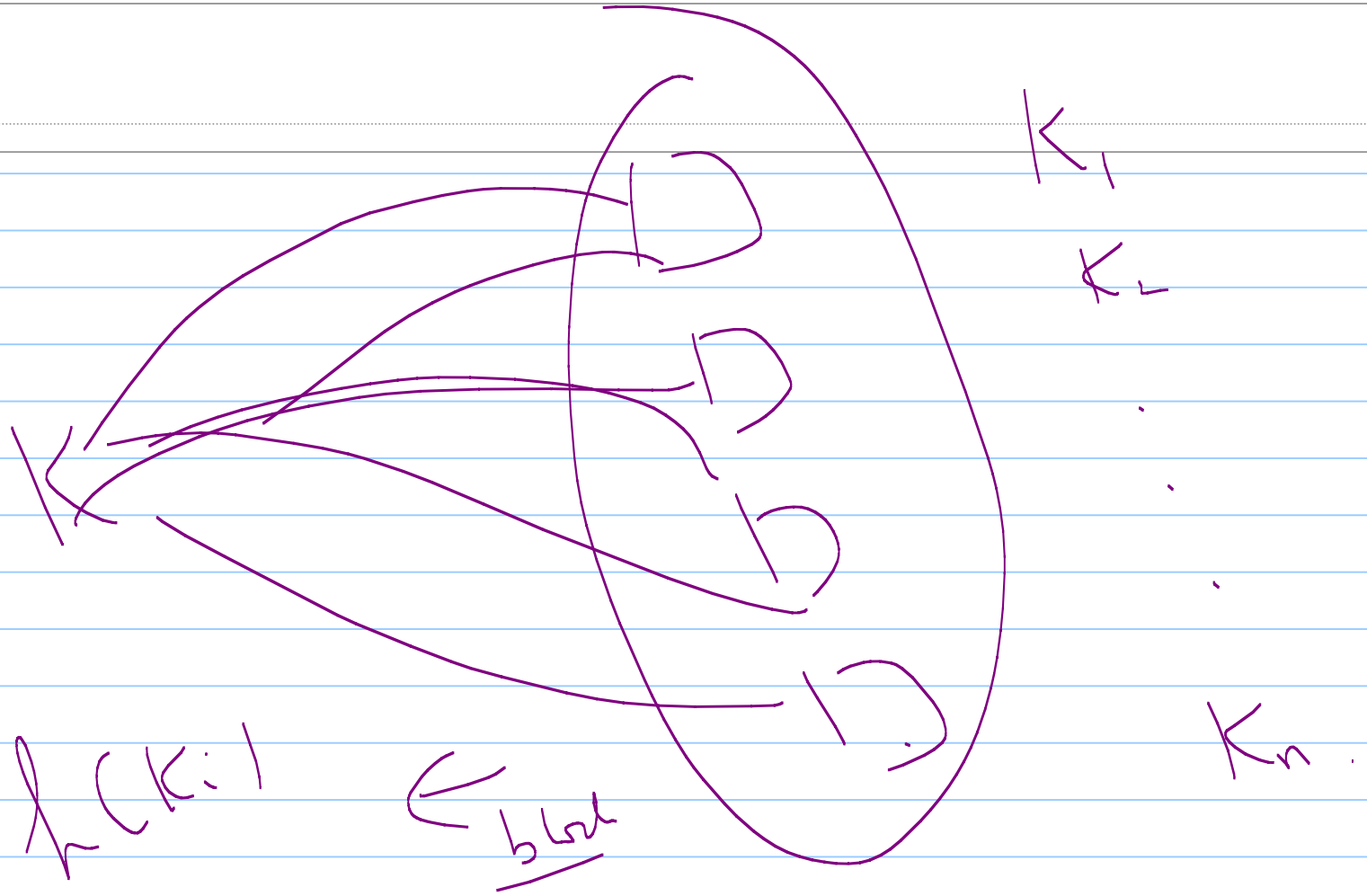
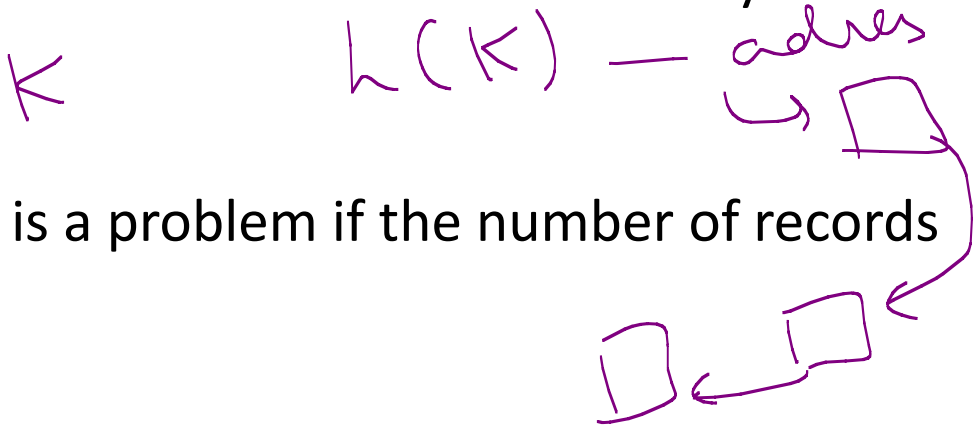
Note Title 1/16/2014

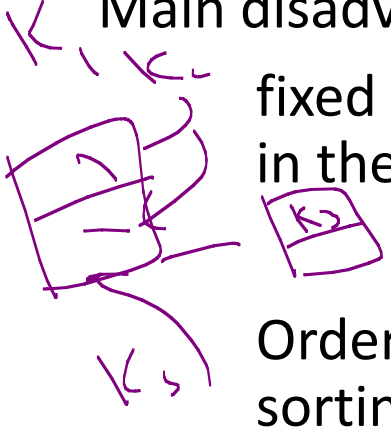


Static External Hashing

The hash function h should distribute the records uniformly among the buckets; otherwise, search time will be increased because many overflow records will exist



Main disadvantages



- fixed number of buckets M is a problem if the number of records in the file grows or shrinks
- Ordered access on the hash key is quite inefficient (requires sorting the records)

Static external hashing

The file blocks are divided into

M equal sized buckets, numbered $bucket_0$, $bucket_1$, …, $bucket_{M-1}$

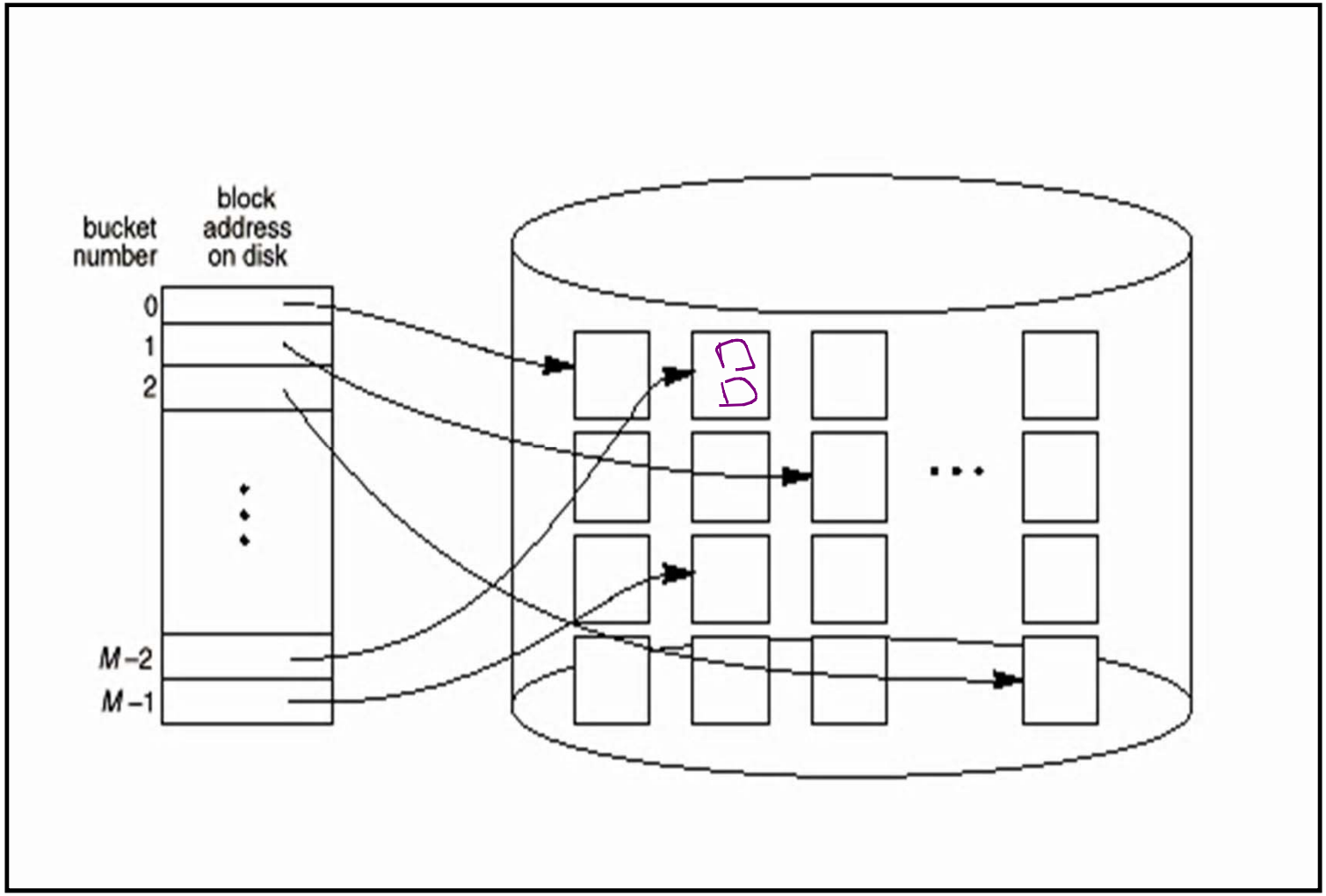
One of the fields is designated to be the hash key of the file

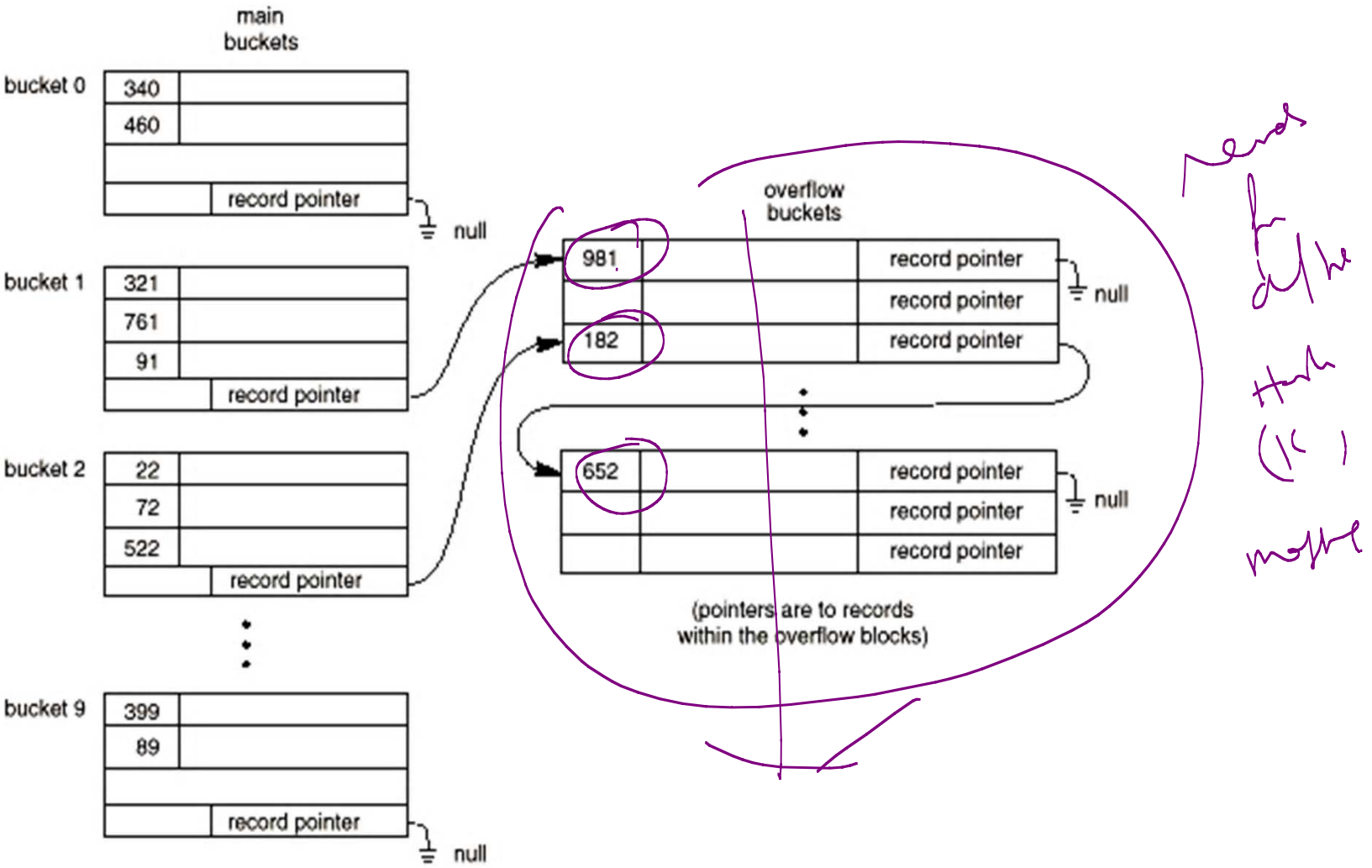
The record with hash key value K is stored in bucketi where i = h(K), and h is the hashing function

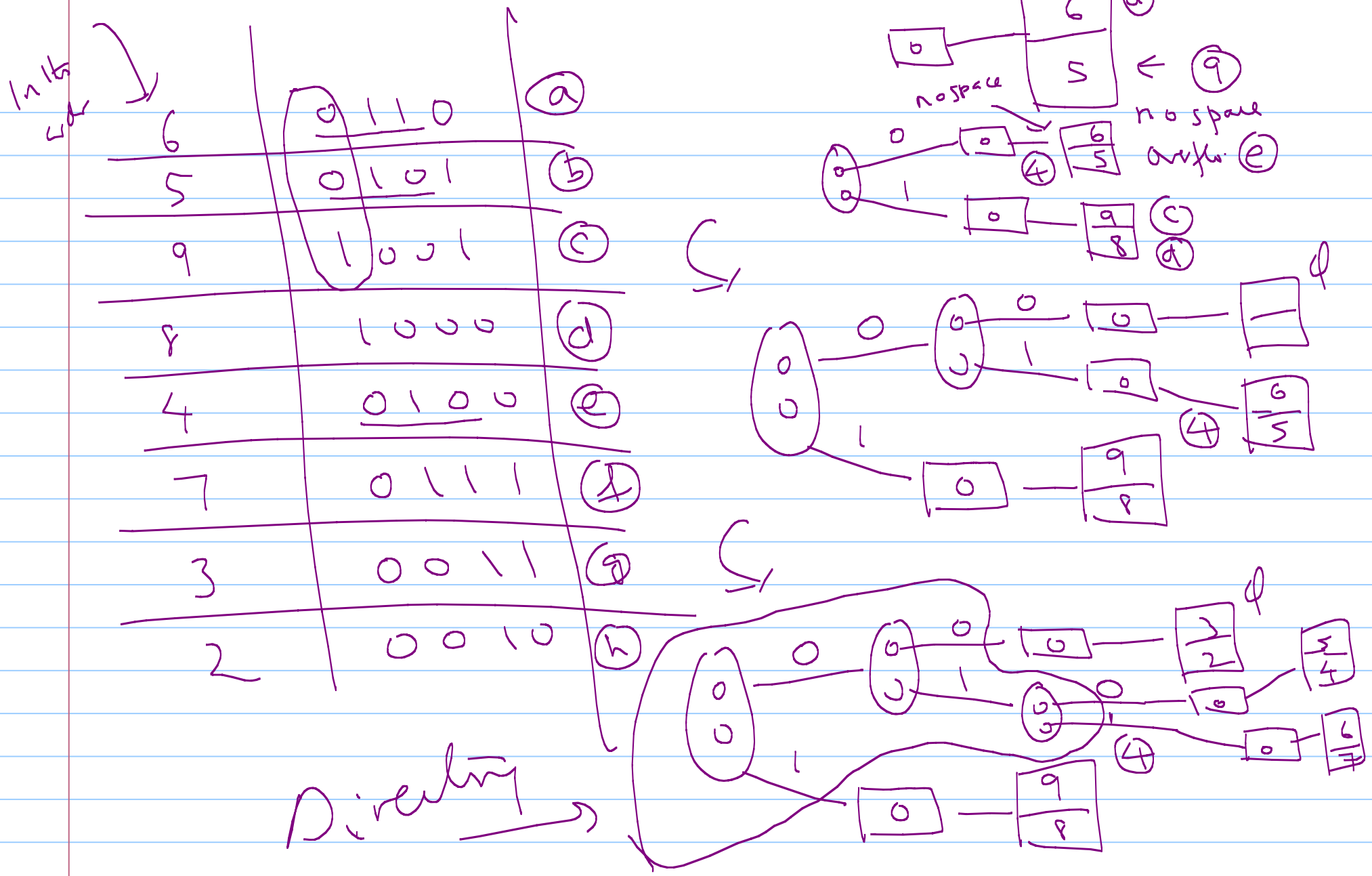
Collisions occur when a new record hashes to a bucket that is already full

an overflow file is kept for storing such record. Overflow records that hash to each bucket can be linked together

to reduce overflow records, a hash file is typically kept 70-80% full


bucket number
block address on disk
0
1
2
M–2
M–1


main
buckets
bucket 0
340
460
record pointer
null
bucket 1
321
761
91
record pointer
bucket 2
22
72
522
record pointer
bucket 9
399
89
record pointer
null
overflow
buckets
981
record pointer
record pointer
182
record pointer
null
652
record pointer
record pointer
record pointer
null
(pointers are to records
within the overflow blocks)


Inits
6
5
9
8
4
7
3
2
a
b
c
d
e
f
g
h
Pivoting
no space
no space

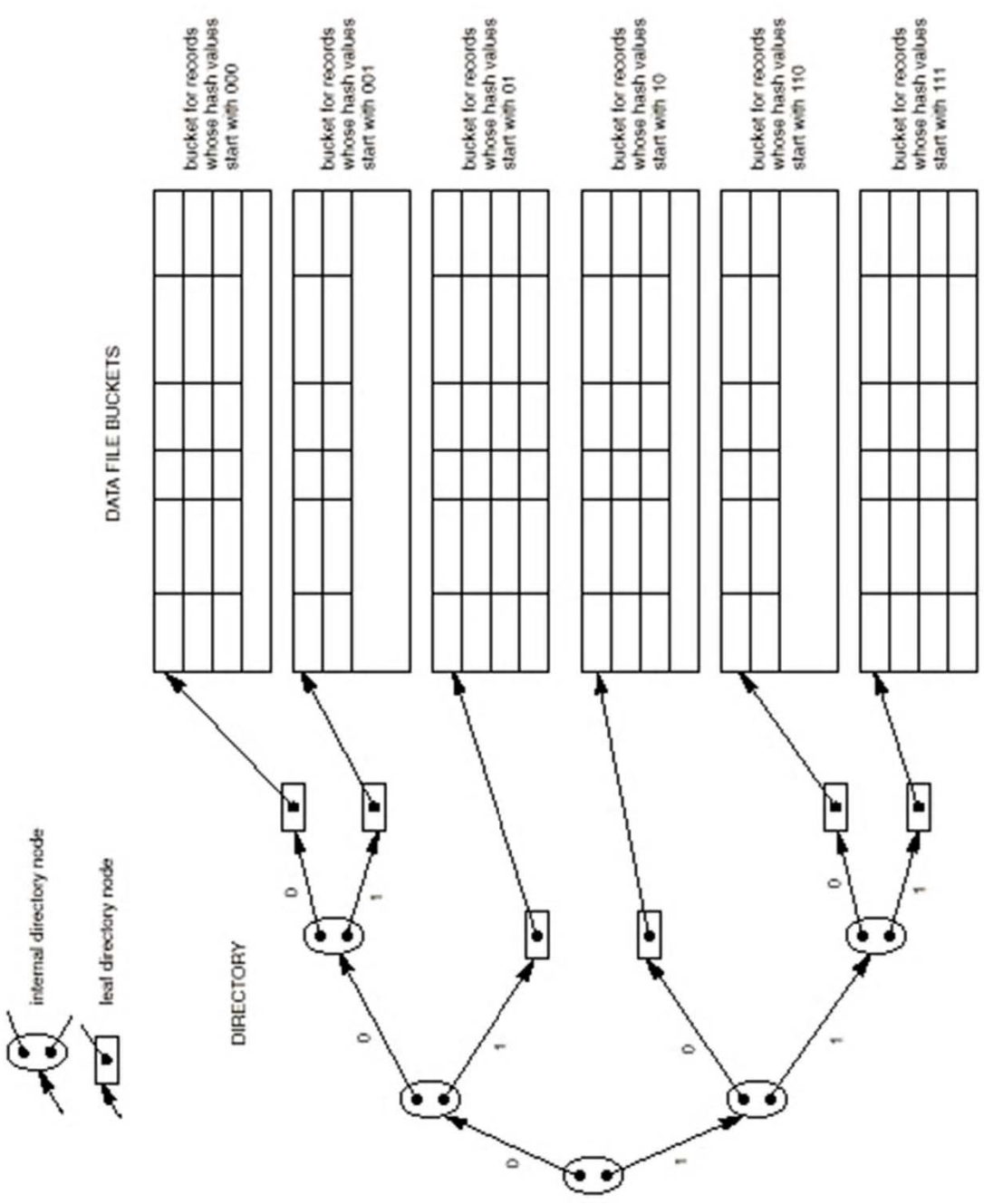
Dynamic and Extendible Hashing

Hashing techniques are adapted to allow the dynamic growth and shrinking of the number of file records

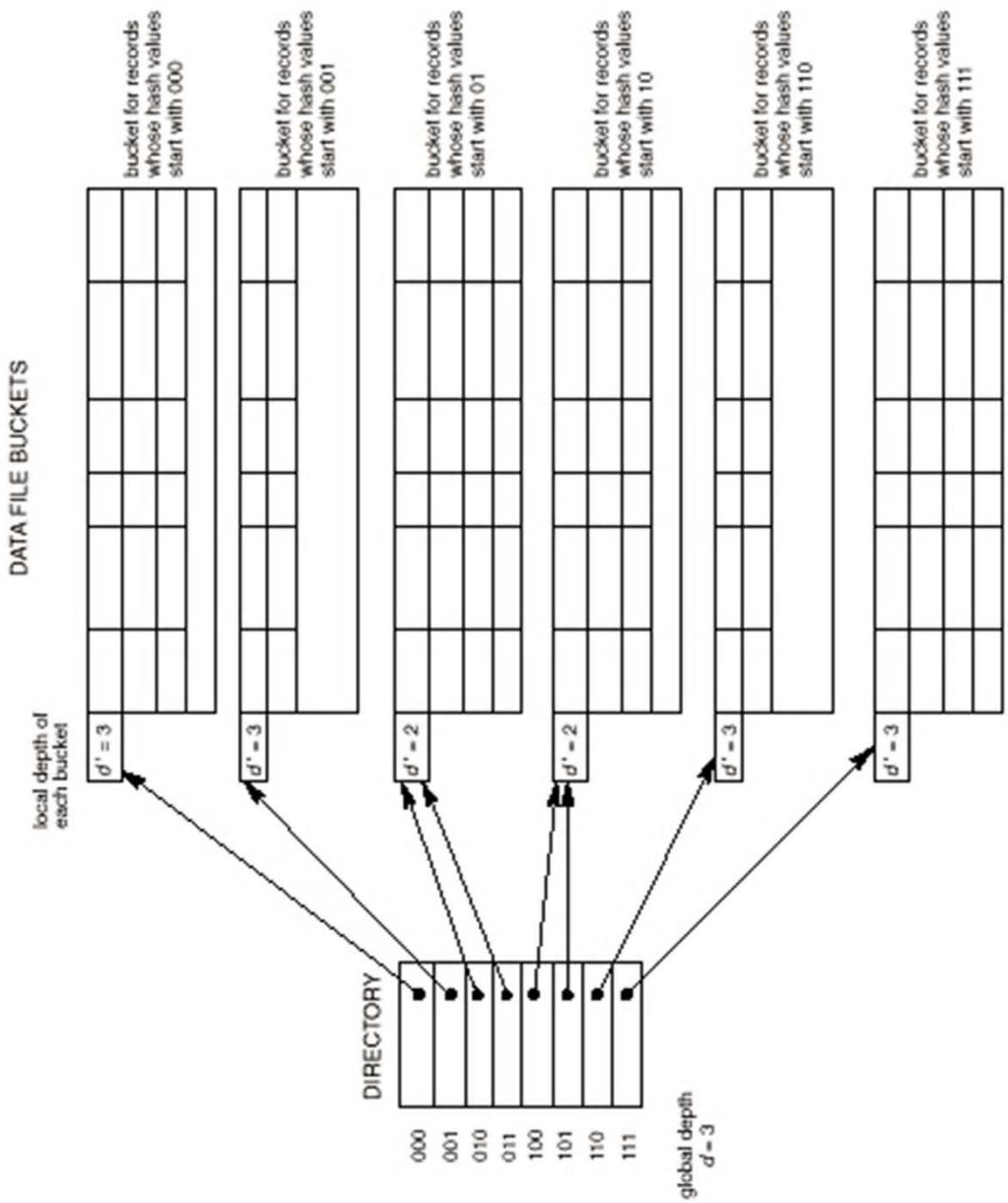
These techniques include the following: dynamic hashing, extendible hashing, and linear hashing

Both dynamic and extendible hashing use the binary representation of the hash key h(K) in order to access a directory. In dynamic hashing the directory is a binary tree. In extendible hashing the directory is an array of size 2^d where d is called global depth

DYNAMIC HASHING



EXTENDIBLE HASHING



Dynamic and Extendible Hashing Techniques

The directories can be stored on disk, and hey expand or shrink dynamically. Directory entries point to the disk blocks that contain the stored records.

An insertion in a disk block that is full causes the block to split into two blocks and the records are redistributed among the two blocks. The directory is updated appropriately.

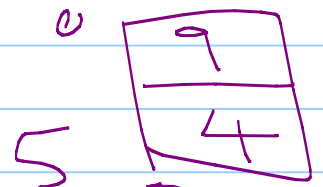
Dynamic and extendible hashing do not require an overflow area.

$h_0 = k \bmod (2^0)$

$9 \to 9 \bmod (2^0) = 0$

$4 \bmod (2^0) = 0$

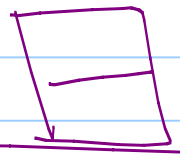
$5 \bmod (2^0) = 0$



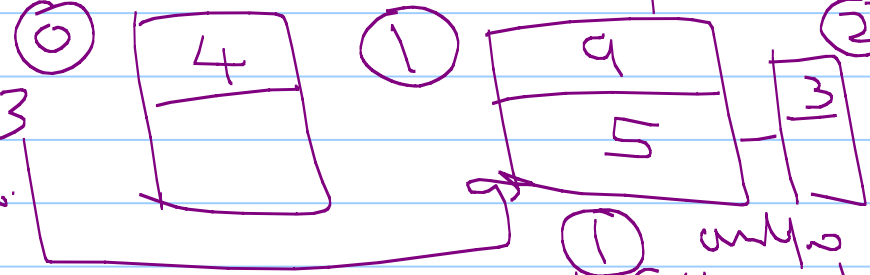
overflow.

$h_0 = k \pmod{2^0}$

$h_1 = k \pmod{2^1}$

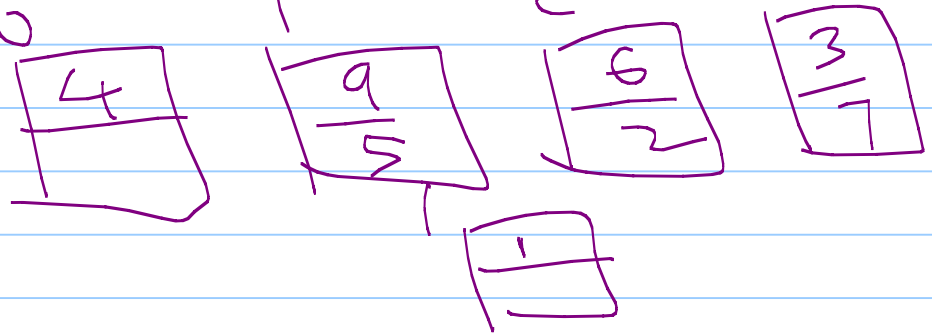


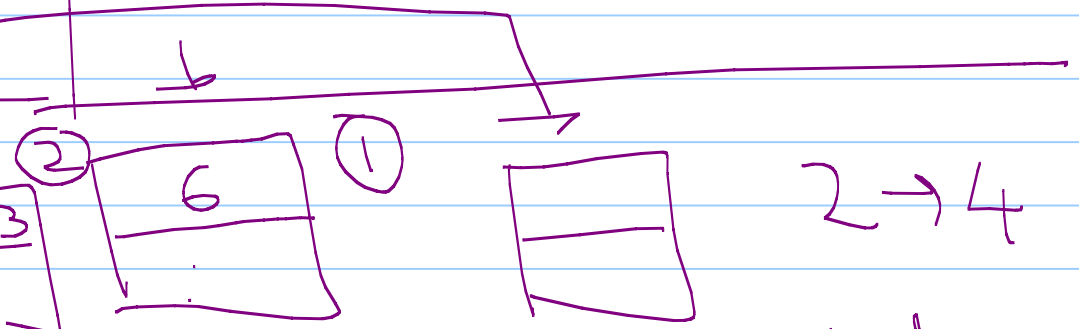
$h_1 = k \bmod (2 \times 2^0)$



$0 < 1$

$h_2 = k \pmod 4$





overflow

$h_2(k) = k \bmod (2 \cdot 2)$

$= k \bmod (4)$

Doubled my block

Linear Hashing

Linear hashing does require an overflow area but does not use a directory. Blocks are split in linear order as the file expands.

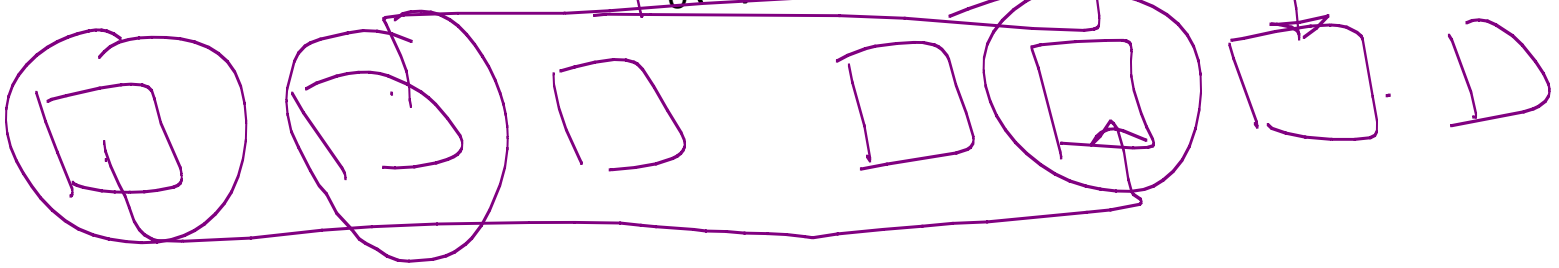
The principle of linear hashing is incremental organization

The file space grows by appending blocks linearly to the end of the hash space. For collisions which still occur overflow blocks are used

Let M (power of 2) to indicate initial basic allocation; the current allocation, M_1 begins with M buckets

The procedure operates unchanged until the file reaches a size of 2M buckets. At that point the records are adjusted to 2M buckets

The initial hash function is $h_0(K) = K \bmod M$



Linear hashing (cont.)

When a collision leads to an overflow record in any file bucket, the bucket n = M_1 – M is split into two buckets, original bucket M_1 – M, and the new bucket M_1 + 1. The value of M_1 is now set to M_1 + 1 (or n is incremented)

All the records in the original block are assigned to two blocks M_1-M and M_1 +1 by using hash function $h_1(K) = K \bmod (2M)$

The key property of two hash functions is that any record hashed to bucket I based on h_0 will be hashed to bucket I or bucket i+M using hash function h_1

To retrieve a record with key K first apply hash function $h_0(K)$; if $h_0(K) < n$; apply hash function $h_1(K)$ because of bucket split

When (n=M or M_1 = 2M) all the blocks are split; and $h_1(K)$ applies to all record, reset M= M_1 = 2M and n=0

Linear Hashing (cont)

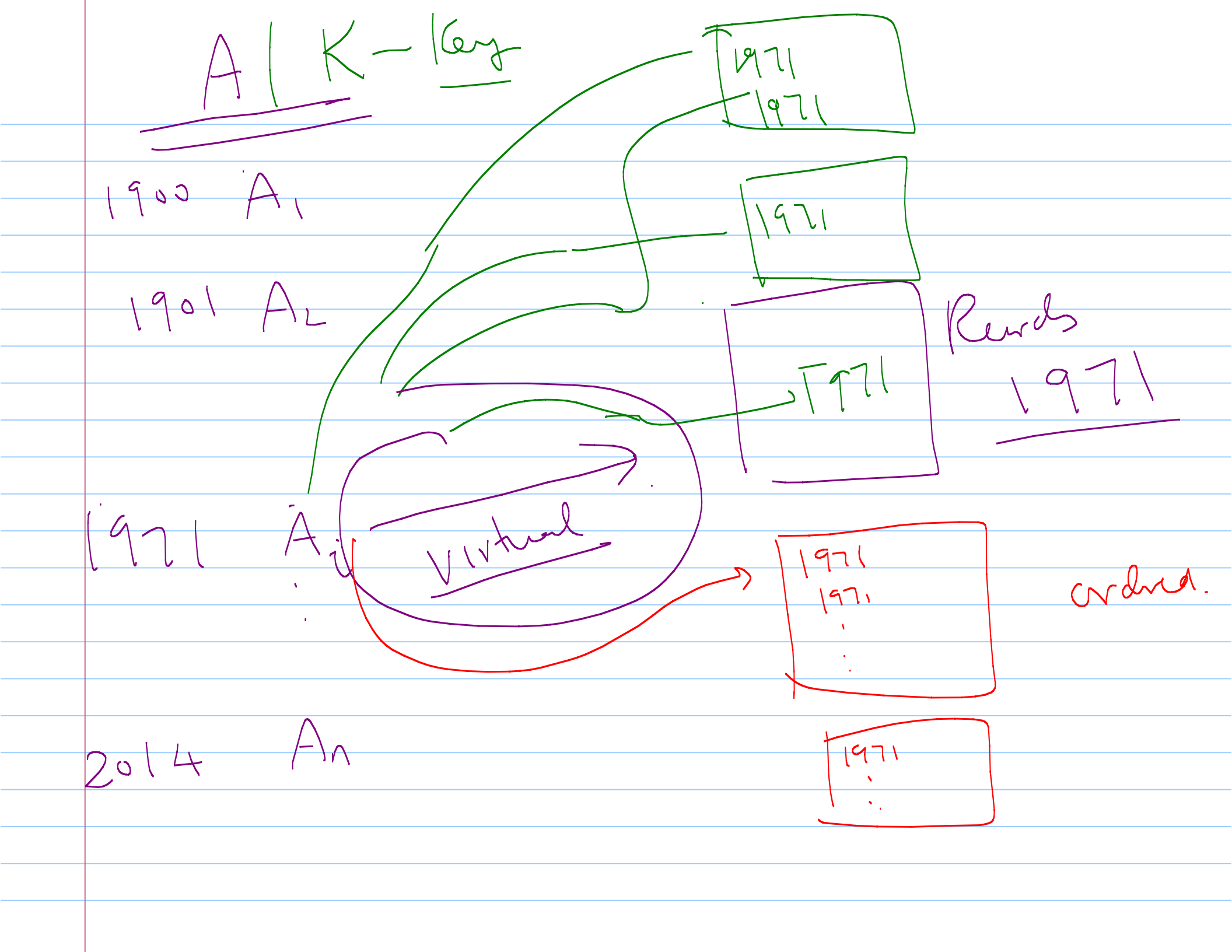
Further, overflowing will cause use of hash function $h_2(K) = K \bmod (4M)$ to be used to split the buckets

In general, a sequence of hash functions $h_j(K) = 2^jM$ is used, where j=0,1,2, …

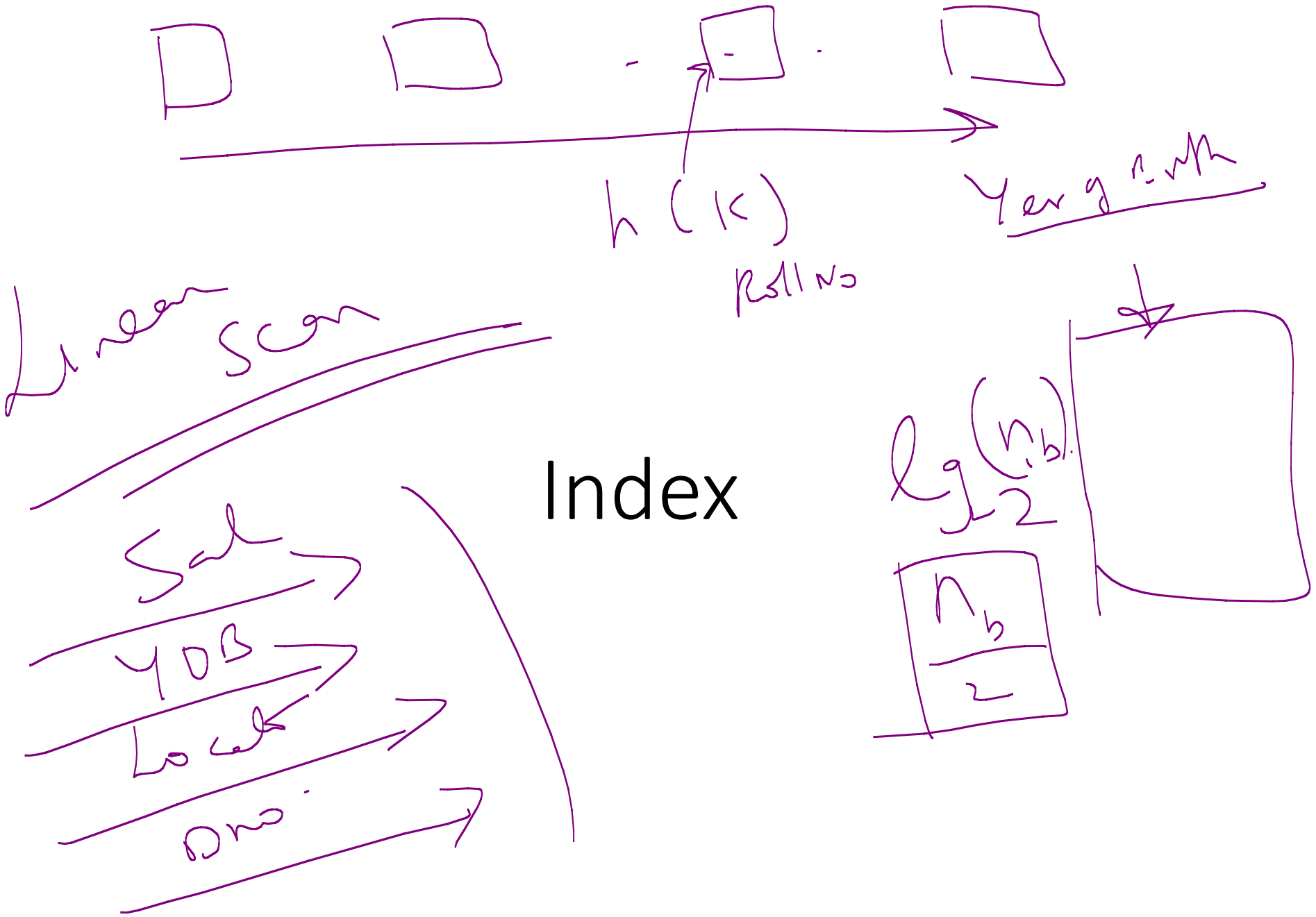
The new hash function $h_{j+1}(K)$ is used when all the buckets 0, 1, 2, …, 2^jM have been split

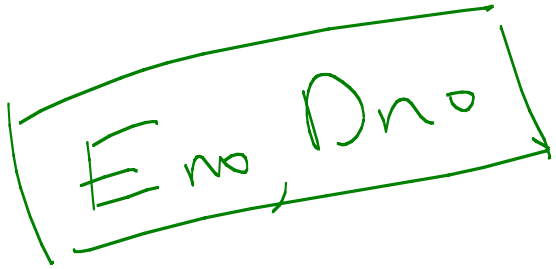
Buckets that have been split can be combined if the load of the file falls below some threshold. The file load factor l=r/(bfr*N), r is number of current records, bfr is maximum number of records per bucket, and N is the number of current file buckets (M_1)

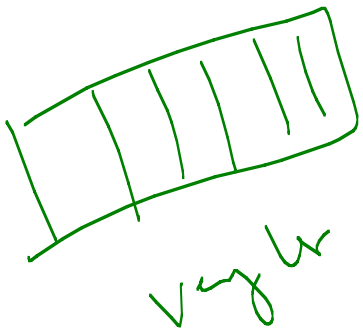
Blocks are combines linearly and M is decremented


A | K - Key
1900 A1
1901 A2
1971 Ai
2014 An
1971
1971
1971
1971
Records
1971
Virtual
1971
1971
ordered.
1971

Index







Index

A single level index is an auxiliary ORDERED file that makes it more efficient to search for a record in the data file

The index is usually specified on one field of the file (although it could be specified on several fields).

One form of an index is a file of entries

<field value, pointer to record>

Which is ordered by field value

The index is called an access path on the fields

The index file usually occupied considerably less disk blocks than the data file because its entries are much smaller

A binary search on the index yields a pointer to the file record

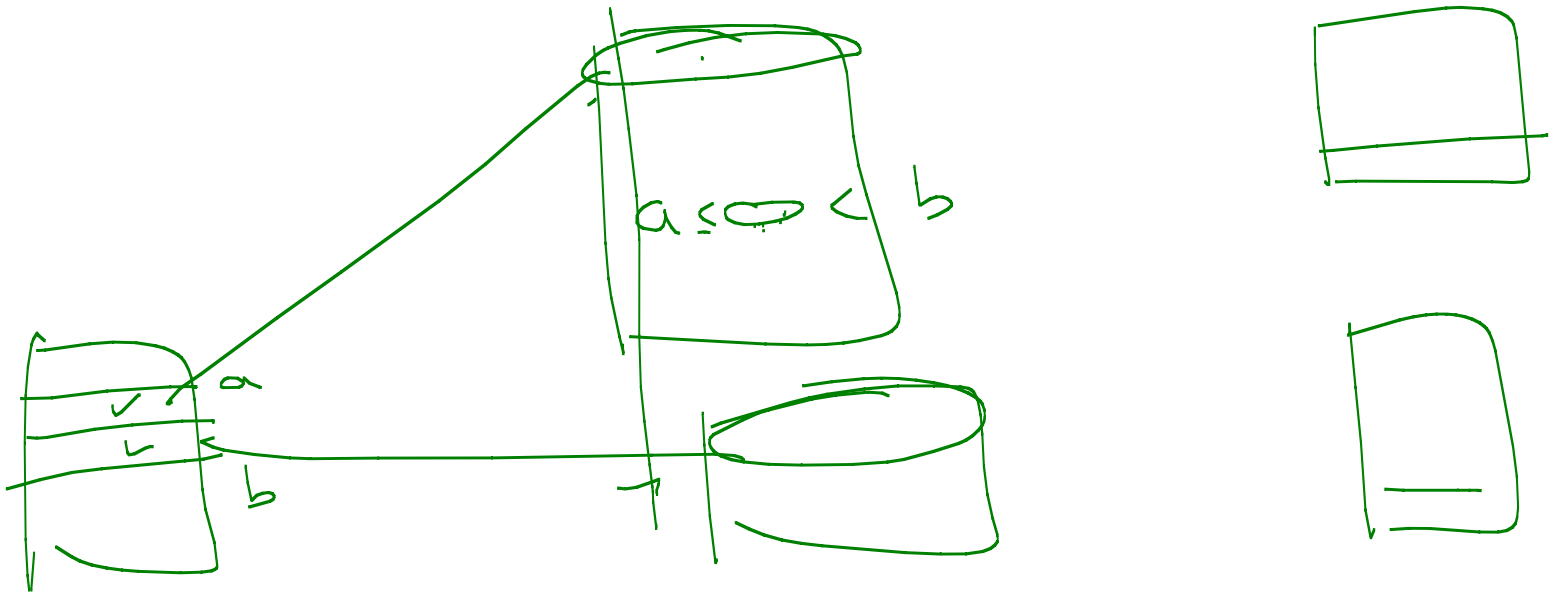
Sparse, Single level, key, Primary Index – file ordered

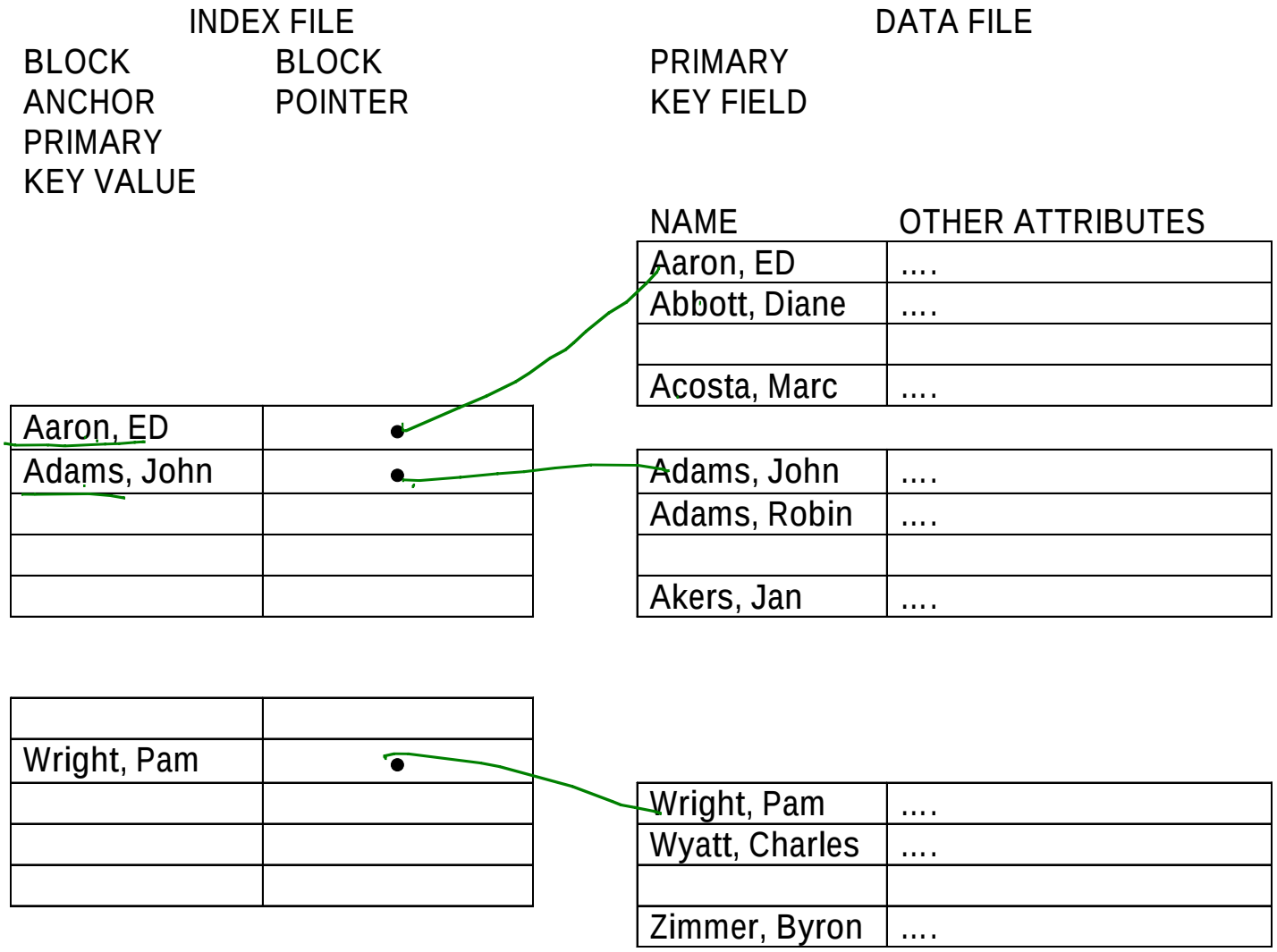
Defined on an ordered data file

The data file is ordered on a key field

Includes one index entry for each block in the data file; the index entry has the key field value for the first record called the block anchor

A similar scheme can use the last record in a block



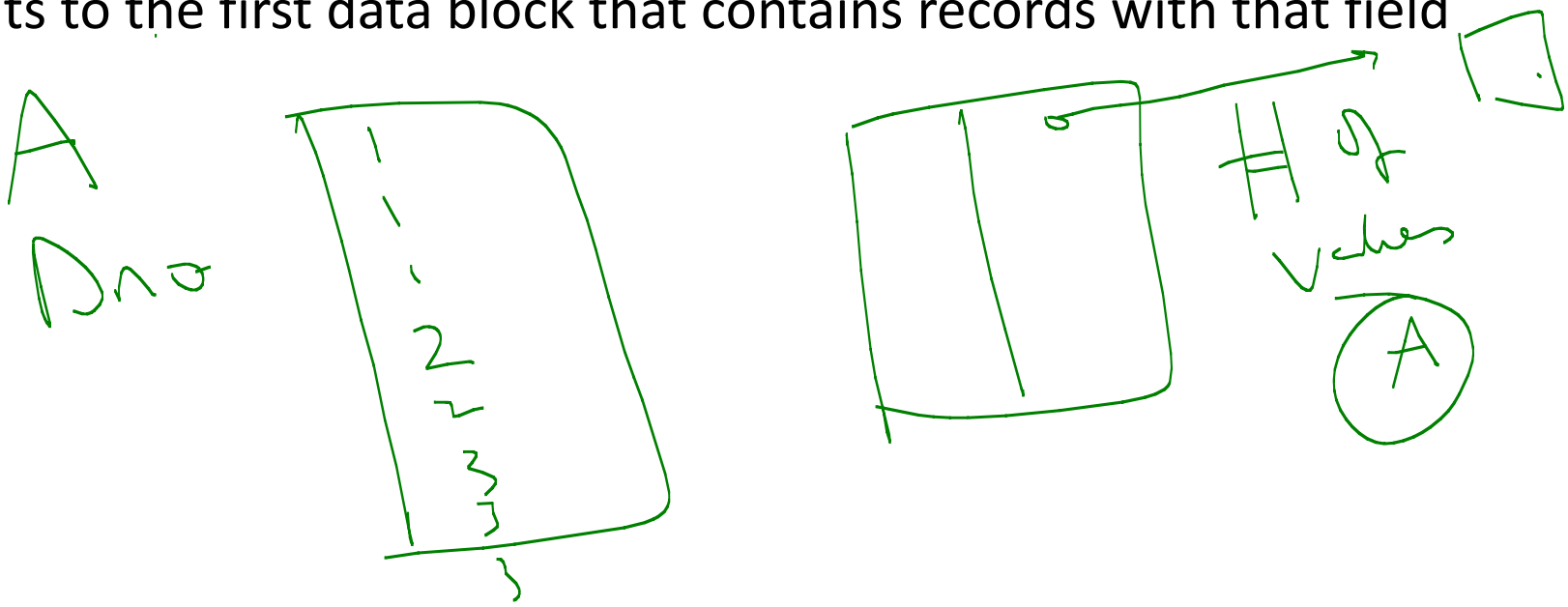

INDEX FILE
BLOCK ANCHOR PRIMARY KEY VALUE
BLOCK POINTER
DATA FILE
PRIMARY KEY FIELD
NAME
OTHER ATTRIBUTES
Aaron, ED
Adams, John
Wright, Pam
Aaron, ED ....
Abbott, Diane ....
Acosta, Marc ....
Adams, John ....
Adams, Robin ....
Akers, Jan ....
Wright, Pam ....
Wyatt, Charles ....
Zimmer, Byron ....

Sparse, single-level, non-key, Clustering Index – file ordered

Defined on an ordered file

The data file is ordered on a non-key field

Includes one index entry for each distinct value of the field; the index entry points to the first data block that contains records with that field value



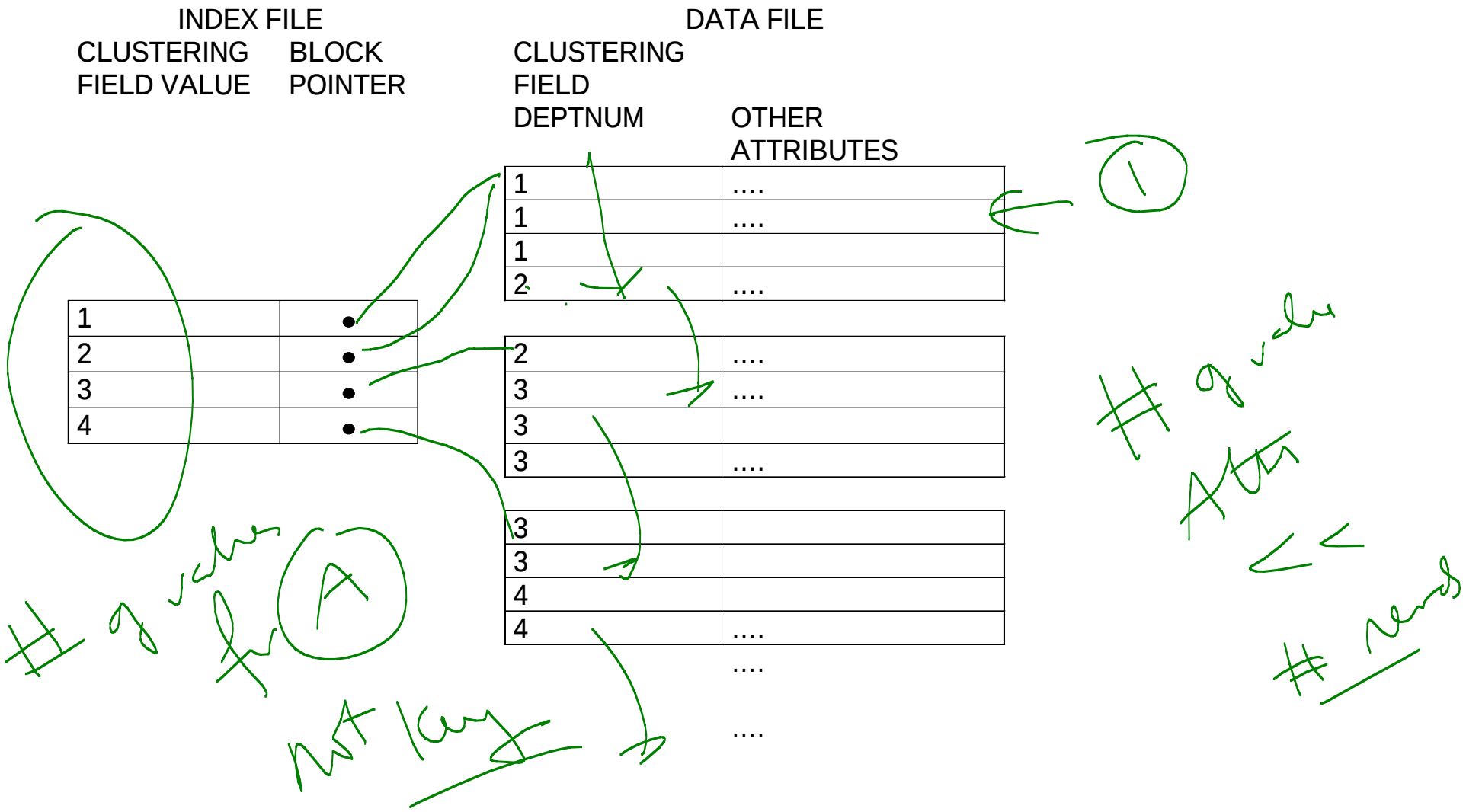
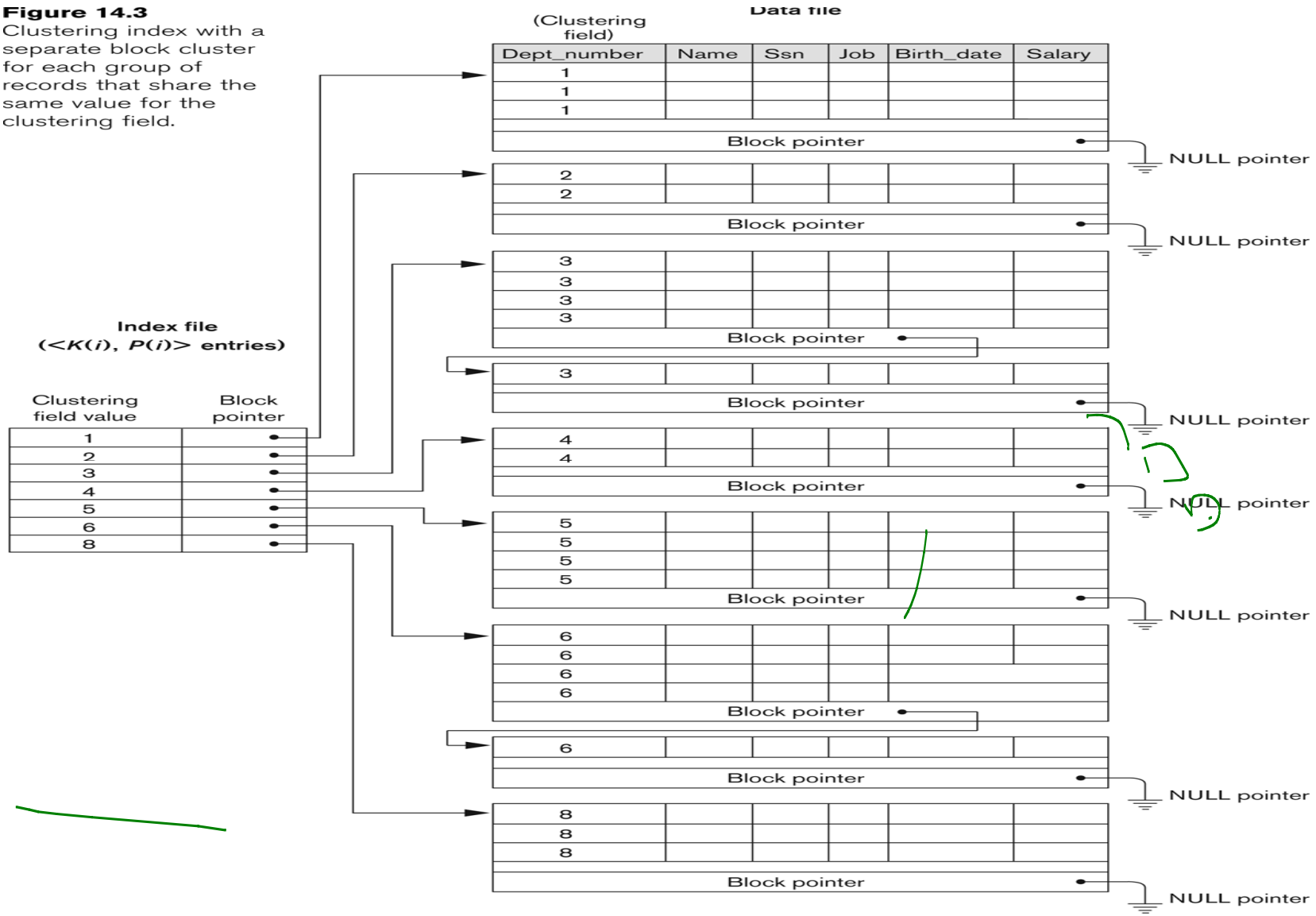

INDEX FILE
CLUSTERING FIELD VALUE
BLOCK POINTER
DATA FILE
CLUSTERING FIELD DEPTNUM
OTHER ATTRIBUTES
1
2
3
4
1 ....
1 ....
1
2 ....
2 ....
3 ....
3
3 ....
3
3
4
4 ....
....
....

Figure 14.3
Clustering index with a separate block cluster for each group of records that share the same value for the clustering field.

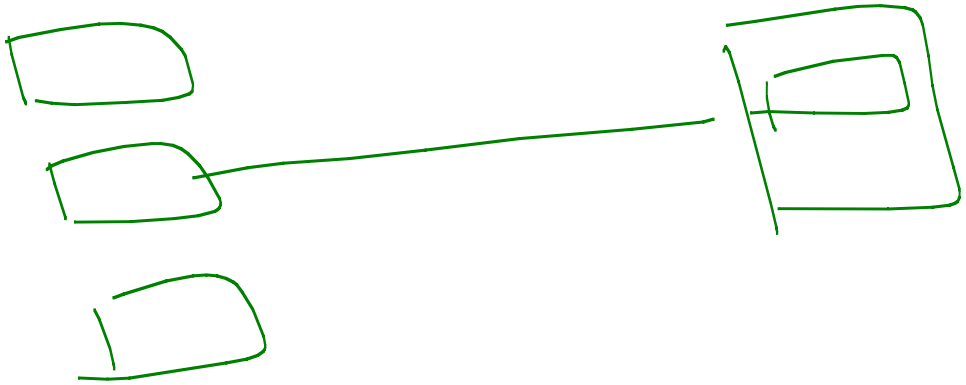


Dense, single-level, key, Secondary Index – file unordered

Defines on an unordered data file

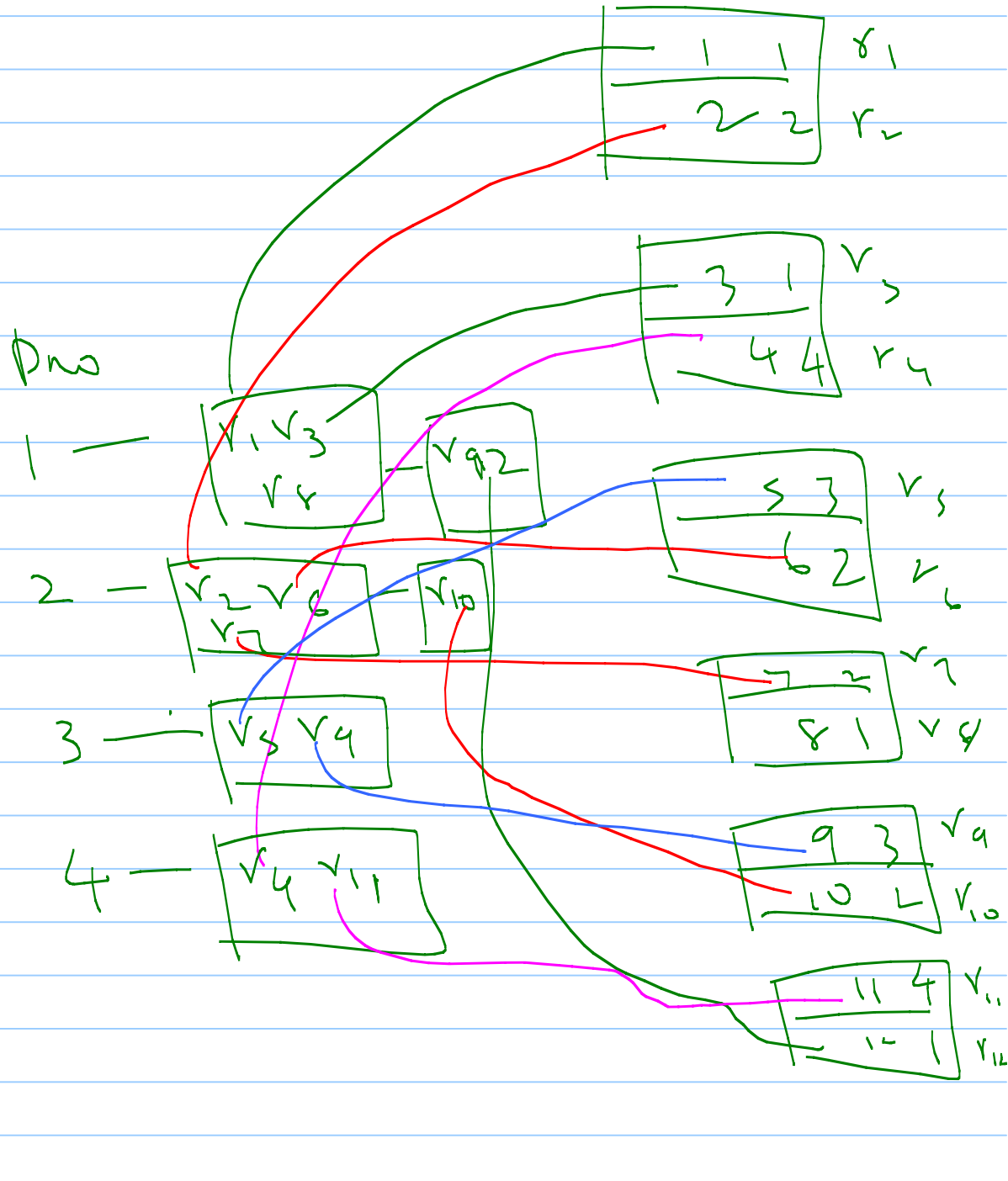
Can be defined on a key field or a non-key field

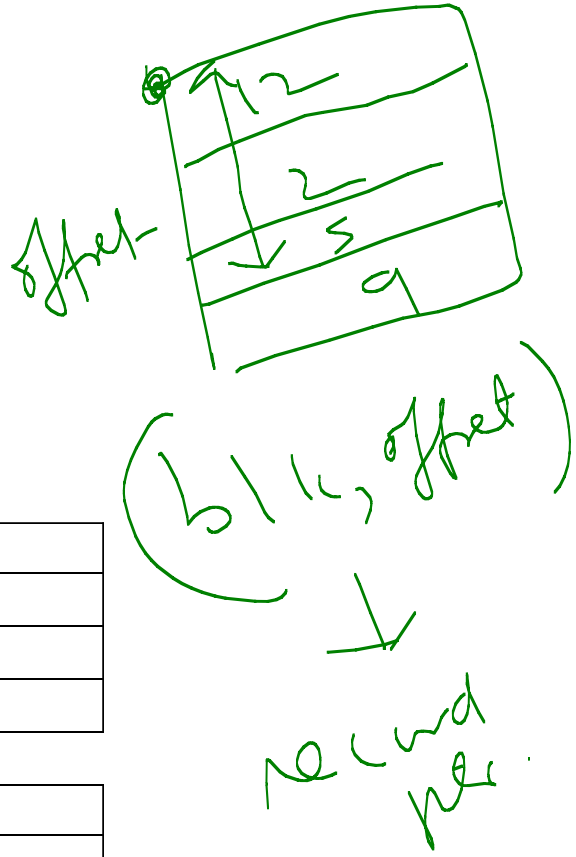
Includes one entry for each record in the data file; hence, it is called a dense index



Non key index on Dno

Eno	Dno
1	1
2	2
3	1
4	4
5	3
6	2
7	2
8	1
9	3
10	2
11	4
12	1





INDEX FILE

CLUSTERING FIELD VALUE	BLOCK POINTER
1	•
2	•
3	•
4	•
5	•
6	•
7	•
8	•
9	•
10	•
11	•
12	•

DATA FILE

INDEXING FIELD (secondary key field)	
8	....
3	....
7	
6	....
12	....
2	....
5	
9	....
1	
11	
10	
4	....
	....

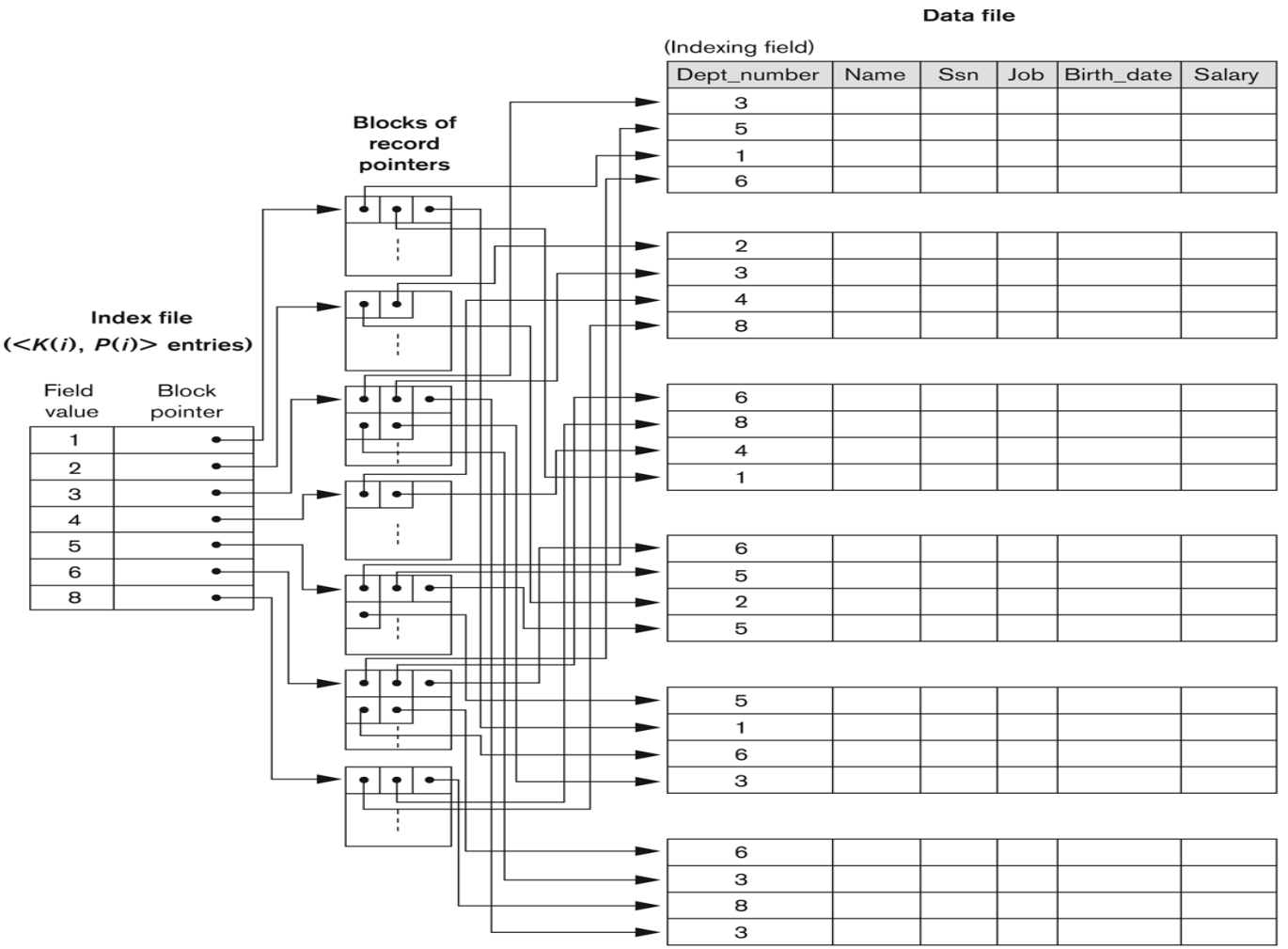
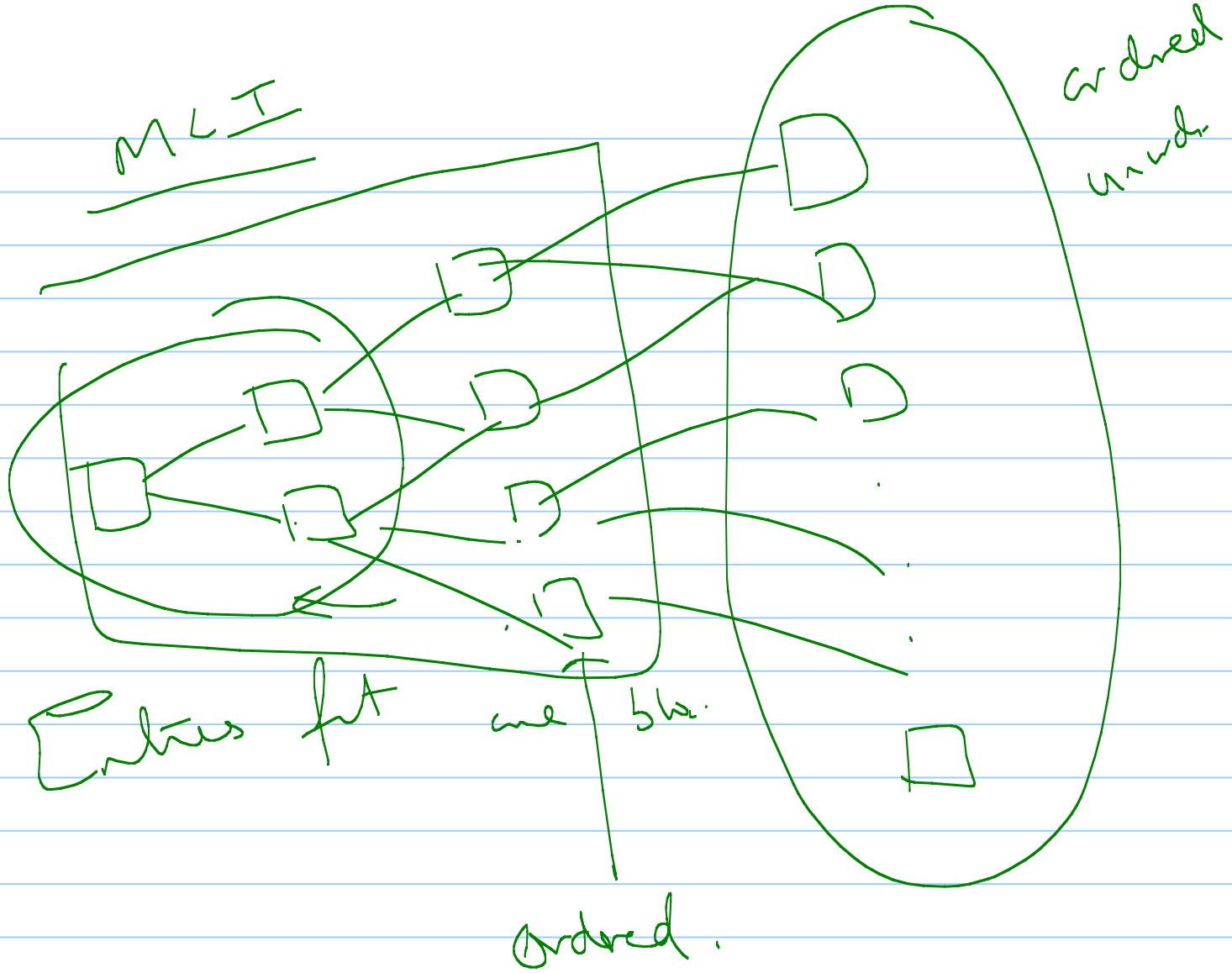


Figure 14.5
A secondary index (with record pointers) on a nonkey field implemented using one level of indirection so that index entries are of fixed length and have unique field values.

Table 14.2
Properties of Index Types

Type of Index	Number of (First-level) Index Entries	Dense or Nondense	Block Anchoring on the Data File
Primary	Number of blocks in data file	Nondense	Yes
Clustering	Number of distinct index field values	Nondense	Yes/no[a]
Secondary (key)	Number of records in data file	Dense	No
Secondary (nonkey)	Number of records[b] or number of distinct index field values[c]	Dense or Nondense	No

[a]Yes if every distinct value of the ordering field starts a new block; no otherwise.
[b]For option 1.
[c]For options 2 and 3.


MLI
Ordered
Entries left
Ordered.

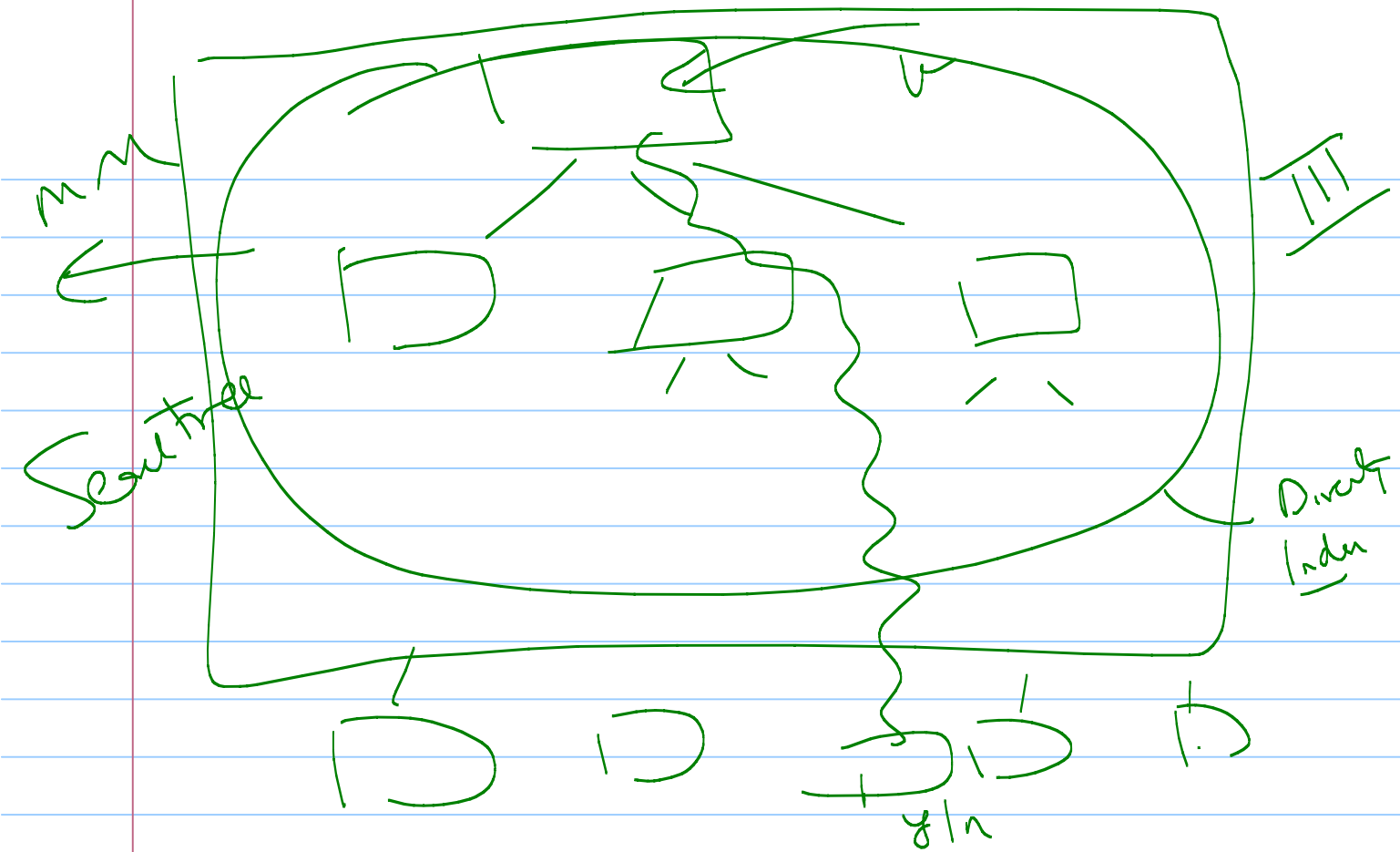
Multi- level Indexes

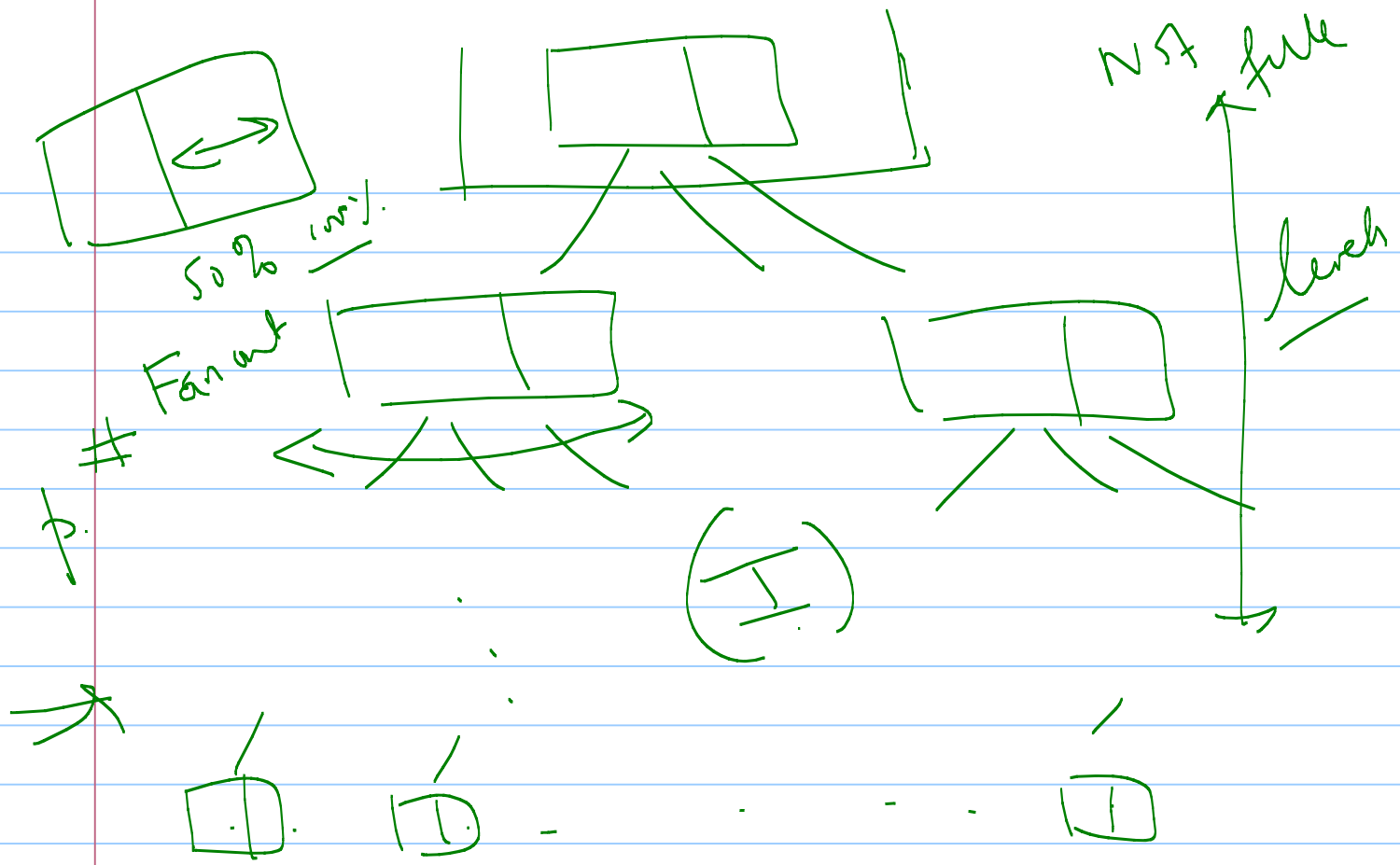
Because a single-level index is an ordered file, we can create a primary index to the index itself; in this case, the original index file is called the first level index, and the index to the index is called the second-level index

We can repeat the process, creating a third, fourth, …, top level until all entries of the top level fit in one disk block

A multi-level index can be created for any type of first-level index (primary, secondary, clustering) as long as the first –level index consists of more than one disk block

Such a multi-level index is a form of a search tree; however, insertion and deletion of new index entries is a severe problem because every level of index is an ordered file


Search
Direct
Index
y/n


50% init.
Fanout
p.
N ↑ full
levels

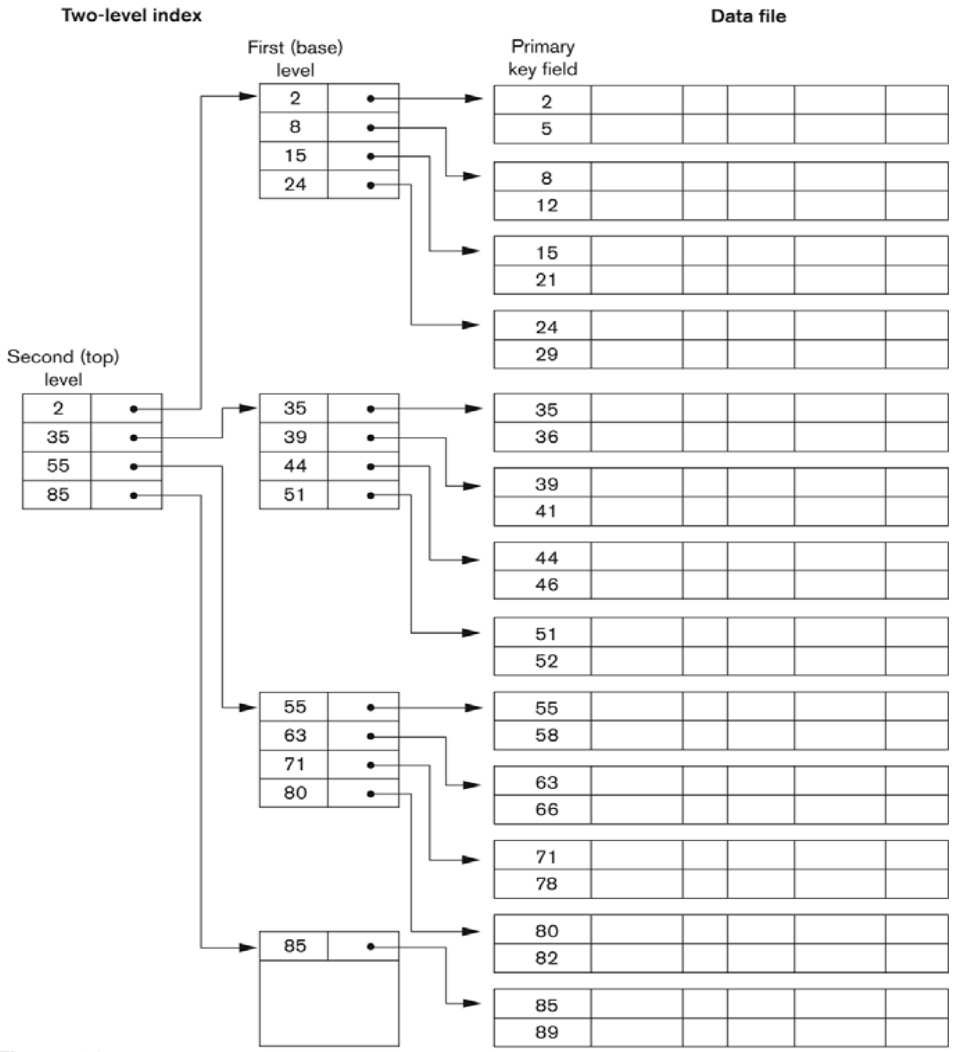
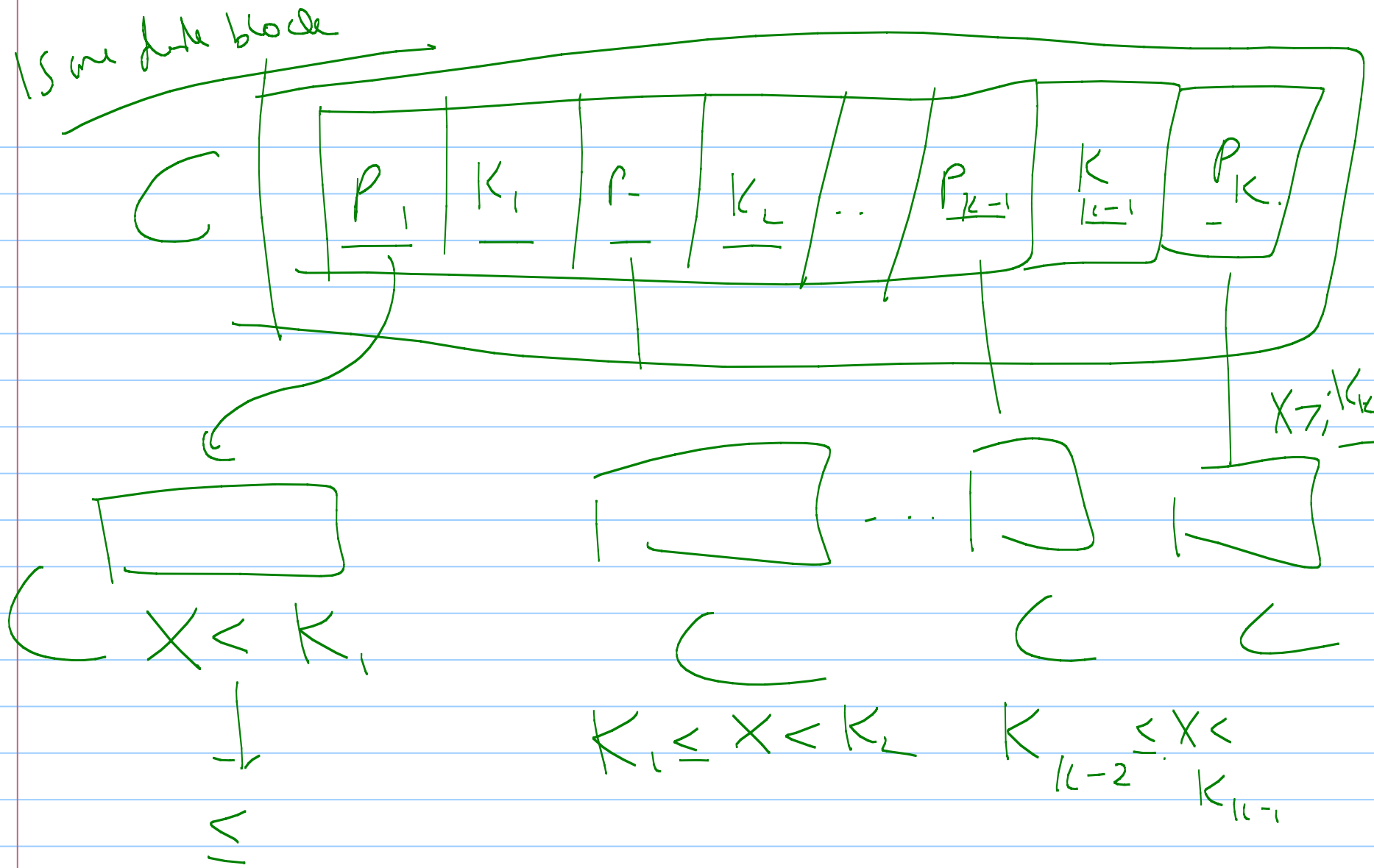
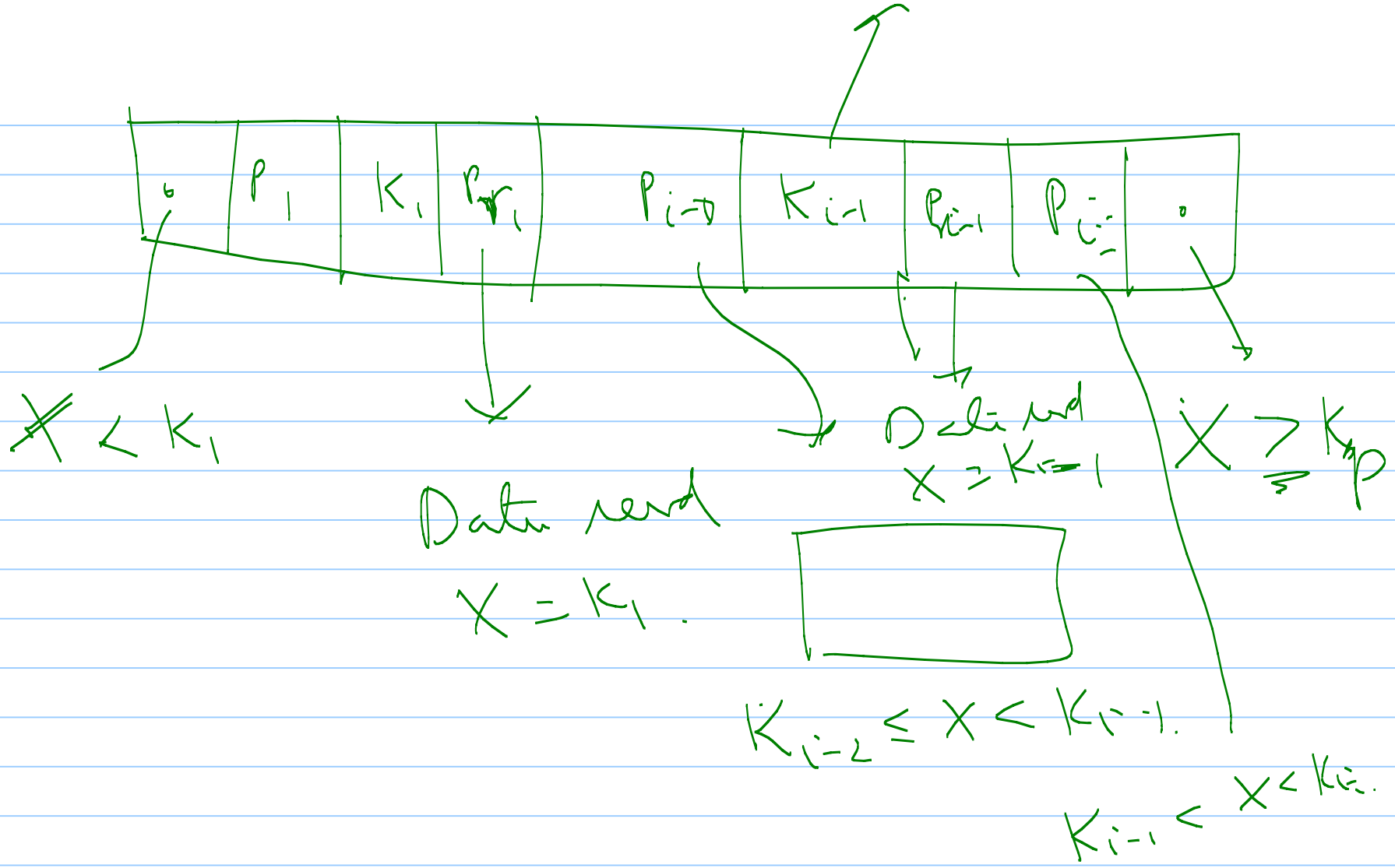


Figure 14.6
A two-level primary index resembling ISAM (Index Sequential Access Method) organization.

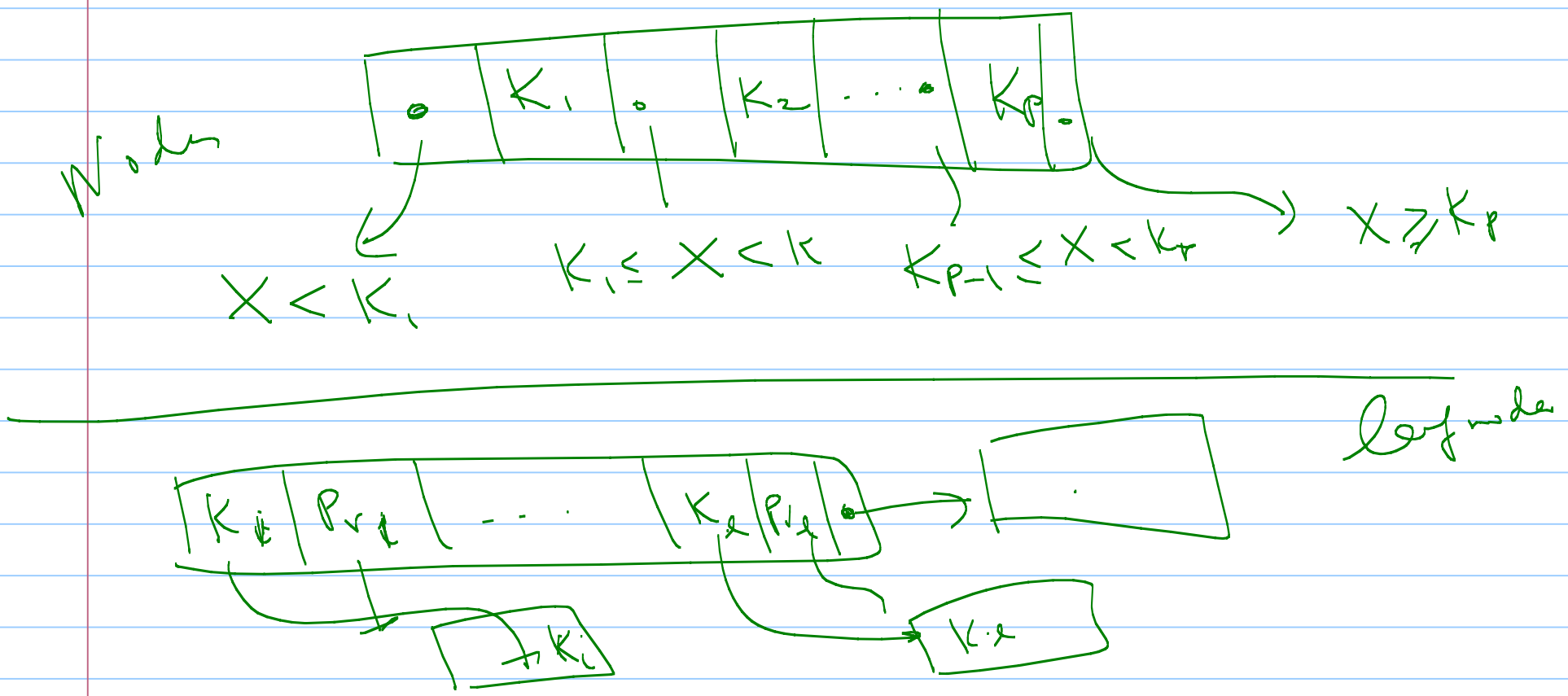
Multi-level Indexes

Because of the insertion and deletion problem, most multi-level indexes use B-tree or B+ tree data structures, which leave space in each tree node (disk block) to allow for new index entries


Is an index block
P_1
K_1
P_2
K_2
P_{k-1}
K_{k-1}
P_k
$X < K_1$
$K_1 \leq X < K_2$
$K_{k-2} \leq X < K_{k-1}$
$X \geq K_{k-1}$


P_0
P_1
K_1
P_{r_1}
P_{i-1}
K_{i-1}
P_i
K_i
P_{i+1}
P_p
$X < K_1$
Data read
$X = K_1$.
Data read
$X = K_{i-1}$
$X \geq K_p$
$K_{i-2} \leq X < K_{i-1}$.
$K_{i-1} < X < K_i$.

B-tree



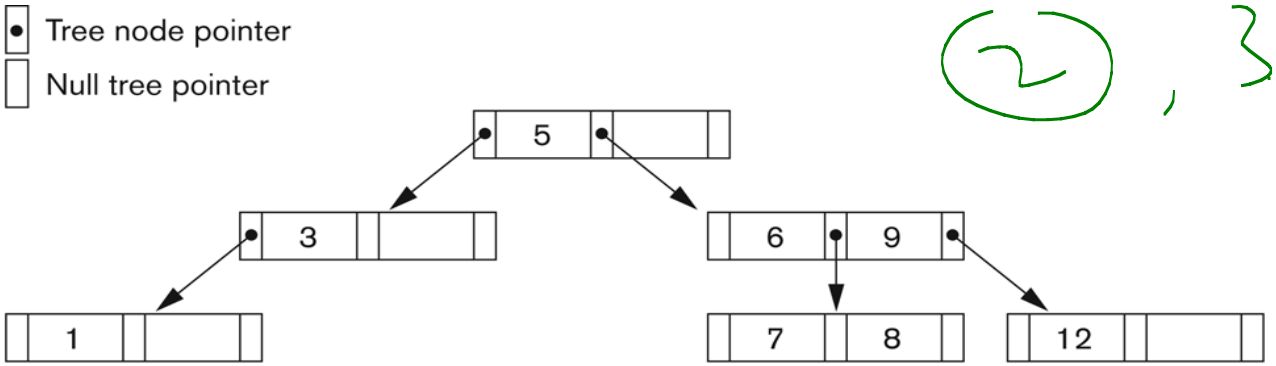


Figure 14.9
A search tree of order $p = 3$.

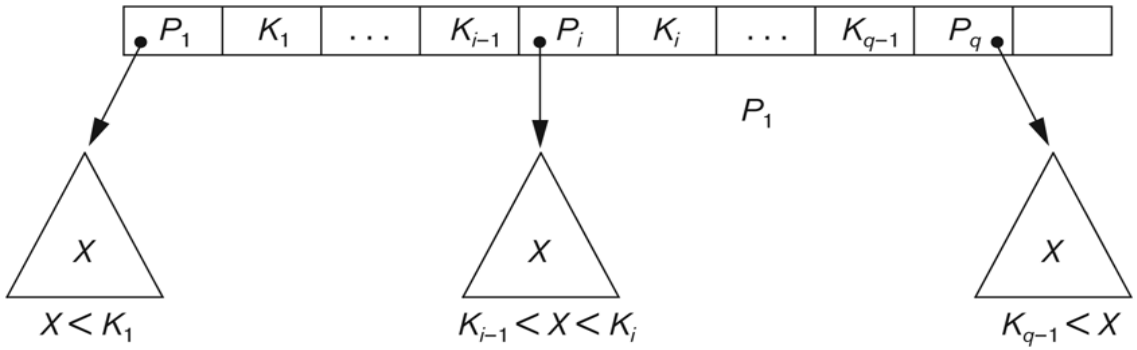


Figure 14.8
A node in a search tree with pointers to subtrees below it.

B Trees and B+ Trees as Dynamic Multi-level Indexes

These data structures are variations of search tress that allow efficient insertion and deletion of new search values

In B tree and B+ tree data structures, each node corresponds to a disk block/page

Each node is kept between half-full and completely full

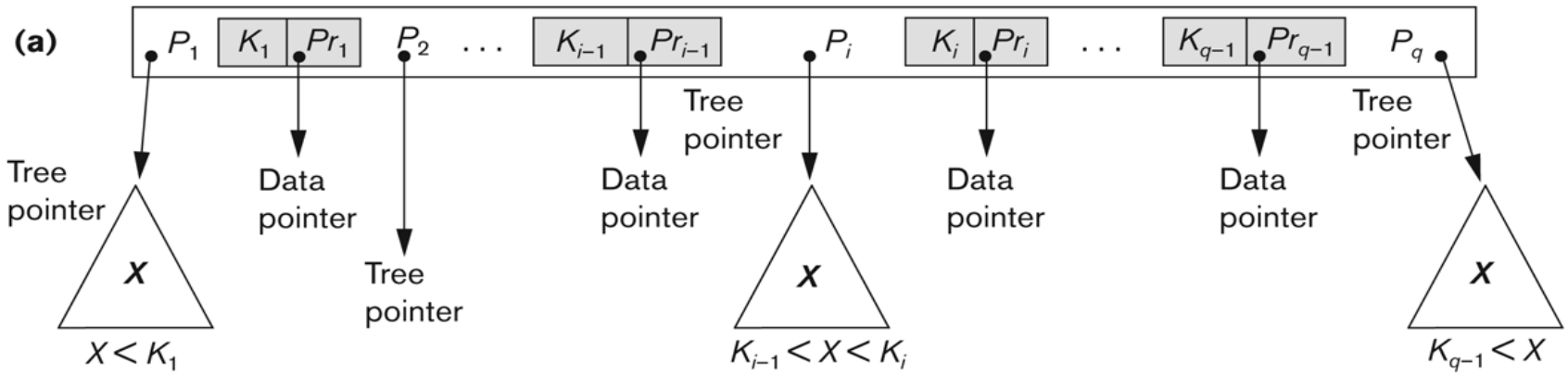
An insertion onto a node that is not full is quite efficient; if a node is full insertion causes a split into two nodes

Splitting may propagate to other tree levels

A deletion is quite efficient if a node does not become less than half full

Figure 14.10

B-Tree structures. (a) A node in a B-tree with $q - 1$ search values. (b) A B-tree of order $p = 3$.The values were inserted in the order 8, 5, 1, 7, 3, 12, 9, 6.



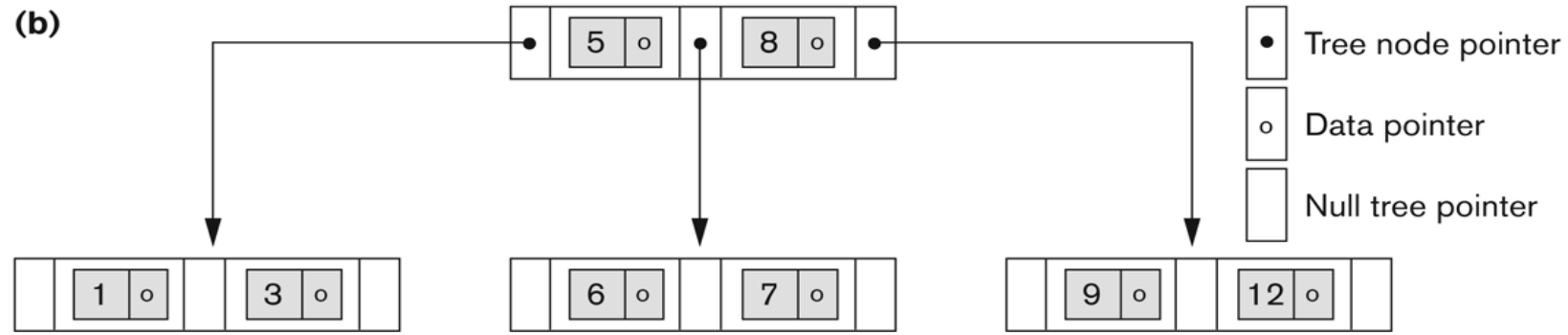
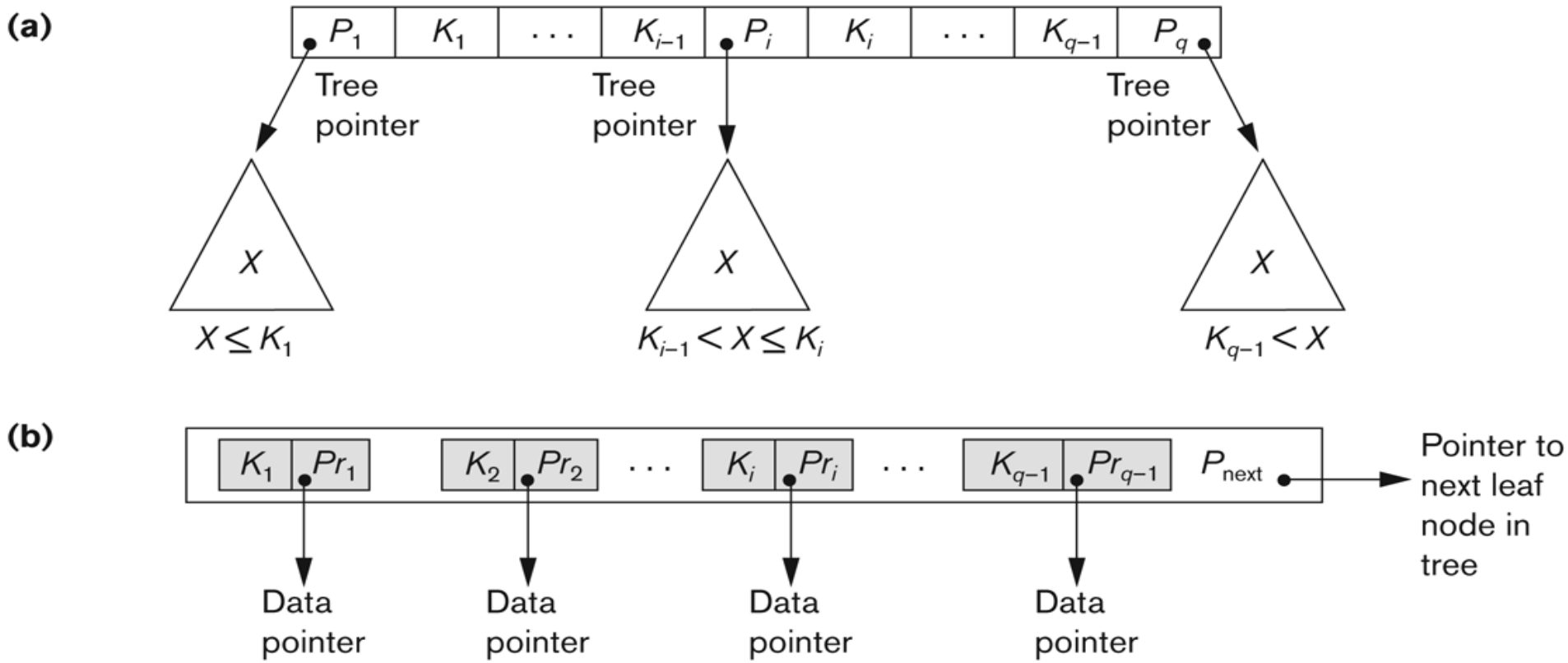


Figure 14.11
The nodes of a B+-tree. (a) Internal node of a B+-tree with $q - 1$ search values. (b) Leaf node of a B+-tree with $q - 1$ search values and $q - 1$ data pointers.



B Trees and B+ Trees as Dynamic Multi=level Indexes

If a deletion causes a node to become less than half full, it must be merged with neighboring nodes

Difference between B tree and B+ tree

- in a B tree, pointers to data nodes exist at all levels of the tree
- in a B+ tree, all pointers to data records exist at only leaf-level nodes
- B+ tree can have less levels (or higher capacity of search values) than the corresponding B tree

Multi-dimension index structure

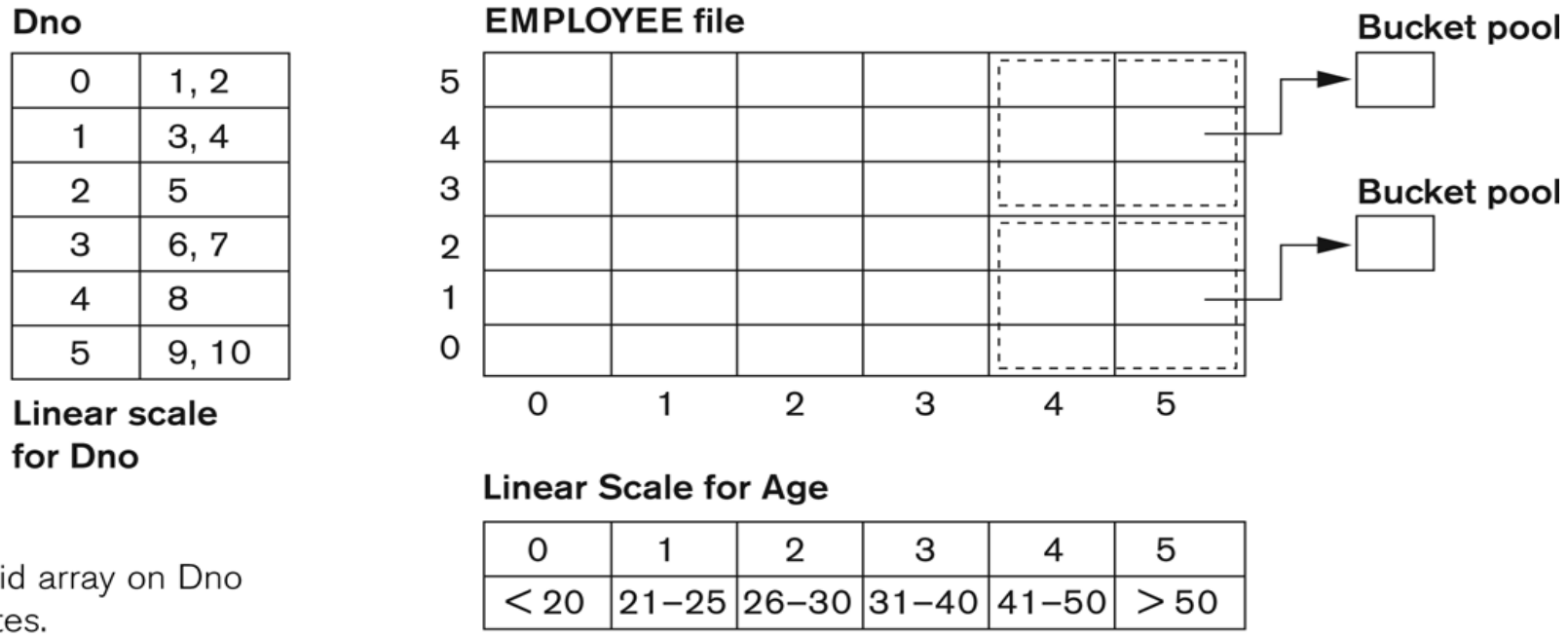


Figure 14.14
Example of a grid array on Dno and Age attributes.