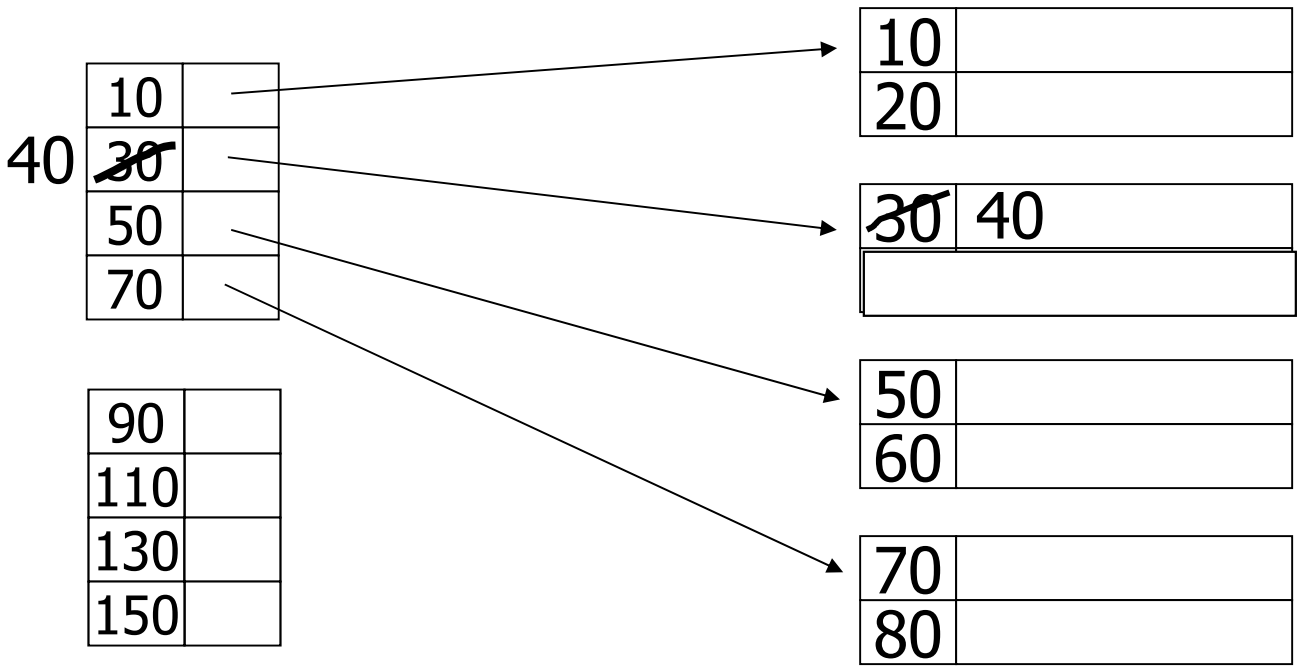
Deletion from Sparse Index

delete record 30



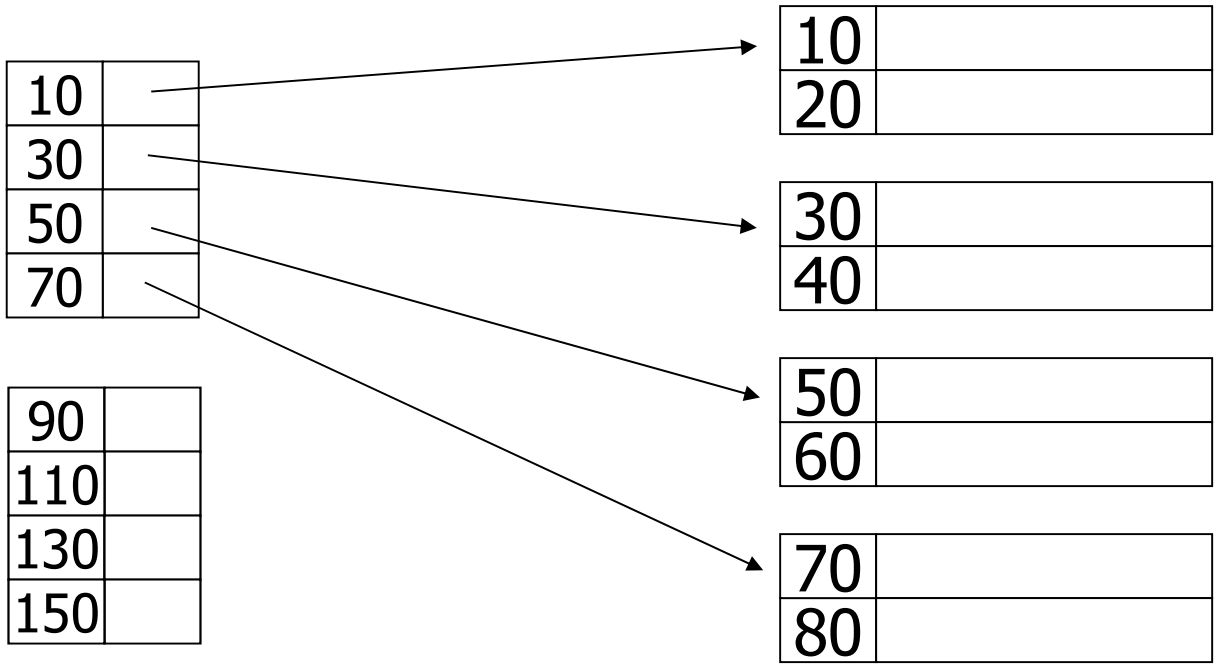
Deletion from Sparse Index

delete record 30



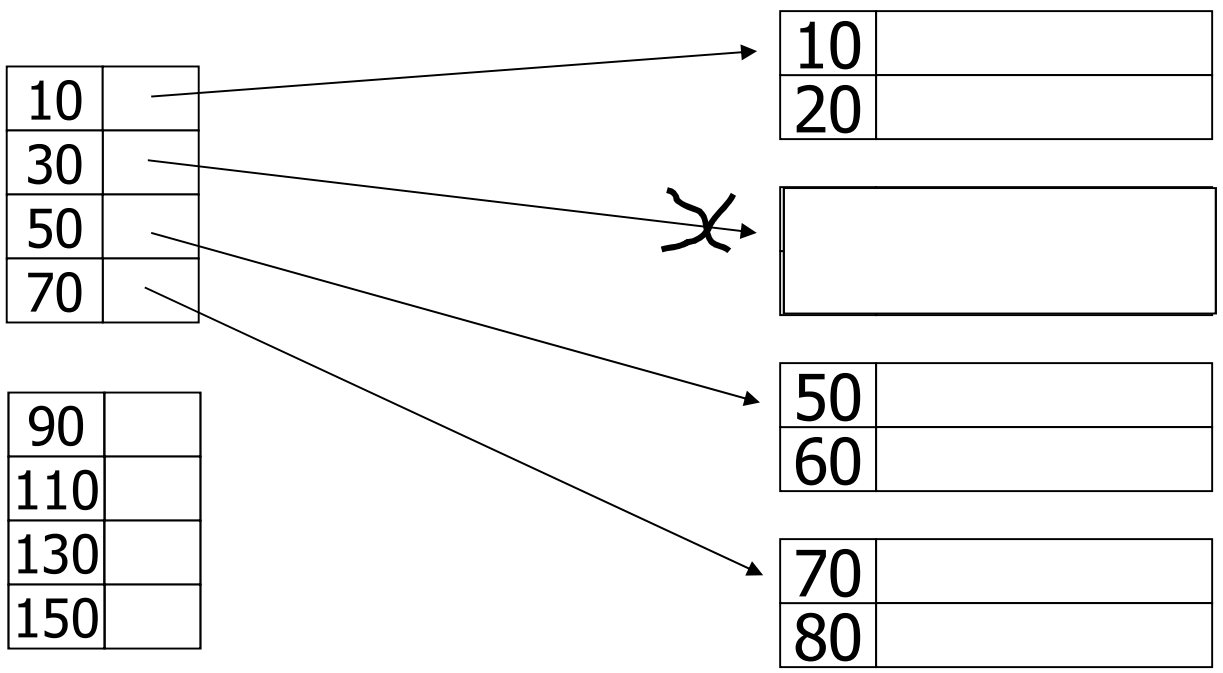
Deletion from Sparse Index

delete record 30 & 40



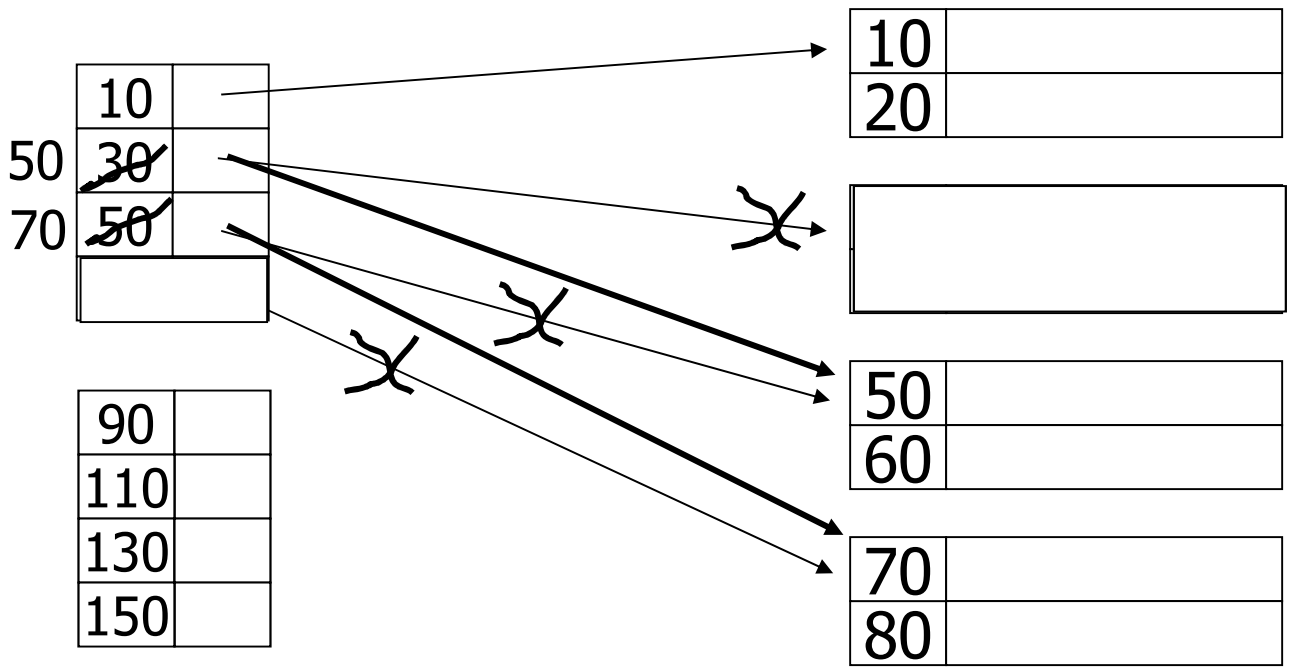
Deletion from Sparse Index

delete record 30 & 40



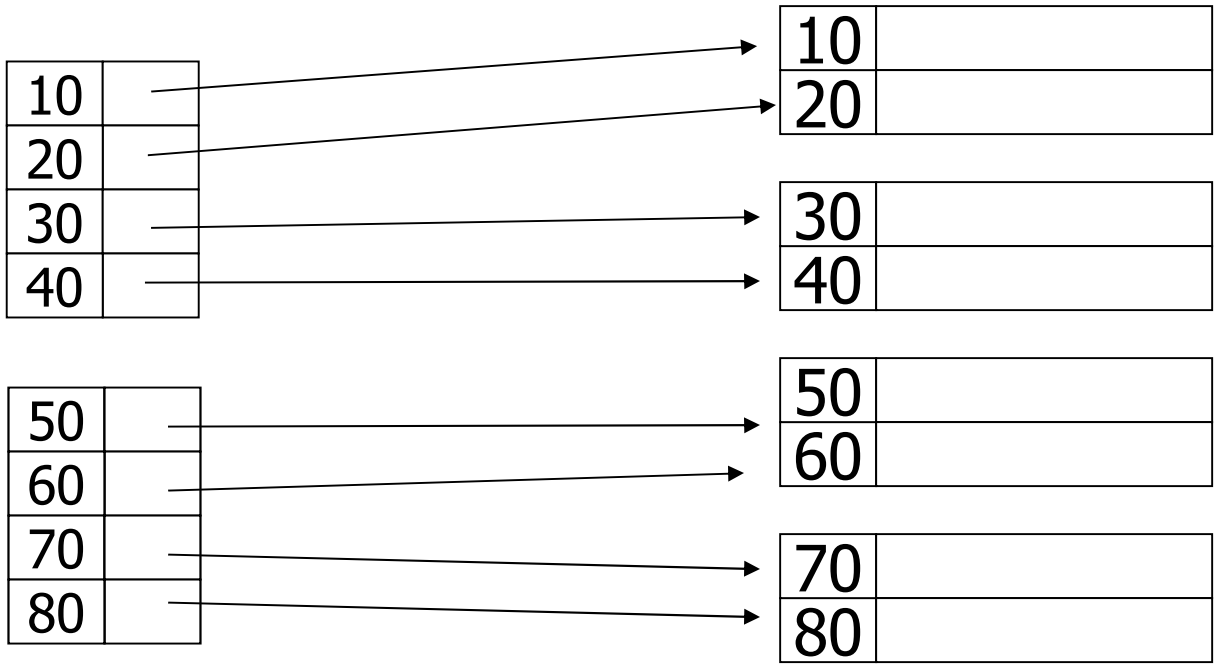
Deletion from Sparse Index

delete record 30 & 40



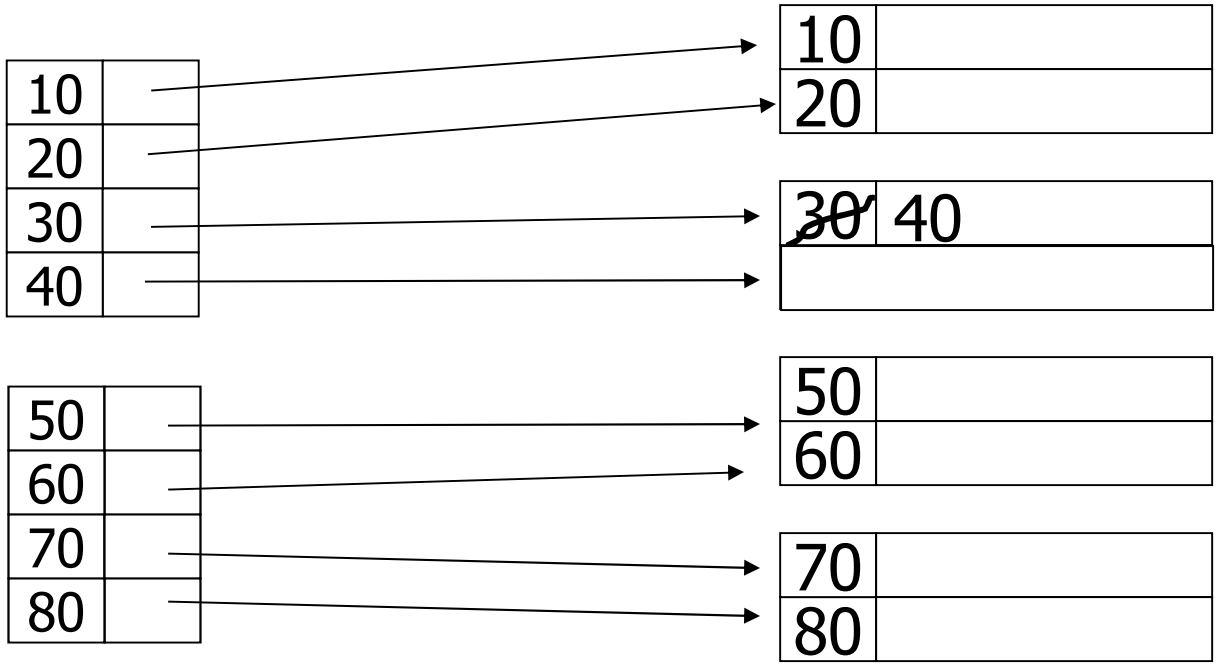
Deletion from Dense Index

delete record 30



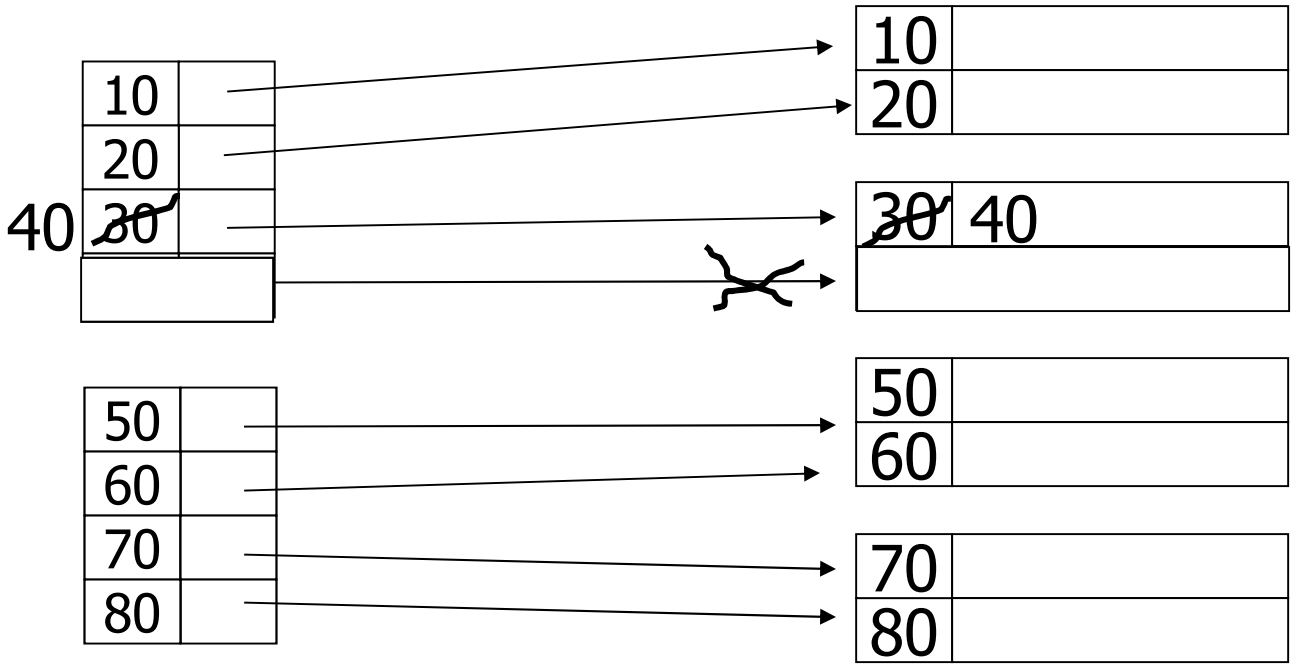
Deletion from Dense Index

delete record 30



Deletion from Dense Index

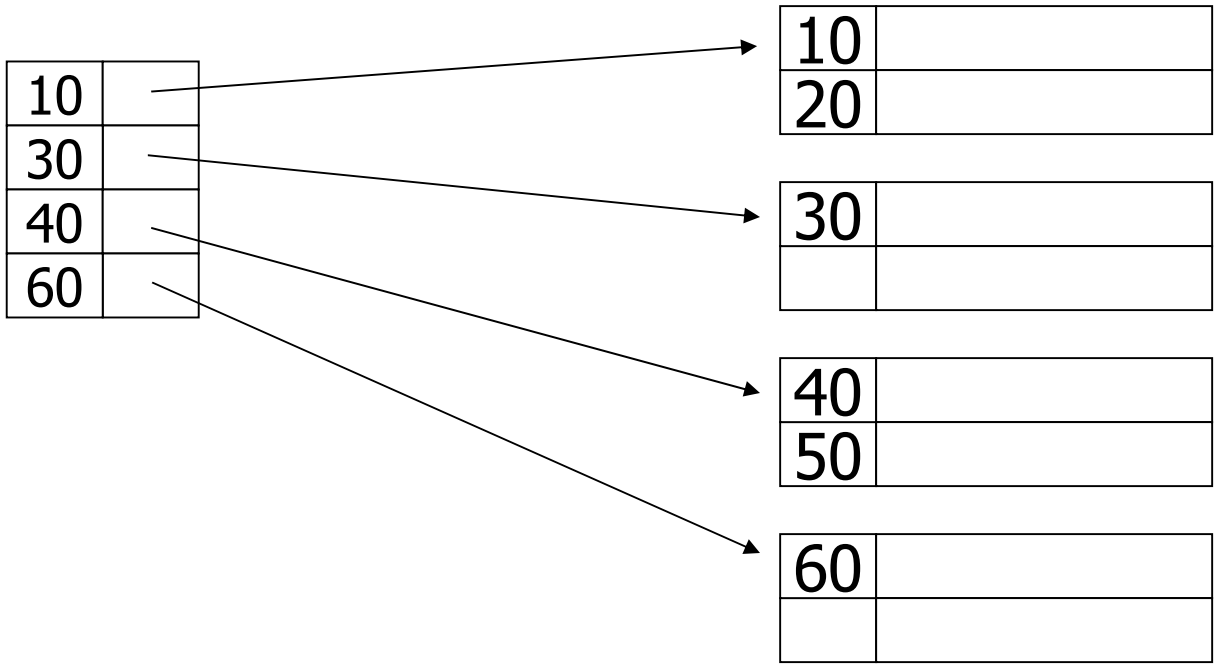
delete record 30



Unlike sparse index, always have to update the index

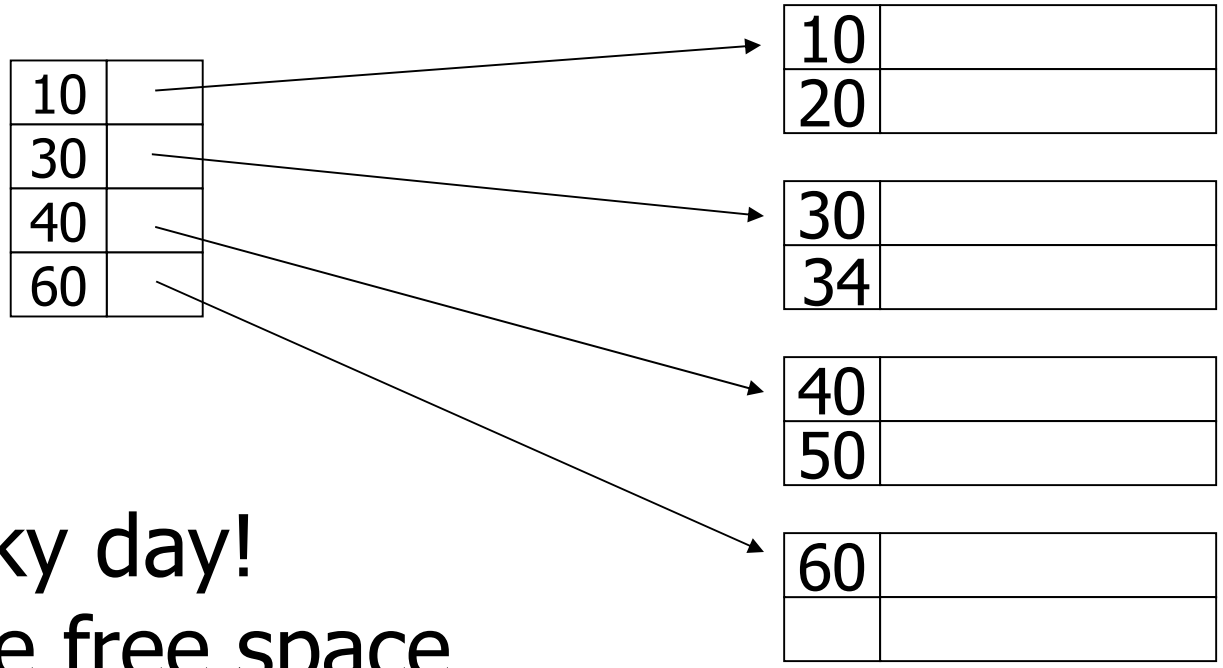
Insertion into Sparse Index

insert record 34



Insertion into Sparse Index

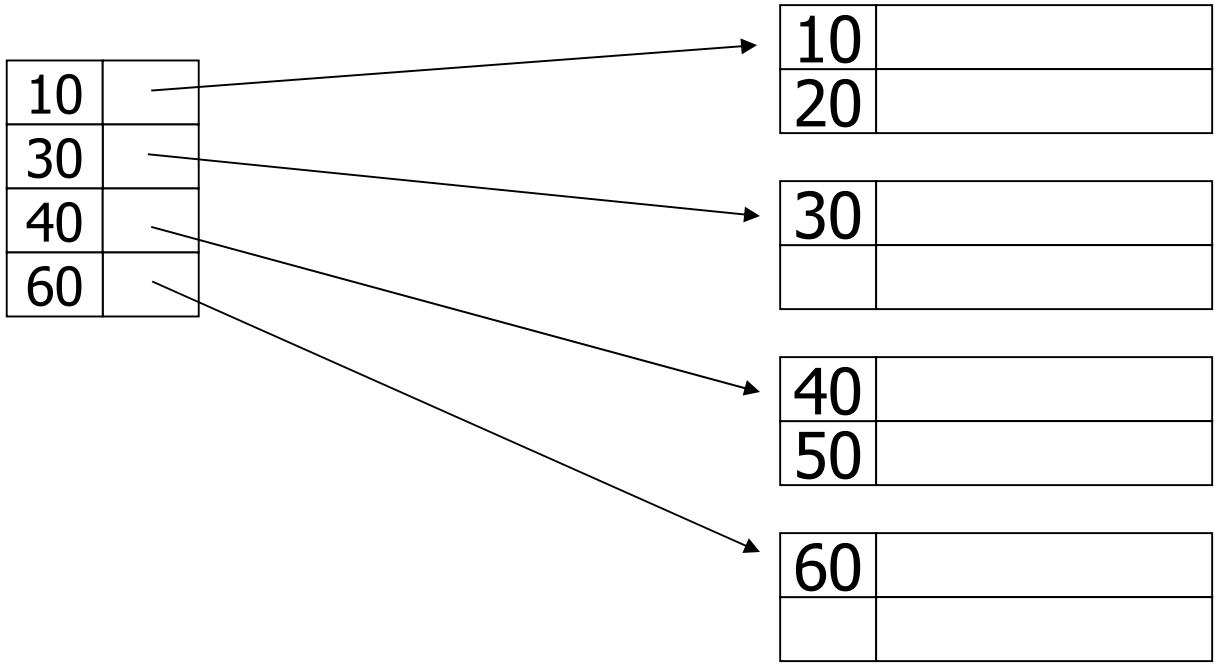
insert record 34



Our lucky day!
We have free space where we need it!

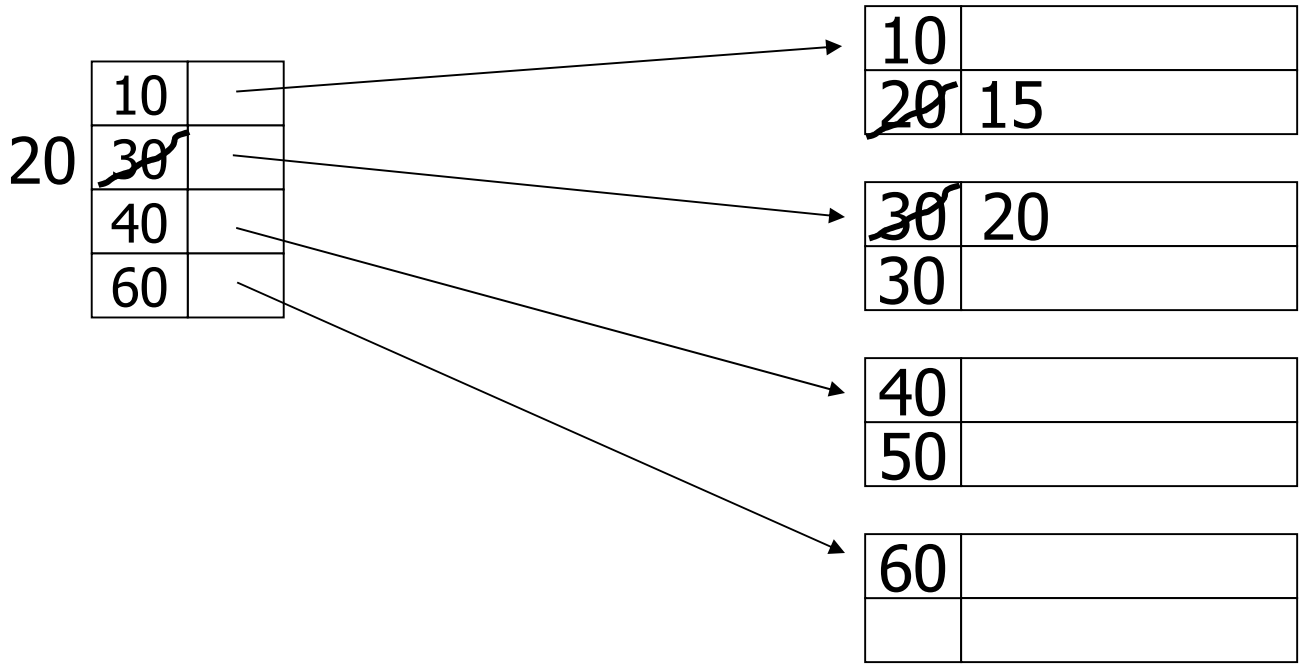
Insertion into Sparse Index

insert record 15



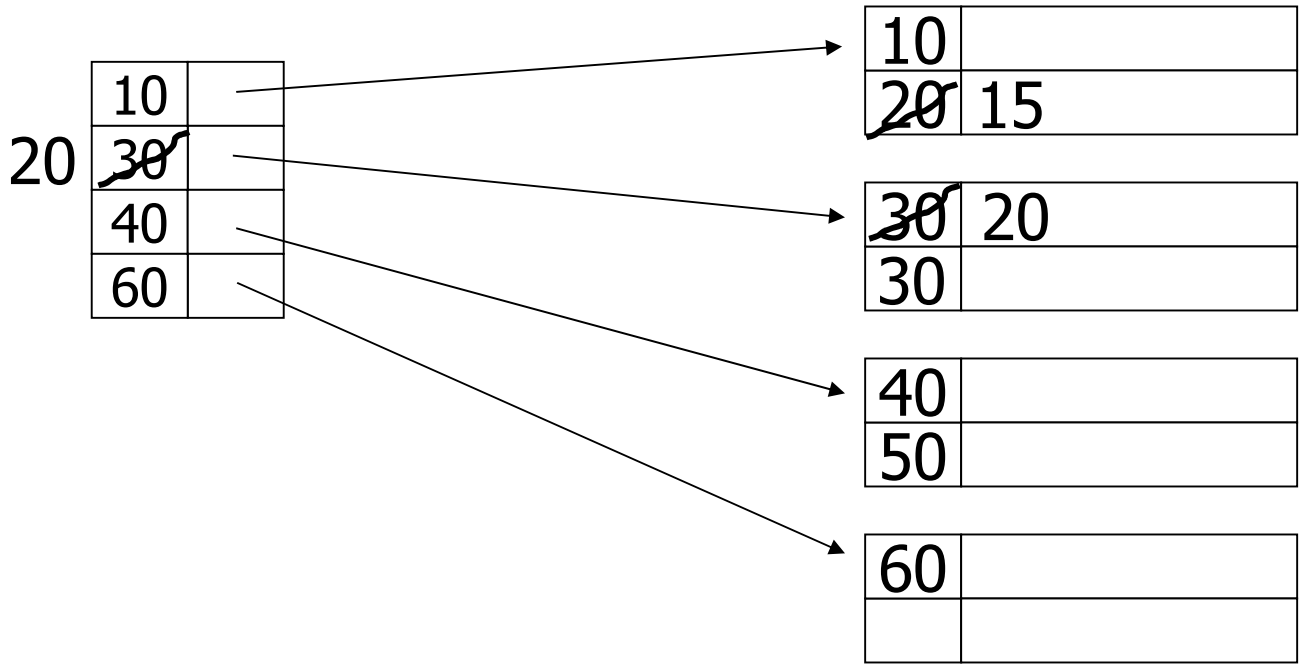
Insertion into Sparse Index

insert record 15



- Immediate reorganization of both the file and the index

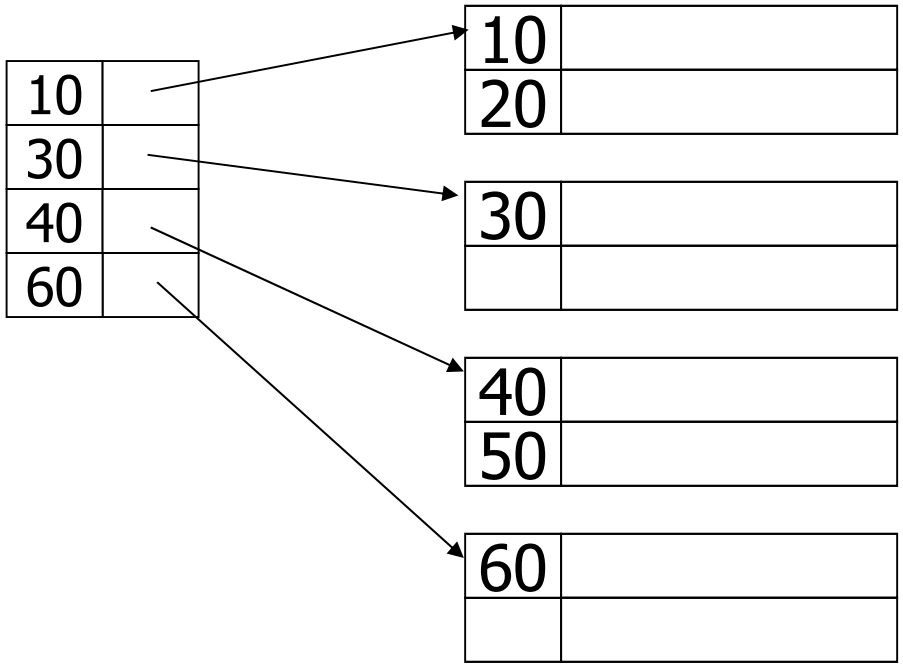
What if we now have to insert 13?



- Can add a new block to the file's chain between the first and second blocks
  - Also need to add new entry to the index
- But blocks will no longer be contiguous

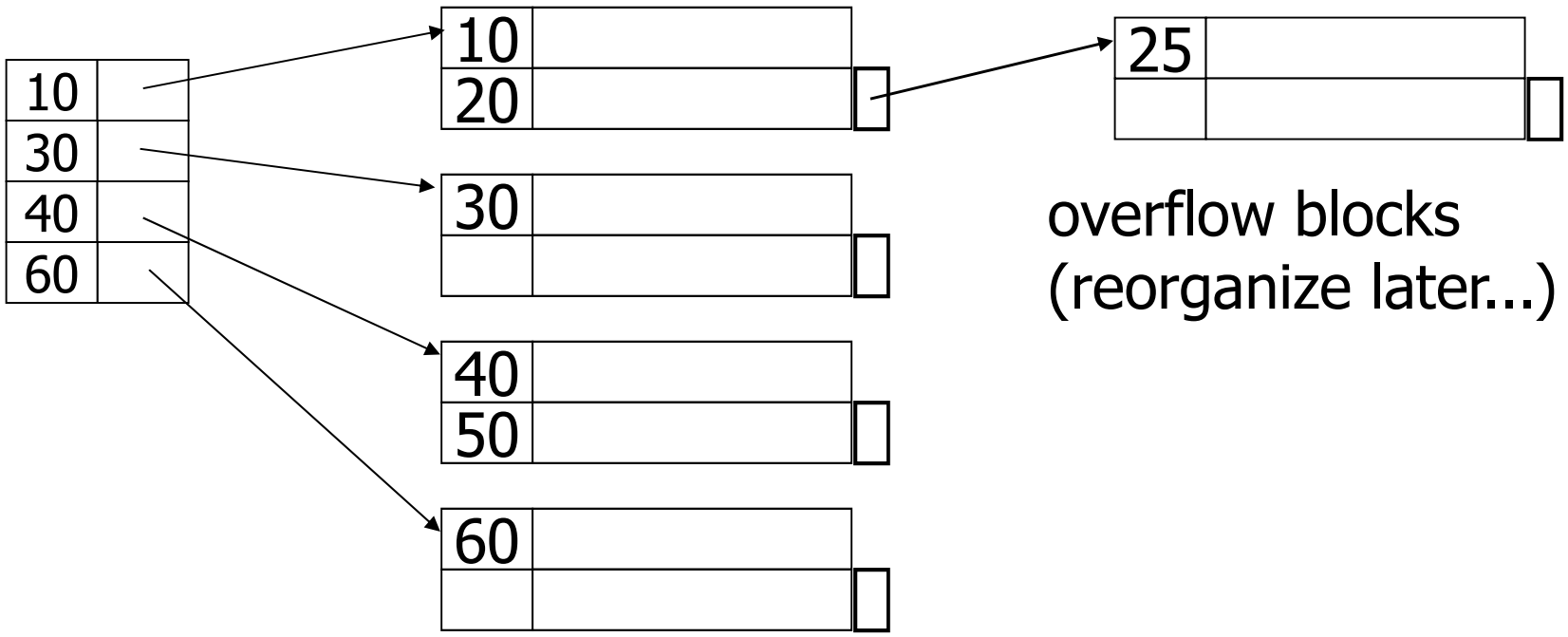
Alternative: Use Overflow Blocks

insert record 25



Alternative: Use Overflow Blocks

insert record 25



No need to update the index

Insertion into Dense Index

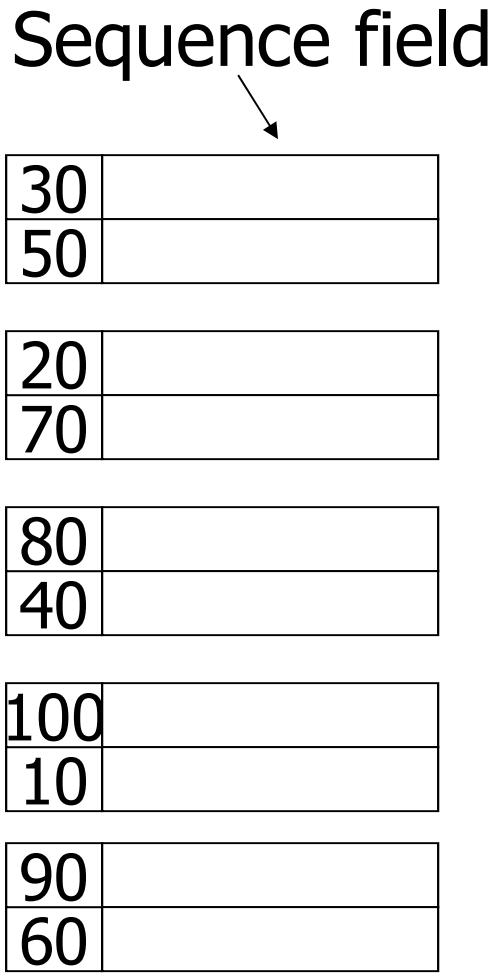
- Similar but often more expensive, because we have to update the index after every insertion

Dense Indexes for Secondary Keys

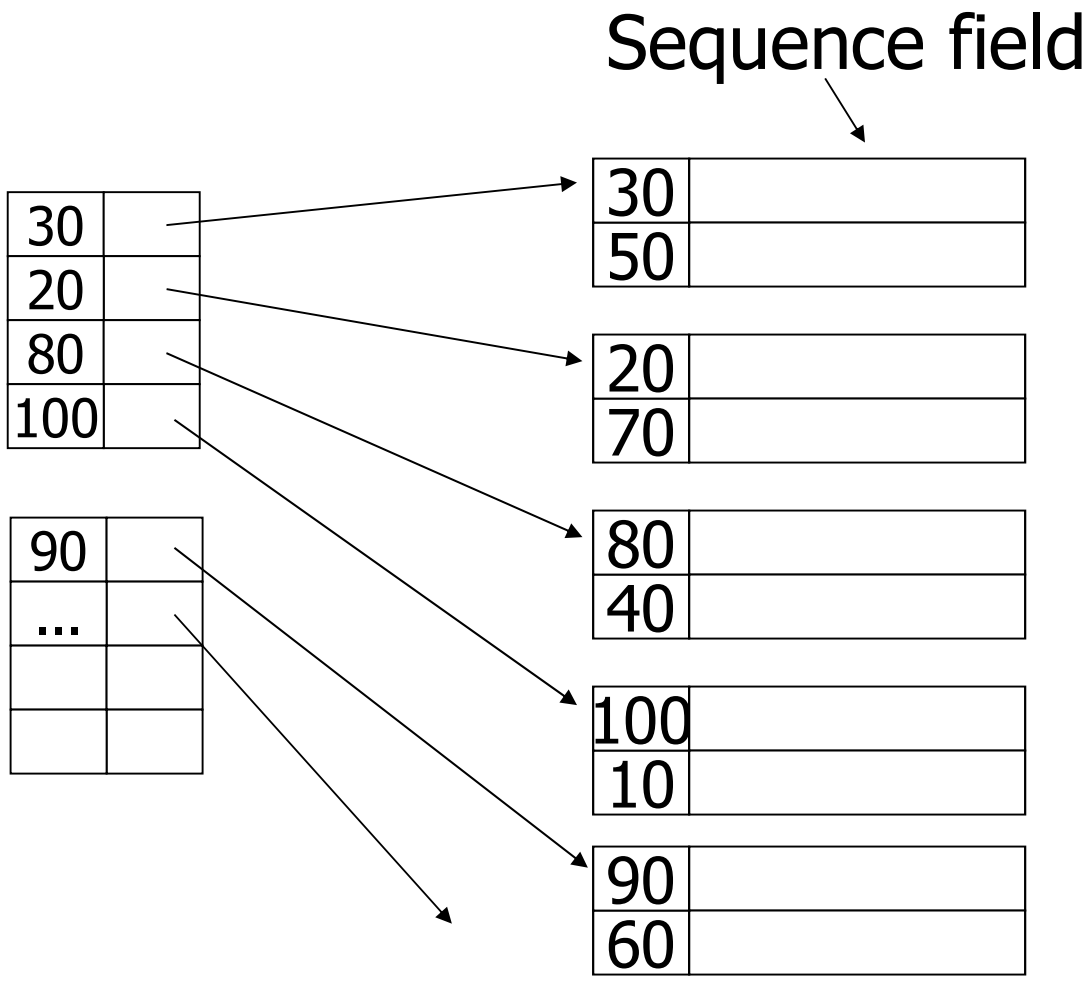
An index for a secondary key is sometimes called a *secondary index*

Secondary Indexes

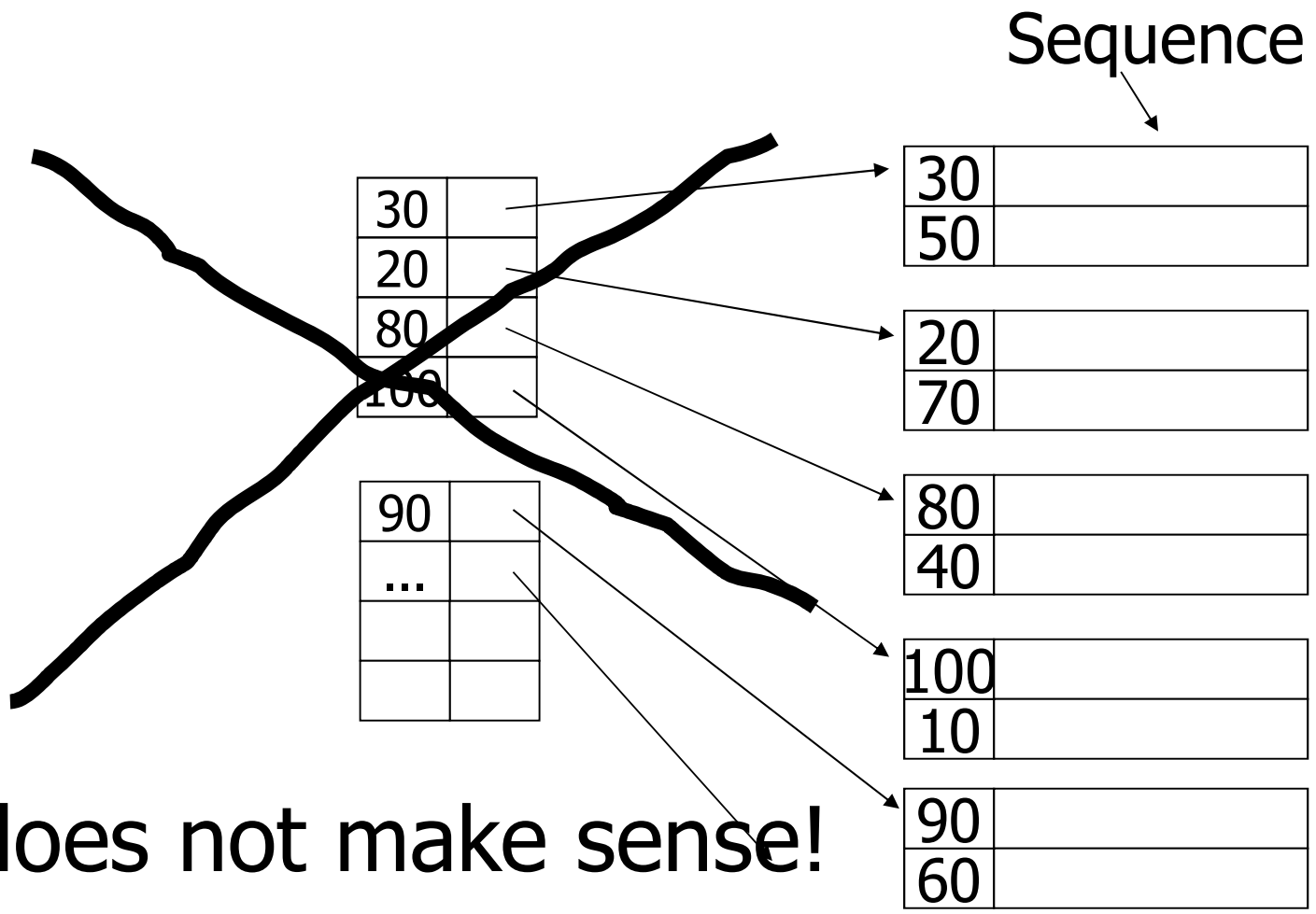
- Only the primary (i.e., organizing) index can determine the physical order of the records on the disk
- Secondary index is on an unsorted field



Sparse Secondary Indexes

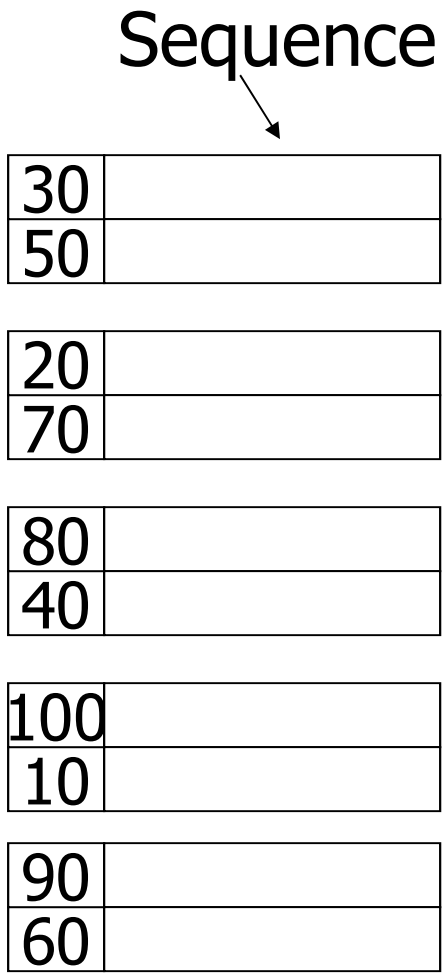


Sparse Secondary Indexes

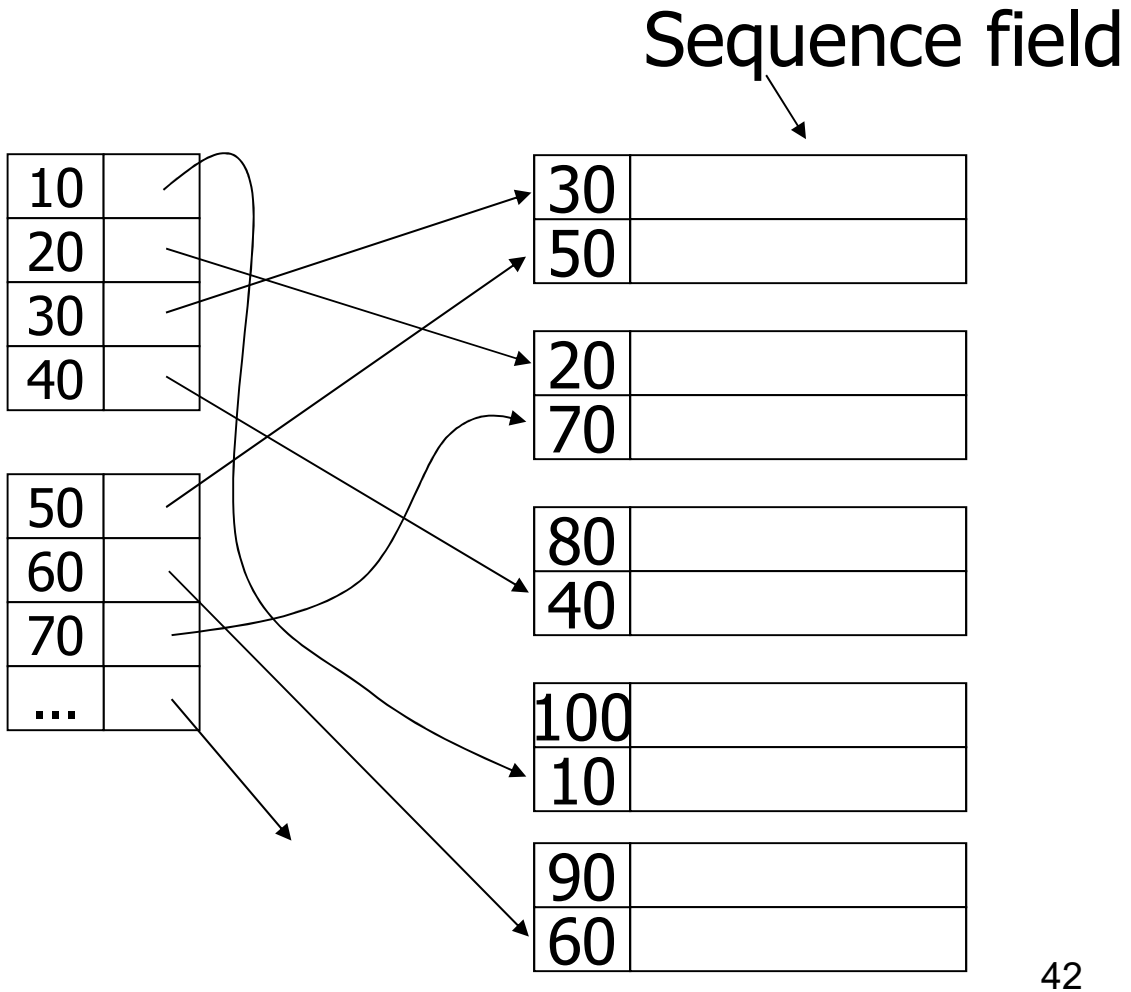


does not make sense!

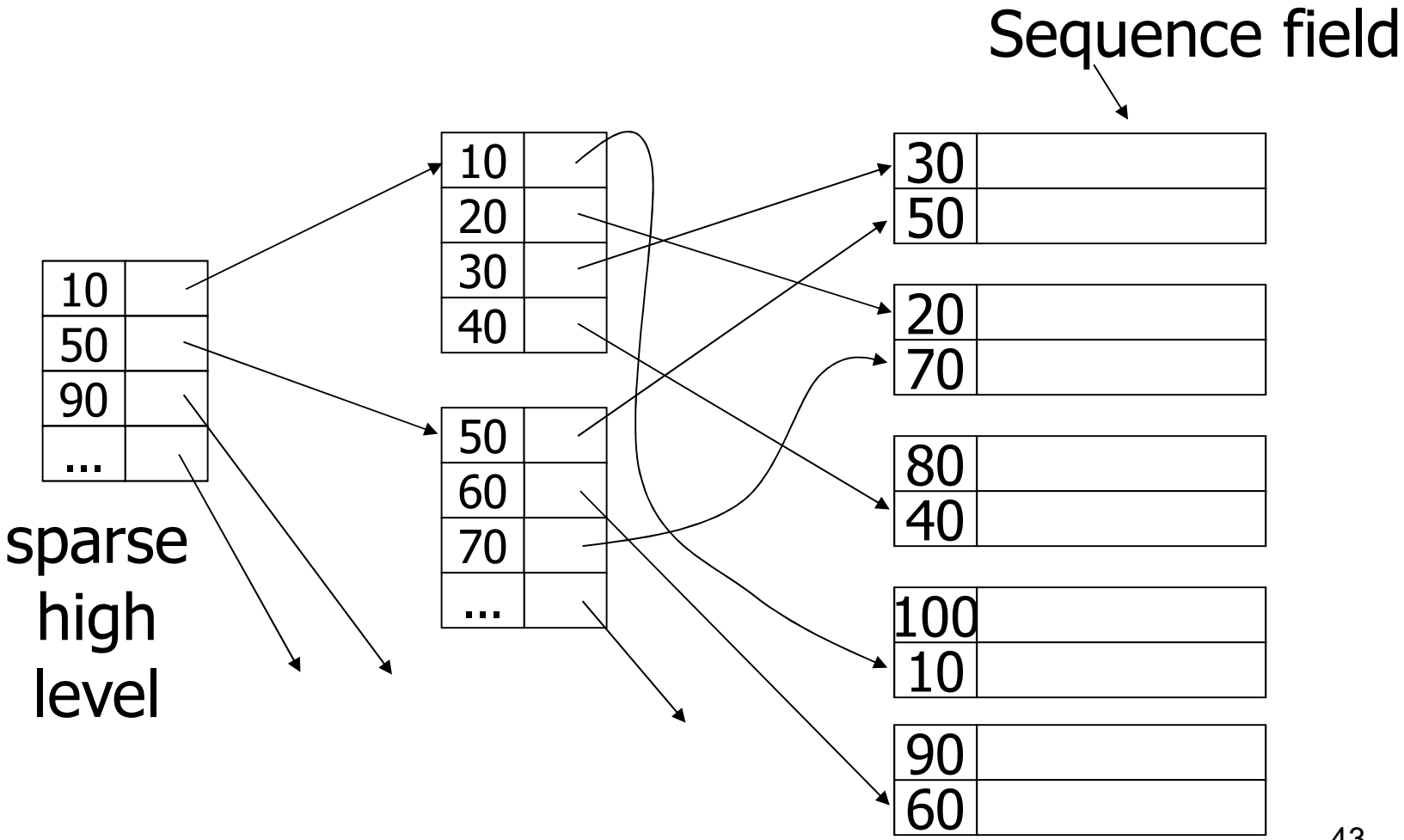
Secondary Index Must be Dense



Dense Secondary Index



But Higher Levels are Sparse



Duplicate Values & Secondary Indexes

20	
10	

20	
40	

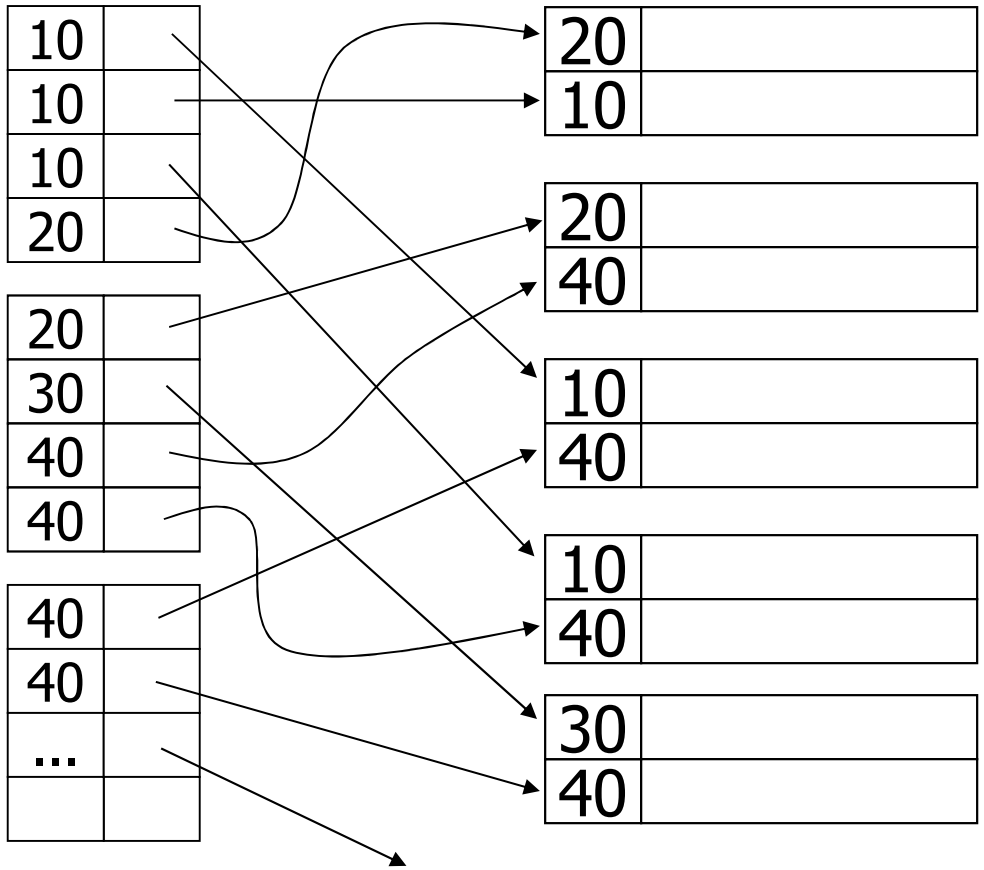
10	
40	

10	
40	

30	
40	

Duplicate Values & Secondary Indexes

one option...



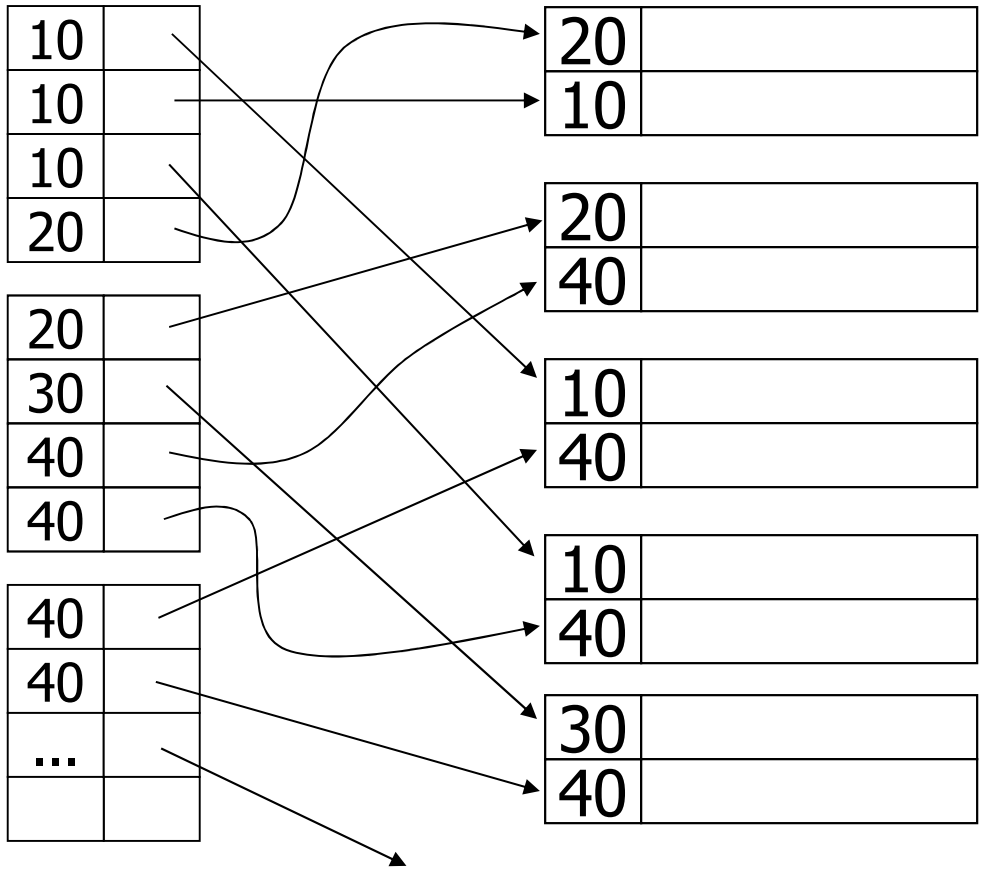
Duplicate Values & Secondary Indexes

one option...

<u>Problem:</u>

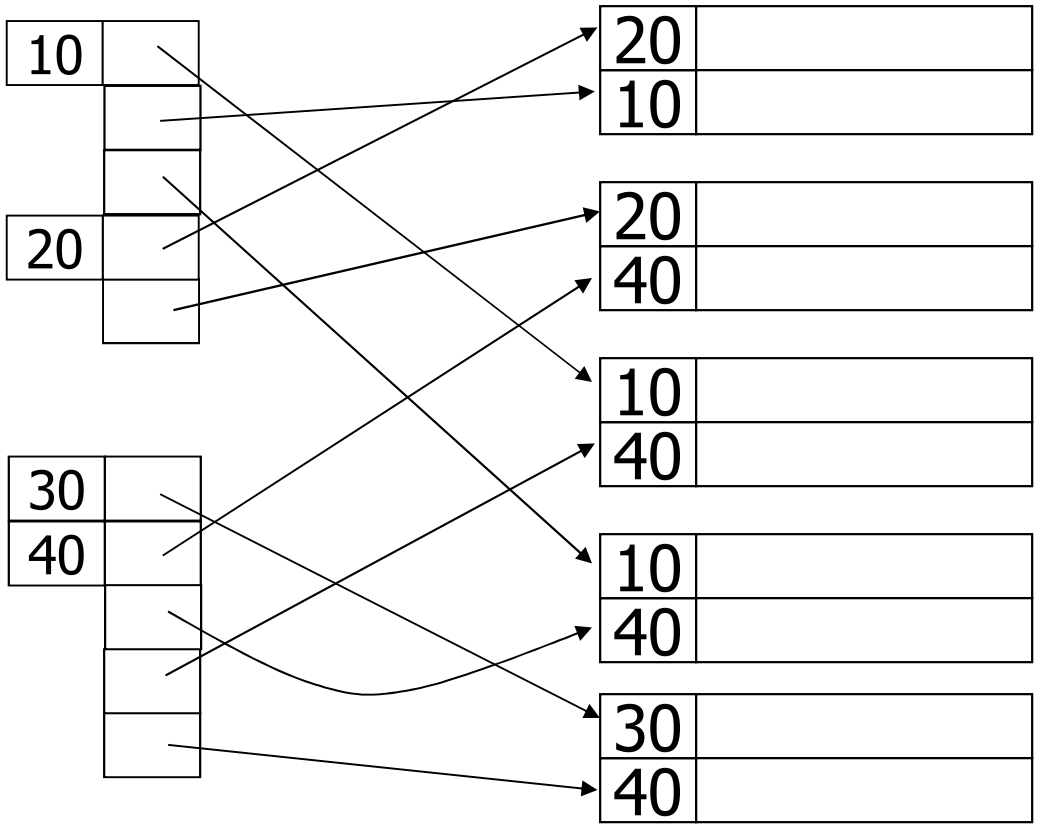
excess overhead!

- disk space
- search time



Duplicate Values & Secondary Indexes

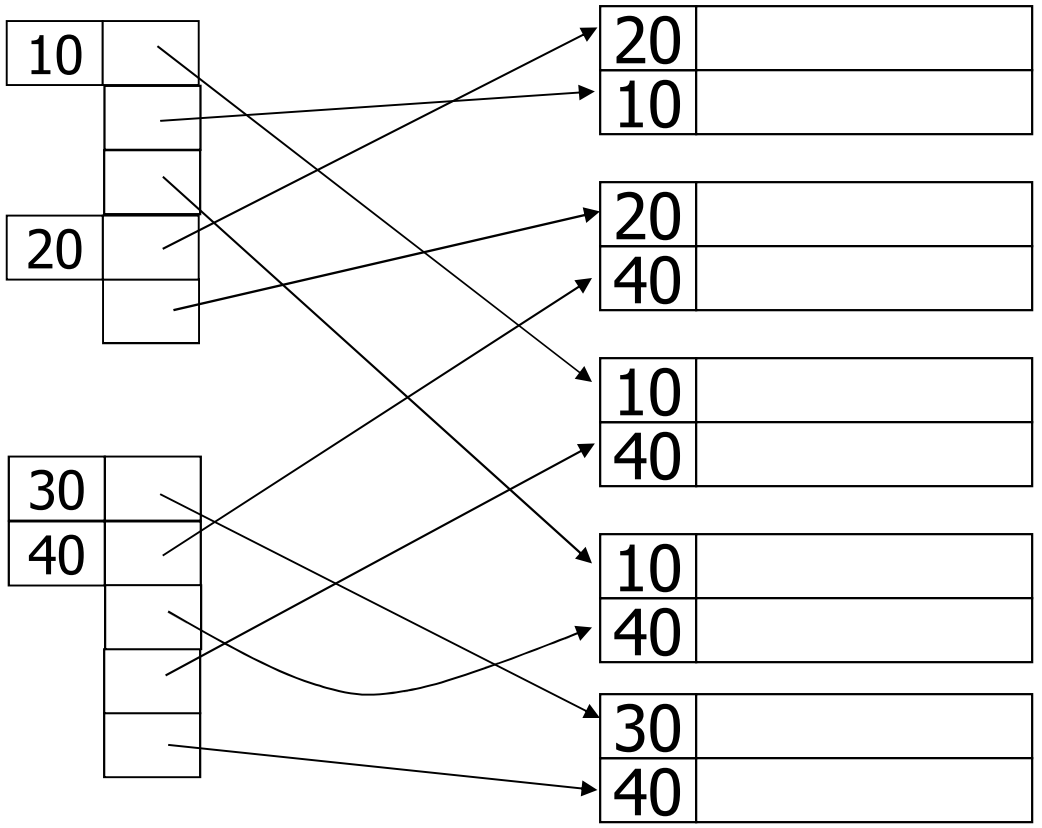
another option...



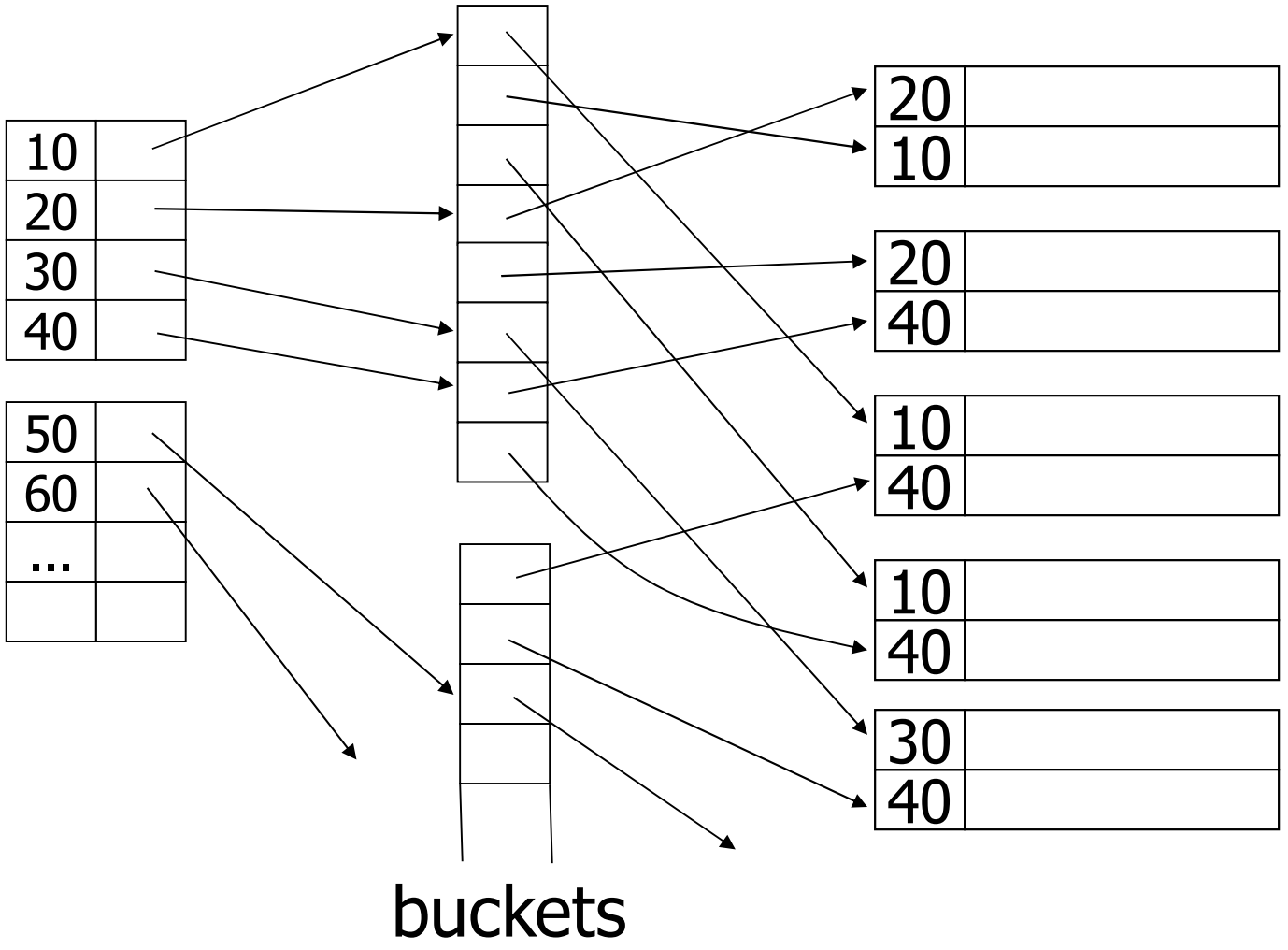
Duplicate Values & Secondary Indexes

another option...

<u>Problem:</u>
Variable-size records in index!



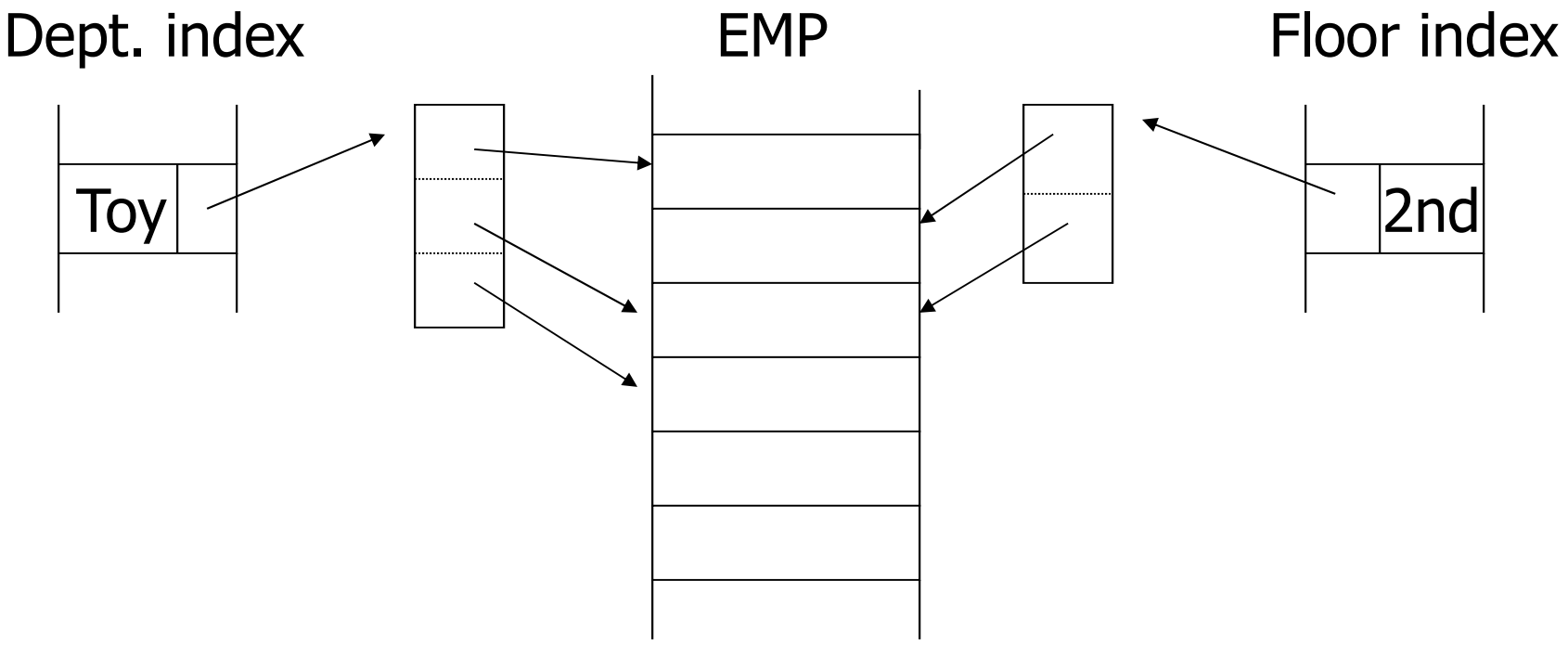
An Intermediate Level of Buckets



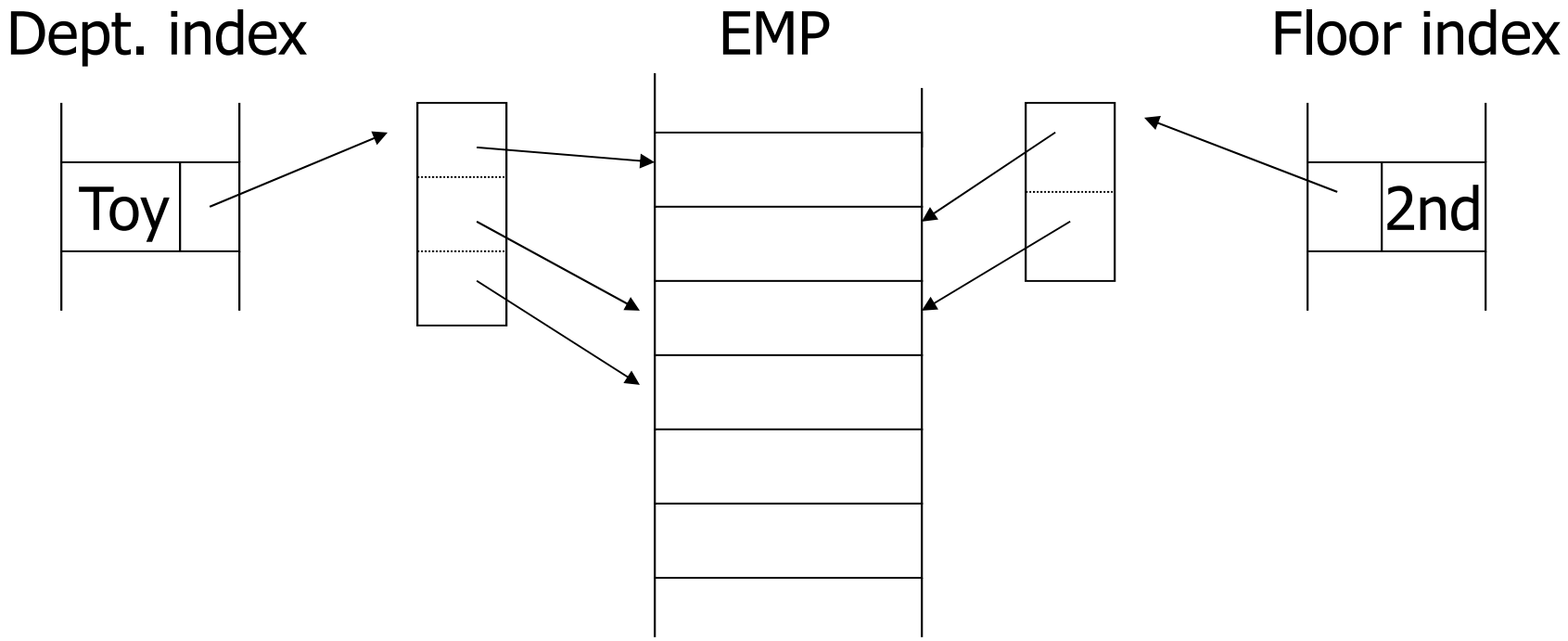
Why "Bucket" Idea is Useful

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

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



Query: Get employees in (Toy Dept) $\wedge$ (2nd floor)



$\rightarrow$ Intersect toy bucket and 2nd Floor bucket to get the set of matching EMP's

Summary of Dense & Sparse Indexes

- Both are simple (as long as there is just one level)
- Sparse is more efficient, because the index is smaller and more of it can be kept in memory
- Insertions are expensive when performed (if immediate reorganization is done), or over time (since performance deteriorates due to overflow)
  - More of a problem in a dense index, because every insertion also changes the index
- Secondary indexes must be dense
- Sometimes dense indexes improve efficiency by intersecting sets of pointers before accessing file

Something to Think About

- To shorten an index, can we use pointers just to blocks (instead of to records)? In which cases?
- In a relational system, can we organize a relation according to a field which is not a key in the FD sense? How?
  - If so, what is the advantage of doing that?
  - If so, how would we enforce the constraint that there should not be two records with the same key?

Note

- If a file is stored in sorted order on some field, then that field must be the primary search key
- If the file is stored as a heap (i.e., not sorted on any field), then the index for the primary search key must be dense
- If the primary search key is based on a lexicographic order of several fields and the file is sorted accordingly, then any prefix of those fields is a secondary search key