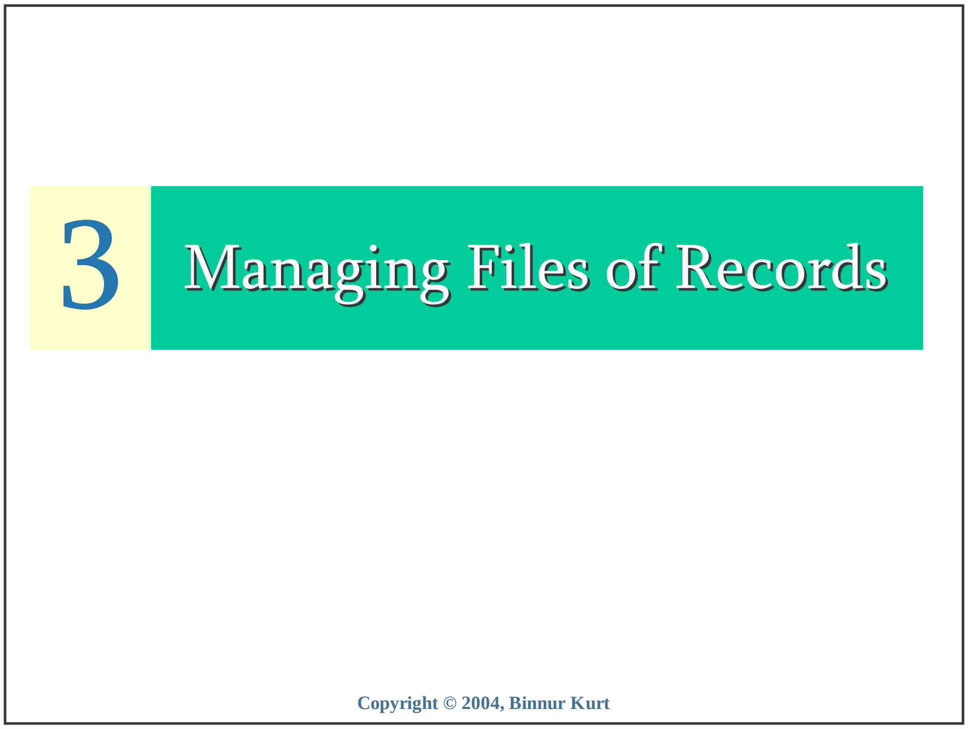
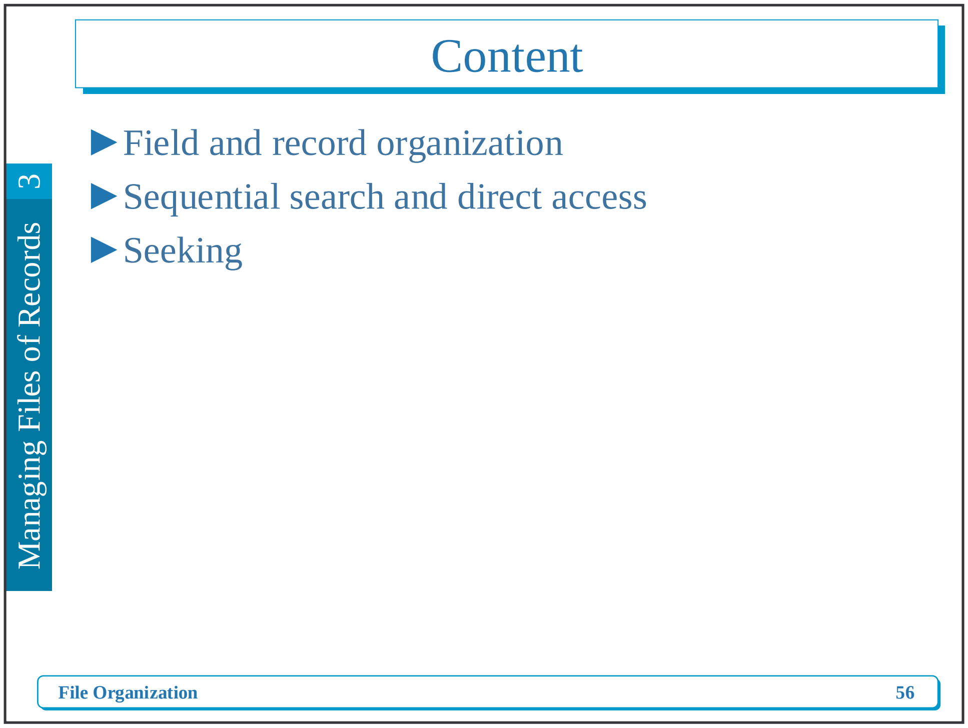
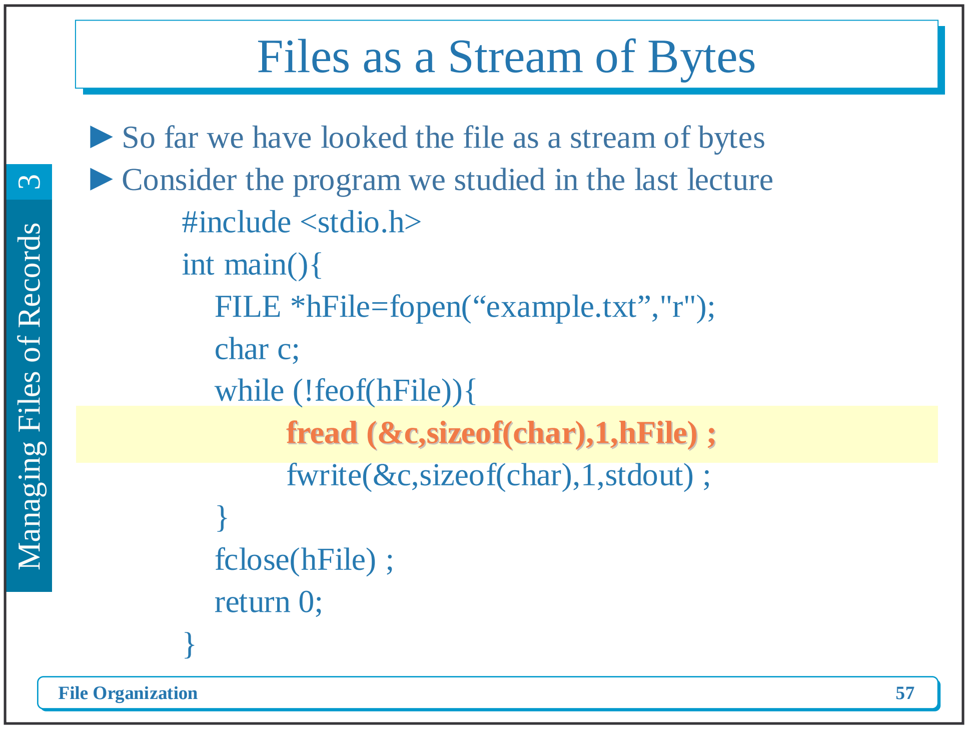


3 Managing Files of Records



Content

- ▶ Field and record organization
- ▶ Sequential search and direct access
- ▶ Seeking

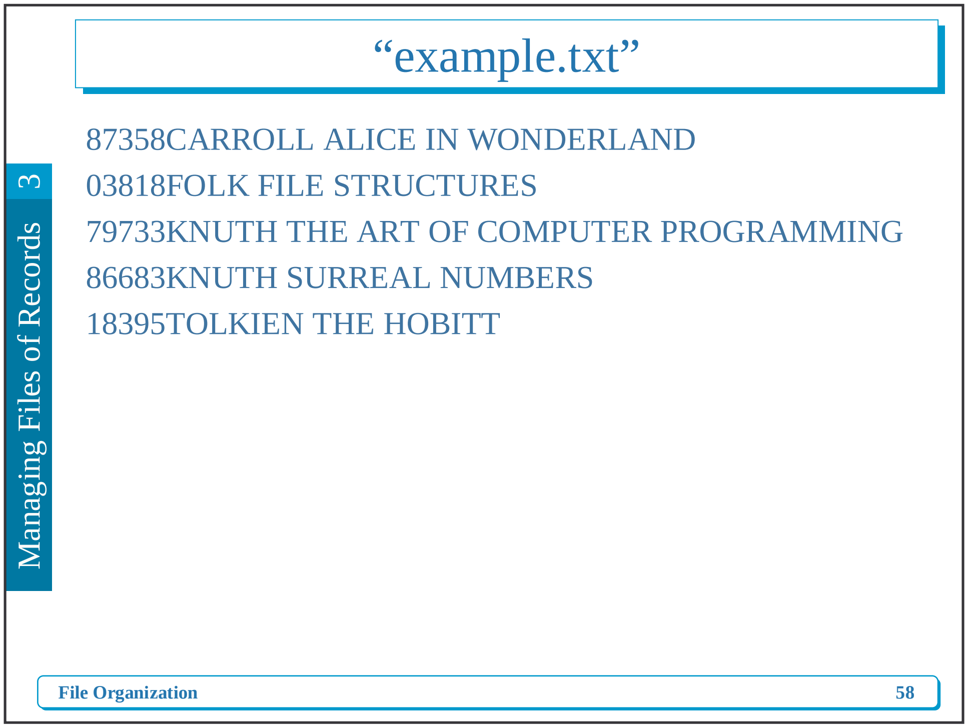


Managing Files of Records 3

Files as a Stream of Bytes

- ▶ So far we have looked the file as a stream of bytes
- ▶ Consider the program we studied in the last lecture

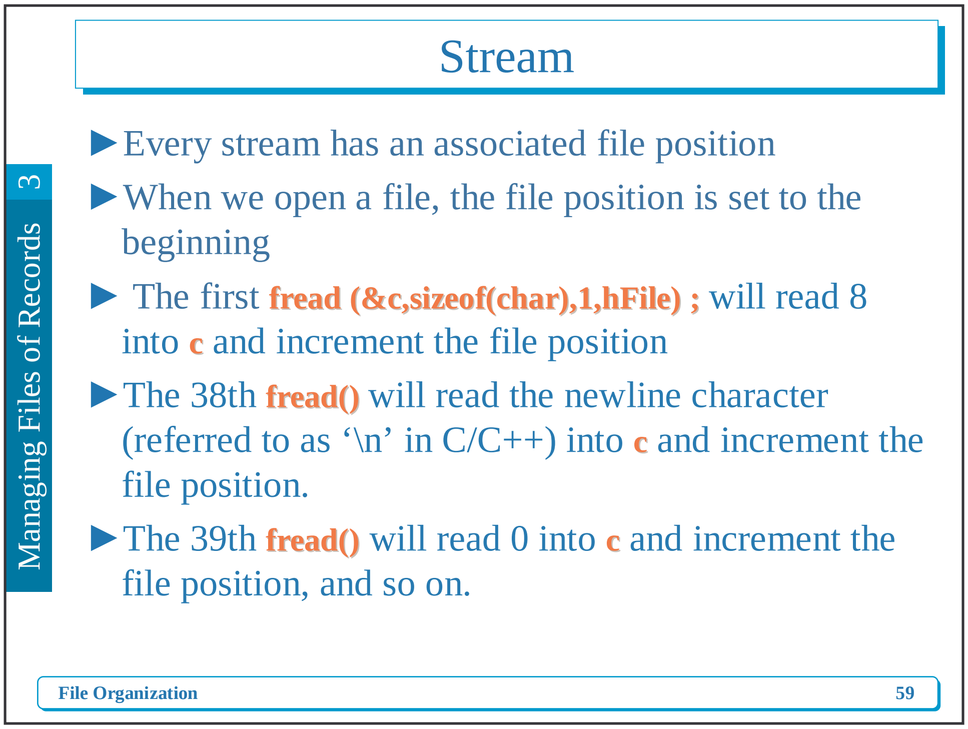
```
#include <stdio.h>
int main(){
    FILE *hFile=fopen(“example.txt”,"r");
    char c;
    while (!feof(hFile)){
            fread (&c,sizeof(char),1,hFile) ;
            fwrite(&c,sizeof(char),1,stdout) ;
    }
    fclose(hFile) ;
    return 0;
}
```



Managing Files of Records 3

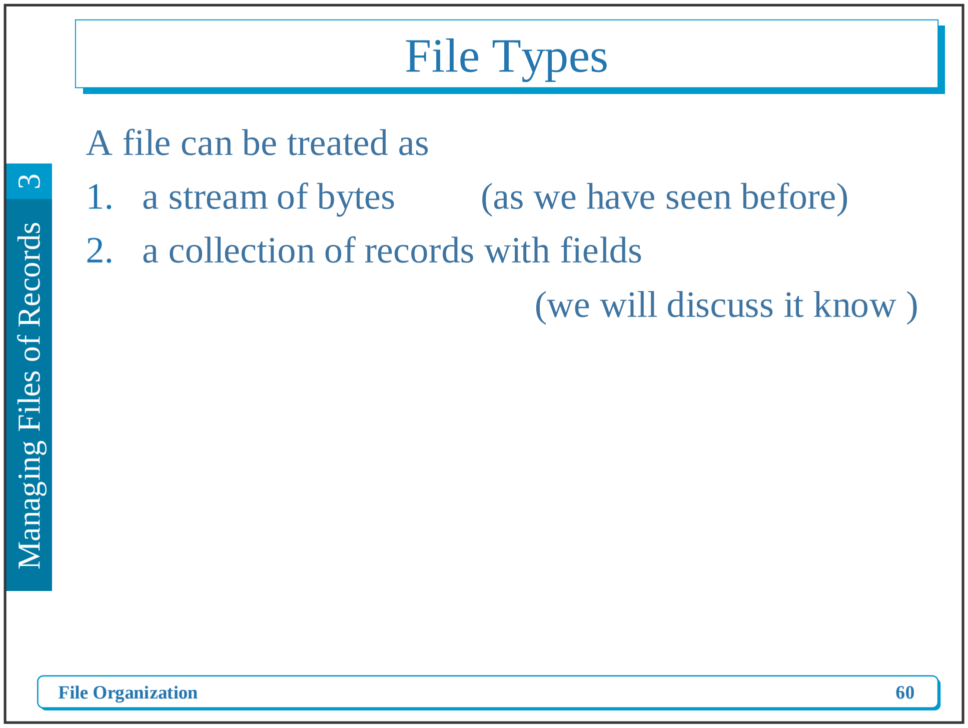
“example.txt”

```
87358CARROLL ALICE IN WONDERLAND
03818FOLK FILE STRUCTURES
79733KNUTH THE ART OF COMPUTER PROGRAMMING
86683KNUTH SURREAL NUMBERS
18395TOLKIEN THE HOBITT
```



Stream

- Every stream has an associated file position
- When we open a file, the file position is set to the beginning
- The first **fread (&c,sizeof(char),1,hFile) ;** will read 8 into **c** and increment the file position
- The 38th **fread()** will read the newline character (referred to as ‘\n’ in C/C++) into **c** and increment the file position.
- The 39th **fread()** will read 0 into **c** and increment the file position, and so on.



File Types

A file can be treated as

1. a stream of bytes (as we have seen before)
2. a collection of records with fields

(we will discuss it know)

Field and Record Organization

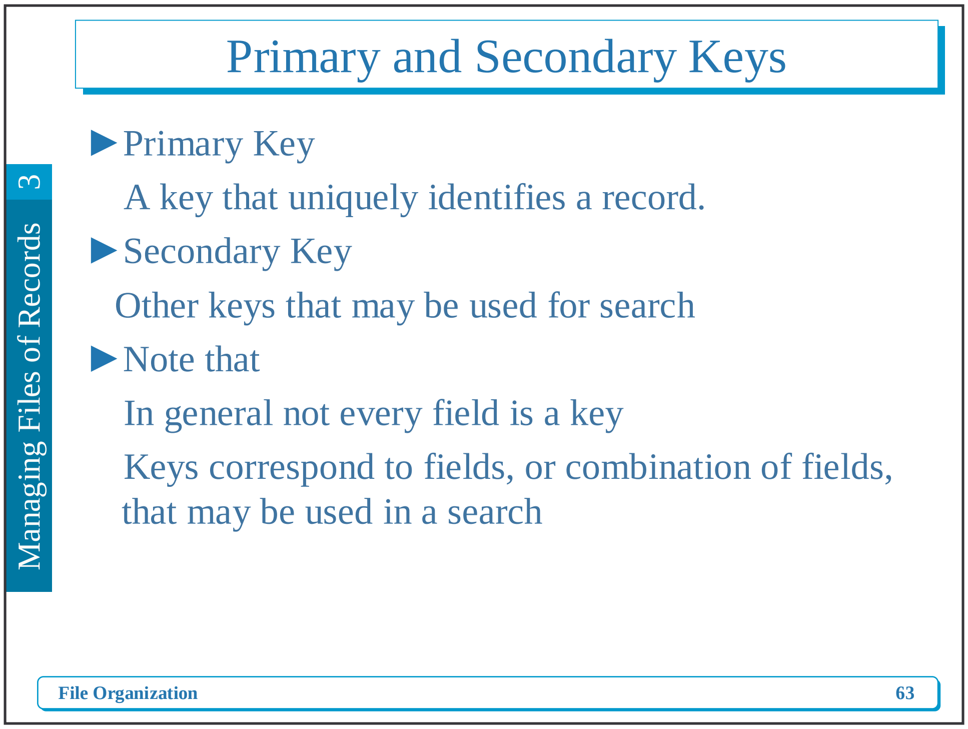
- ▶ Field: a data value, smallest unit of data with logical meaning
- ▶ Record: A group of fields that forms a logical unit
- ▶Key: a subset of the fields in a record used to uniquely identify the record

Memory	File
object	record
member	field

```
87358CARROLL ALICE IN WONDERLAND
03818FOLK FILE STRUCTURES
79733KNUTH THE ART OF COMPUTER PROGRAMMING
86683KNUTH SURREAL NUMBERS
18395TOLKIEN THE HOBITT
```

In our example, “example.txt” contains information about books:

- ▶Each line of the file is a ***record***.
- ▶Fields in each record:
  - – ISBN Number,
  - – Author Name,
  - – Book Title



Primary and Secondary Keys

- Primary Key

  A key that uniquely identifies a record.

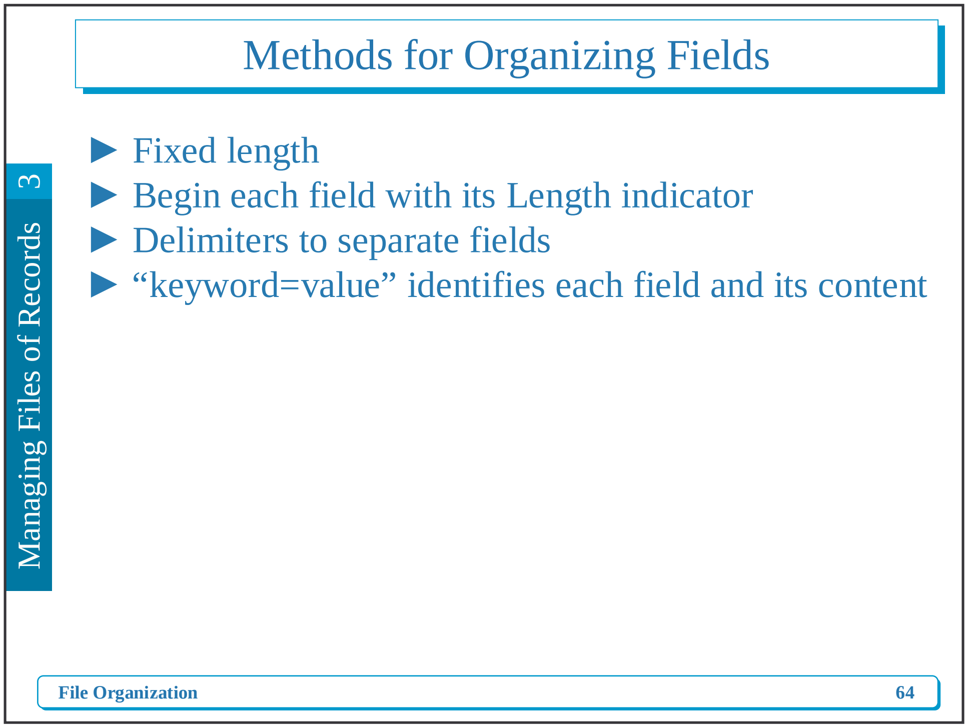
- Secondary Key

  Other keys that may be used for search

- Note that

  In general not every field is a key

  Keys correspond to fields, or combination of fields, that may be used in a search



Methods for Organizing Fields

- Fixed length
- Begin each field with its Length indicator
- Delimiters to separate fields
- “keyword=value” identifies each field and its content

Fixed-Length Fields

Like in our file of books (field lengths are 5,7, and 25).

```
87358CARROLL ALICE IN WONDERLAND
03818FOLK    FILE STRUCTURES
86683KNUTH   SURREAL NUMBERS
18395TOLKIEN THE HOBITT
```

Length indicator

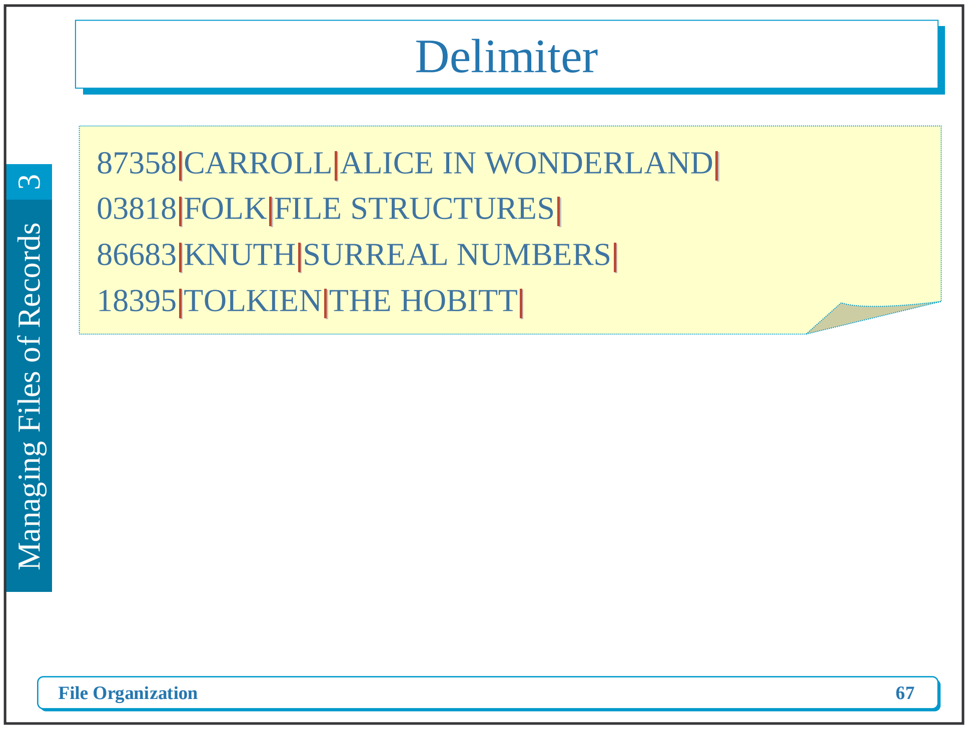
Like in our file of books (field lengths are 5,7, and 25).

0587358**07**CARROLL**19**ALICE IN WONDERLAND

0503818**04**FOLK**15**FILE STRUCTURES

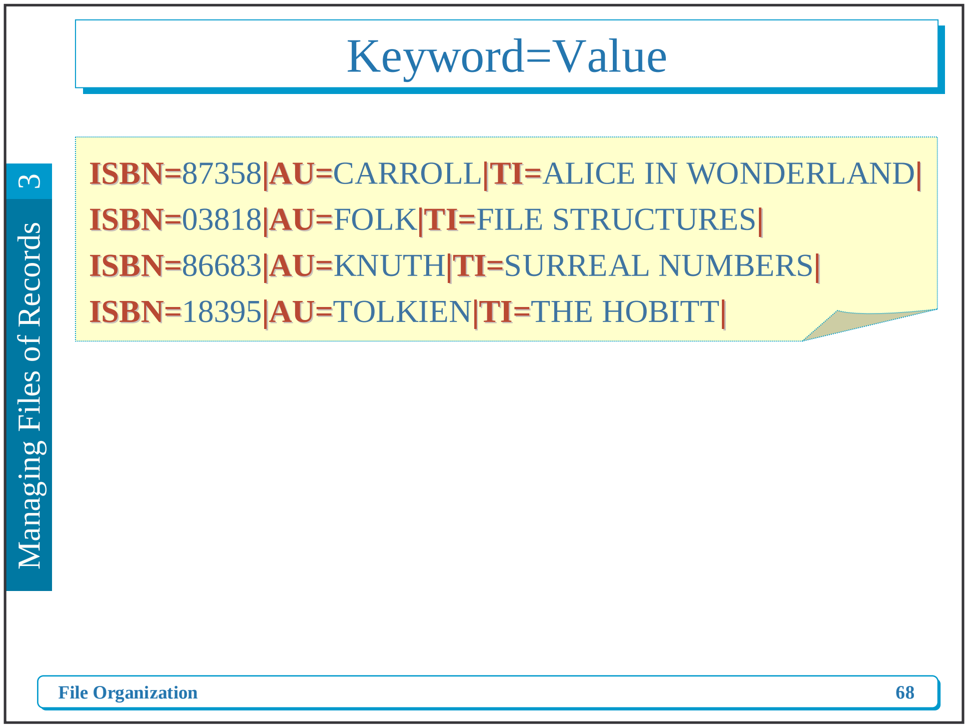
0586683**05**KNUTH**15**SURREAL NUMBERS

0518395**07**TOLKIEN**10**THE HOBITT



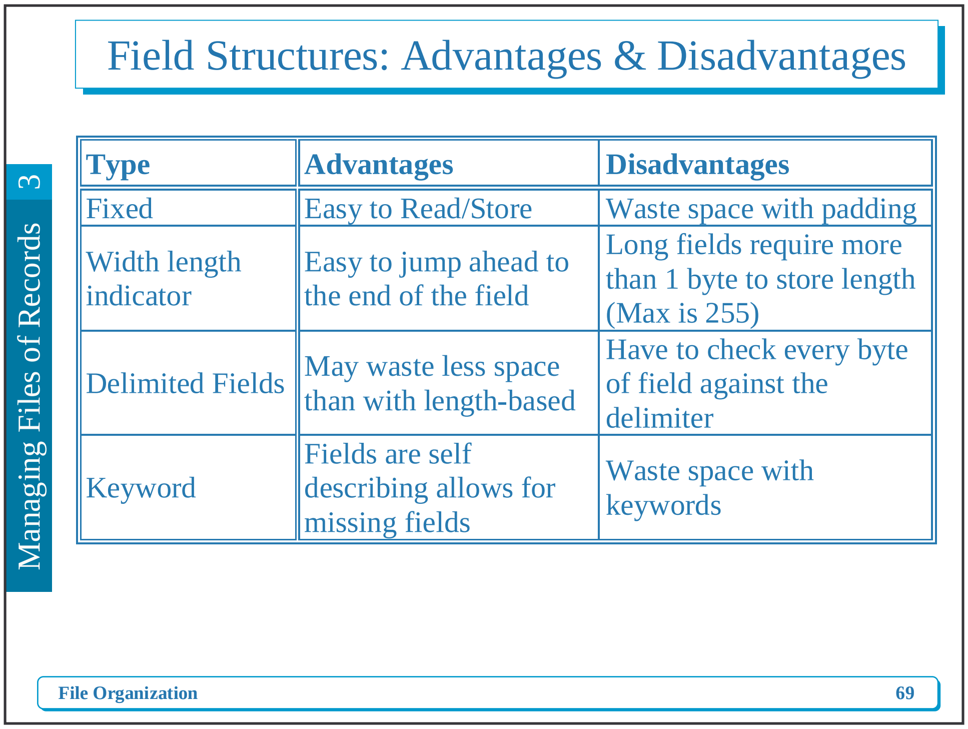
Delimiter

87358|CARROLL|ALICE IN WONDERLAND|
03818|FOLK|FILE STRUCTURES|
86683|KNUTH|SURREAL NUMBERS|
18395|TOLKIEN|THE HOBITT|



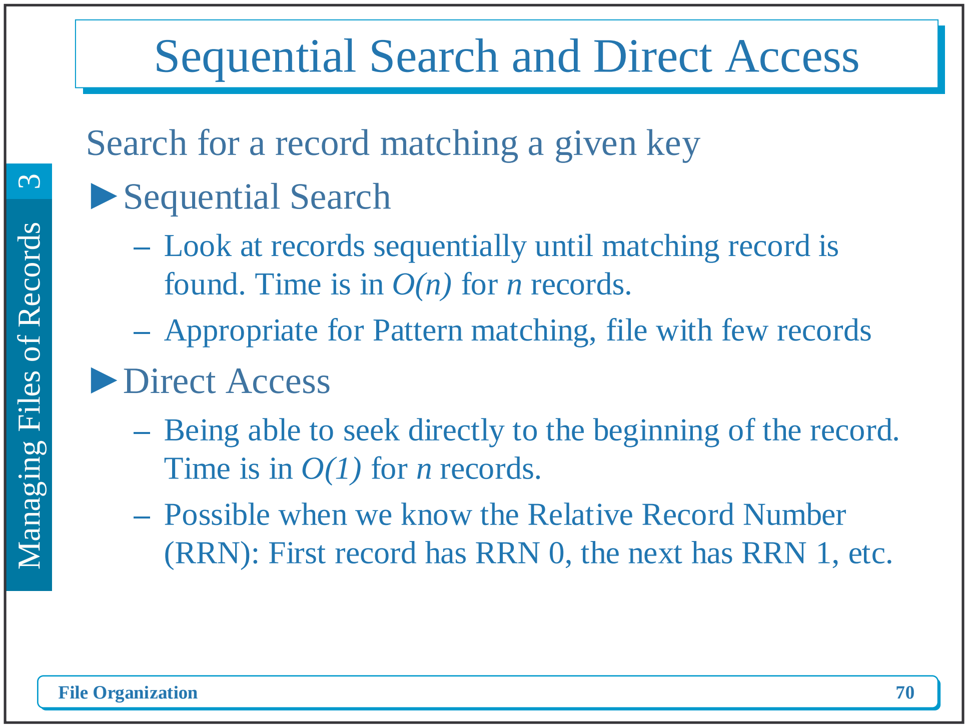
Keyword=Value

ISBN=87358**|AU=**CARROLL**|TI=**ALICE IN WONDERLAND**|**
ISBN=03818**|AU=**FOLK**|TI=**FILE STRUCTURES**|**
ISBN=86683**|AU=**KNUTH**|TI=**SURREAL NUMBERS**|**
ISBN=18395**|AU=**TOLKIEN**|TI=**THE HOBITT**|**



Field Structures: Advantages & Disadvantages

Type	Advantages	Disadvantages
Fixed	Easy to Read/Store	Waste space with padding
Width length indicator	Easy to jump ahead to the end of the field	Long fields require more than 1 byte to store length (Max is 255)
Delimited Fields	May waste less space than with length-based	Have to check every byte of field against the delimiter
Keyword	Fields are self describing allows for missing fields	Waste space with keywords



Sequential Search and Direct Access

Search for a record matching a given key

- ▶ Sequential Search
  - – Look at records sequentially until matching record is found. Time is in *O(n)* for *n* records.
  - – Appropriate for Pattern matching, file with few records
- ▶ Direct Access
  - – Being able to seek directly to the beginning of the record. Time is in *O(1)* for *n* records.
  - – Possible when we know the Relative Record Number (RRN): First record has RRN 0, the next has RRN 1, etc.

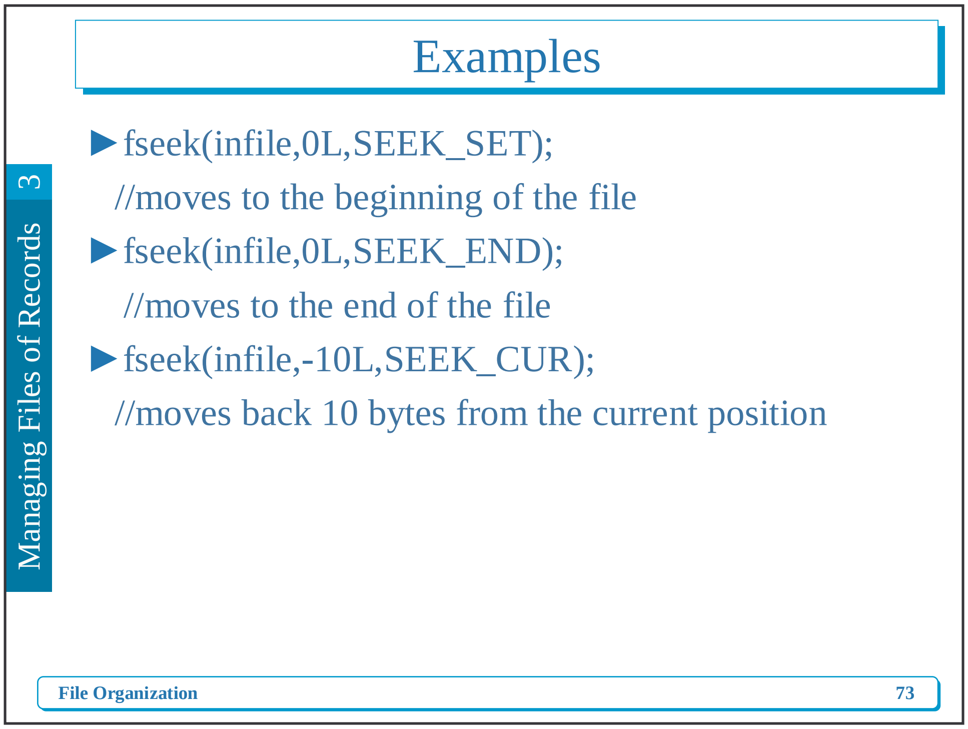
Direct Access by RRN

- ▶ Requires records of fixed length.
  - RRN=30 (31st record)
  - Record length = 101 bytes
  - Byte offset = 30 × 101 = 3030
- ▶ Now, how to go directly to the byte 3030 in the file
  - By seeking

Seeking in C

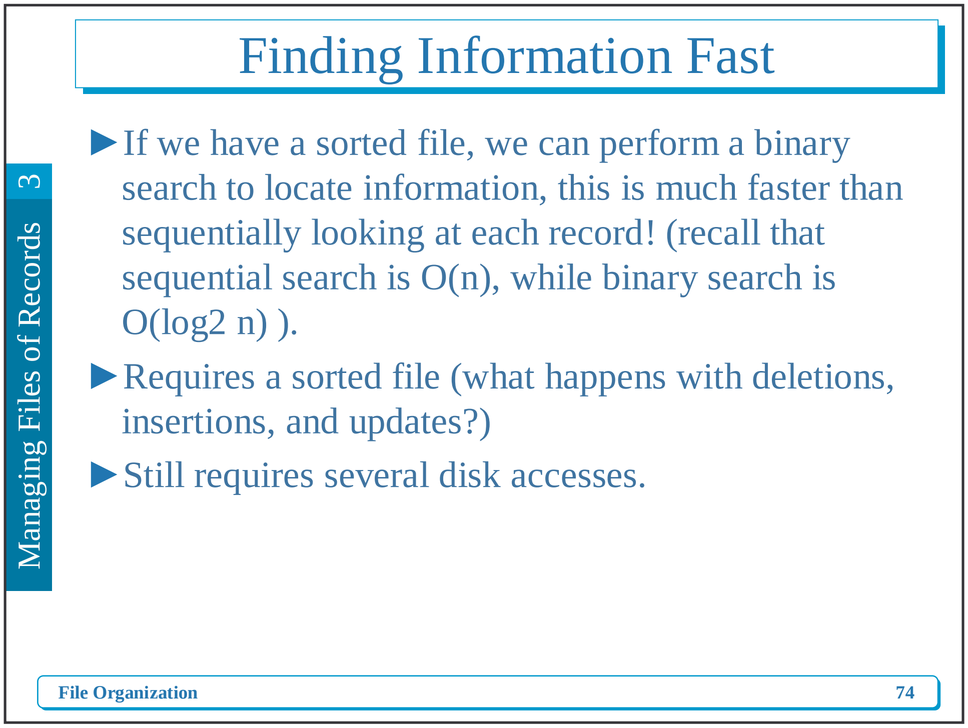
- ▶ int fseek(FILE *stream, long offset, int whence);
- ▶ Repositions a file pointer on a stream.
- ▶ fseek sets the file pointer associated with stream to a new position that is offset bytes from the file location given by whence.
- ▶ Whence must be one of the values 0. 1, or 2 which represent three symbolic constants (defined in stdio.h) as follows:

– SEEK_SET	0	File beginning
– SEEK_CUR	1	Current file pointer position
– SEEK_END	2	End-of-file



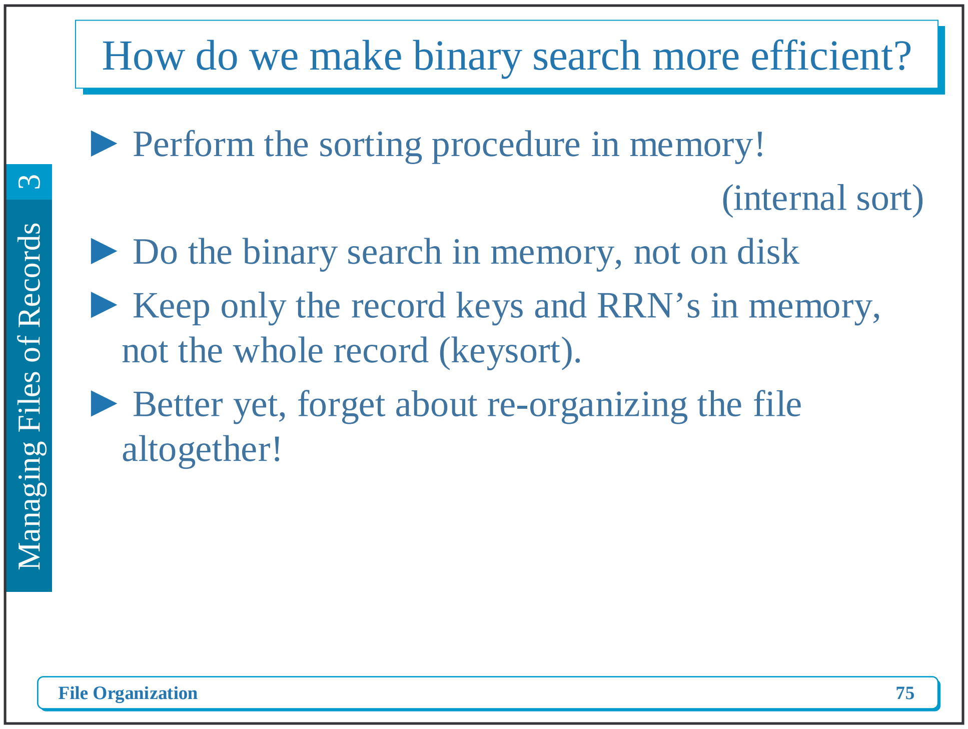
Examples

- fseek(infile,0L,SEEK_SET);
  //moves to the beginning of the file
- fseek(infile,0L,SEEK_END);
  //moves to the end of the file
- fseek(infile,-10L,SEEK_CUR);
  //moves back 10 bytes from the current position



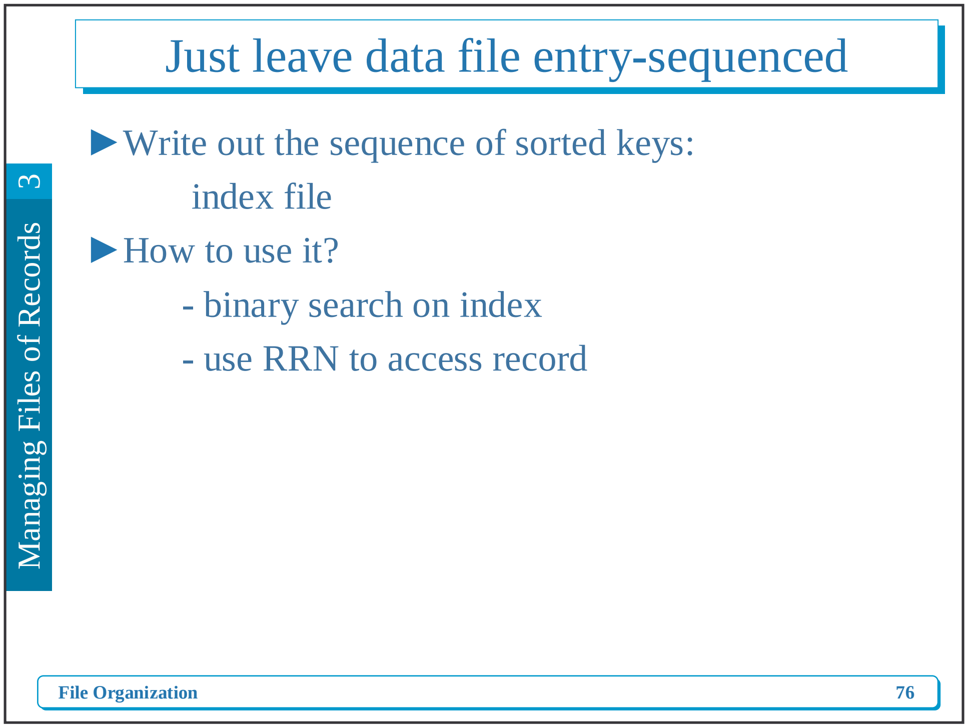
Finding Information Fast

- If we have a sorted file, we can perform a binary search to locate information, this is much faster than sequentially looking at each record! (recall that sequential search is O(n), while binary search is O(log2 n)).
- Requires a sorted file (what happens with deletions, insertions, and updates?)
- Still requires several disk accesses.



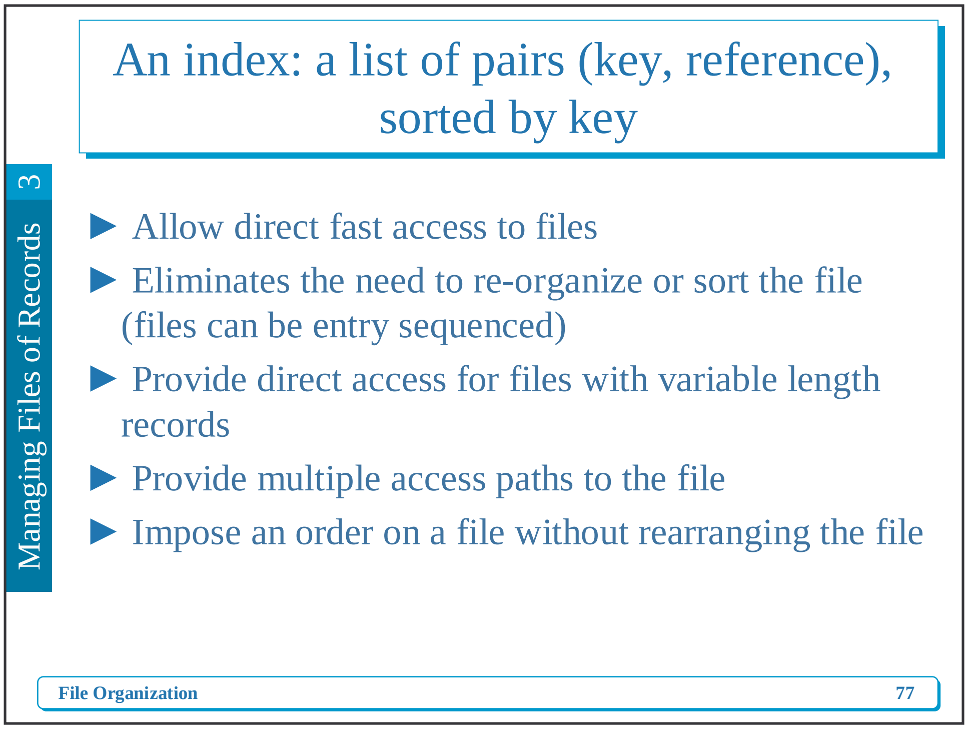
How do we make binary search more efficient?

- Perform the sorting procedure in memory!
  (internal sort)
- Do the binary search in memory, not on disk
- Keep only the record keys and RRN’s in memory, not the whole record (keysort).
- Better yet, forget about re-organizing the file altogether!



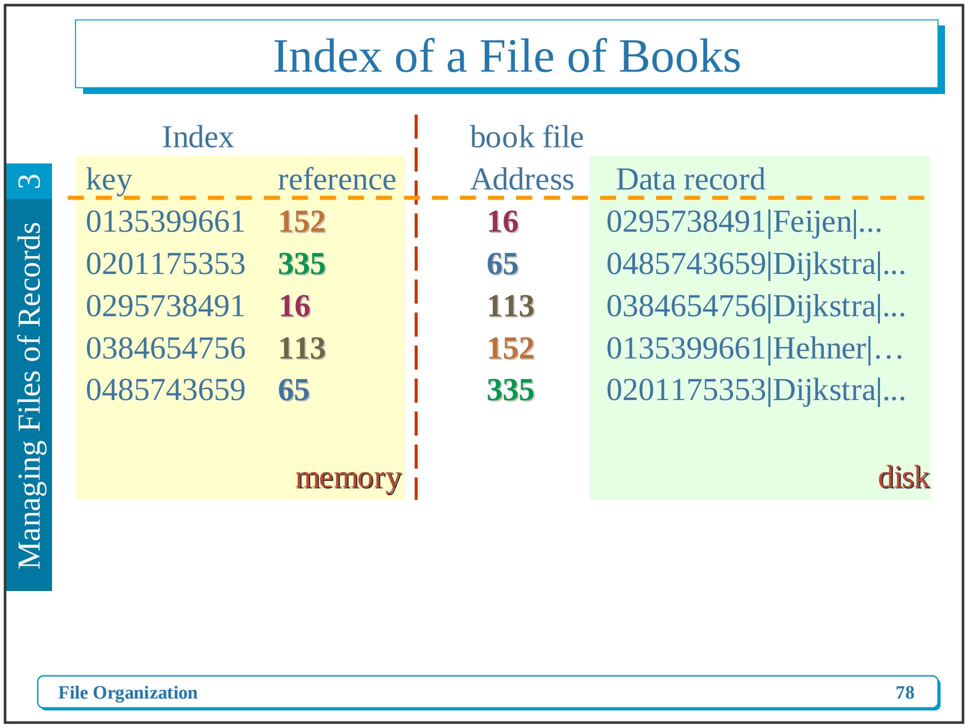
Just leave data file entry-sequenced

- Write out the sequence of sorted keys:
  index file
- How to use it?
  - binary search on index
  - use RRN to access record



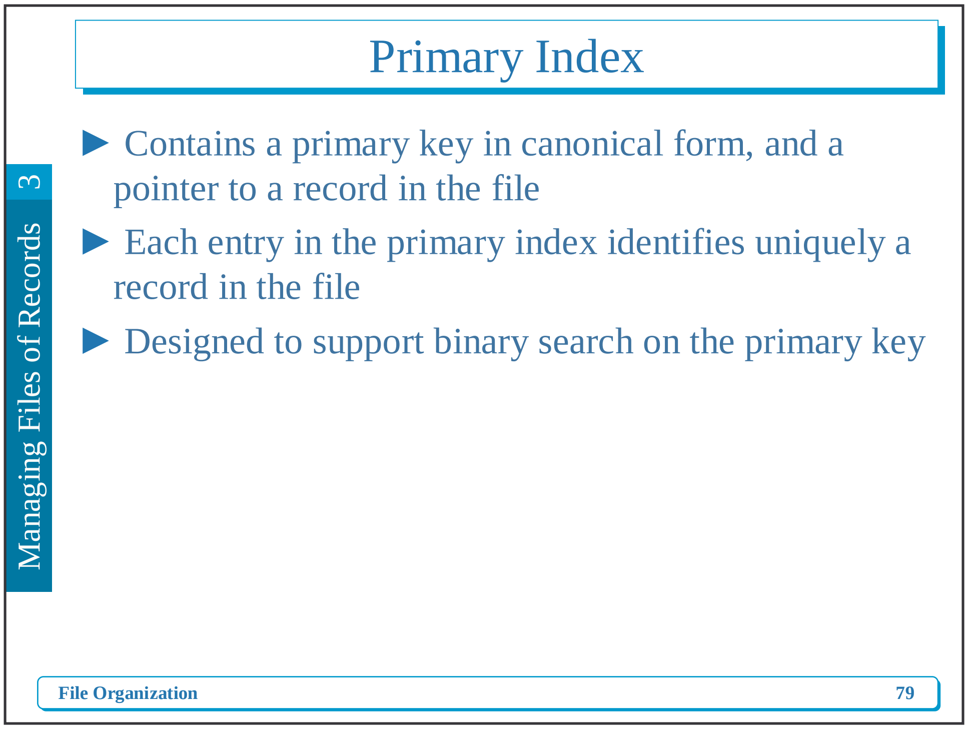
An index: a list of pairs (key, reference), sorted by key

- ▶ Allow direct fast access to files
- ▶ Eliminates the need to re-organize or sort the file (files can be entry sequenced)
- ▶ Provide direct access for files with variable length records
- ▶ Provide multiple access paths to the file
- ▶ Impose an order on a file without rearranging the file



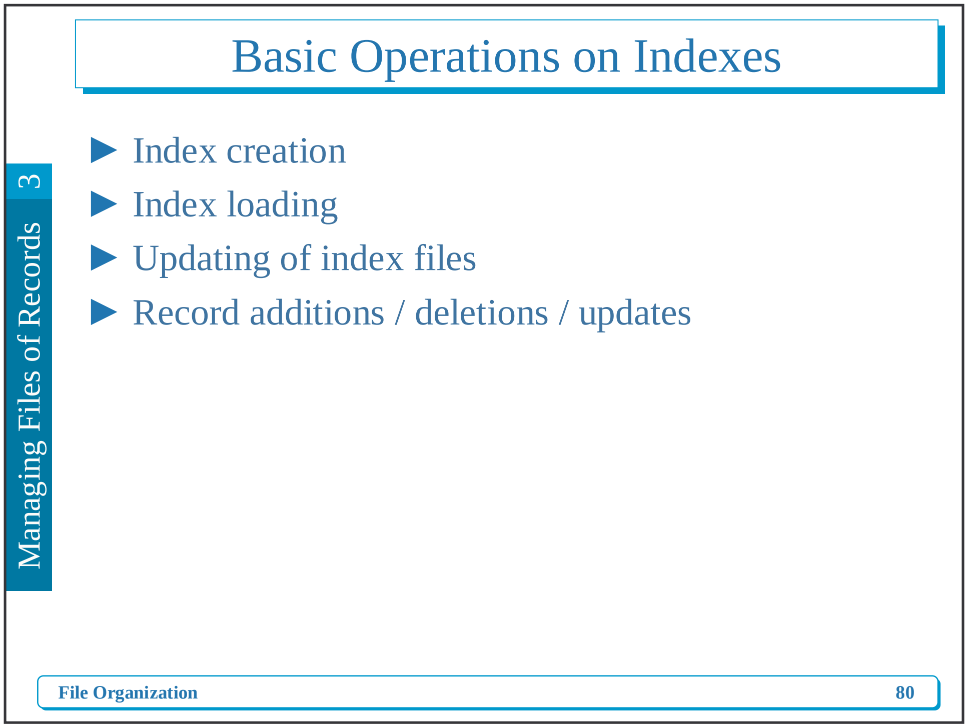
Index of a File of Books

Index		book file	
key	reference	Address	Data record
0135399661	**152**	**16**	0295738491|Feijen|...
0201175353	**335**	**65**	0485743659|Dijkstra|...
0295738491	**16**	**113**	0384654756|Dijkstra|...
0384654756	**113**	**152**	0135399661|Hehner|…
0485743659	**65**	**335**	0201175353|Dijkstra|...
	memory		disk



Primary Index

- Contains a primary key in canonical form, and a pointer to a record in the file
- Each entry in the primary index identifies uniquely a record in the file
- Designed to support binary search on the primary key



Basic Operations on Indexes

- Index creation
- Index loading
- Updating of index files
- Record additions / deletions / updates

Use of Multiple Indexes

Managing Files of Records 3

- ▶ Provides multiple views of a data file
- ▶ Allows us to search for particular values within fields that are not primary keys
- ▶ Allows us to search using combinations of secondary / primary keys
- ▶ Each entry in a secondary index contains a key value and a primary key (or list of primary keys).

Secondary Key

Managing Files of Records 3

- ▶ Does not identify records uniquely
- ▶ It is not dataless
- ▶ Has a canonical form (i.e.there are restrictions on the values that the key must take)

Secondary Index Structure

- List of secondary keys, sorted first by value of the secondary key, and then by the value of the primary key
- Updates to the file must now be applied on the secondary indexes as well.
- The fact that we store primary keys instead of pointers into the file minimizes the impact of file updates on the secondary index.

Author Index

Secondary key	Set of primary keys
Dijkstra	0201175353 0384654756 0485743659
Feijen	0295738491
Hehner	0135399661

Deletion of a Record

- ▶ Change only data file and primary index
- ▶ Search secondary key, find primary key,
    search for p.k. in primary index
    ---> record-not-found
- ▶ saved from reading wrong data

Update a Record

- ▶ Change secondary key:

X rearrange secondary index

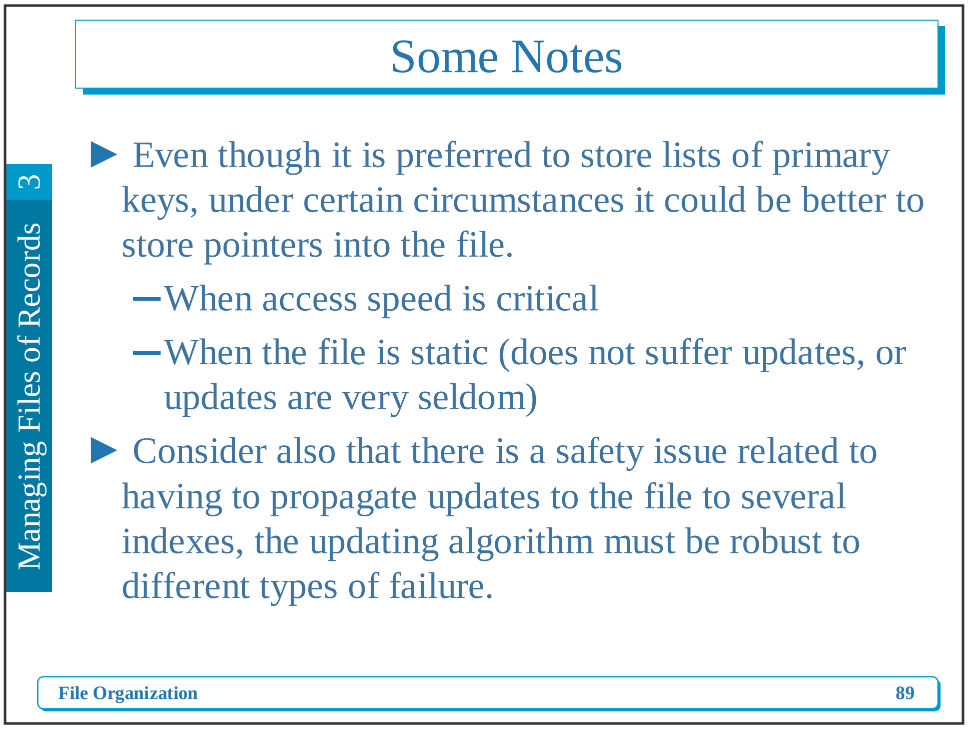
- ▶ Change primary key:
    rearrange primary index
    rewrite reference fields of secondary
    index (no rearrangement)
- ▶ Change other fields: no effect on secondary index

Improving Secondary Indexes

- We can store several primary keys per row in the secondary index
  - This, however, wastes space for some records, and is not sufficient for other secondary keys.
- We can store a pointer to a linked list of primary keys
  - We want these lists to be stored in a file, and to be easy to manage; hence, the inverted list

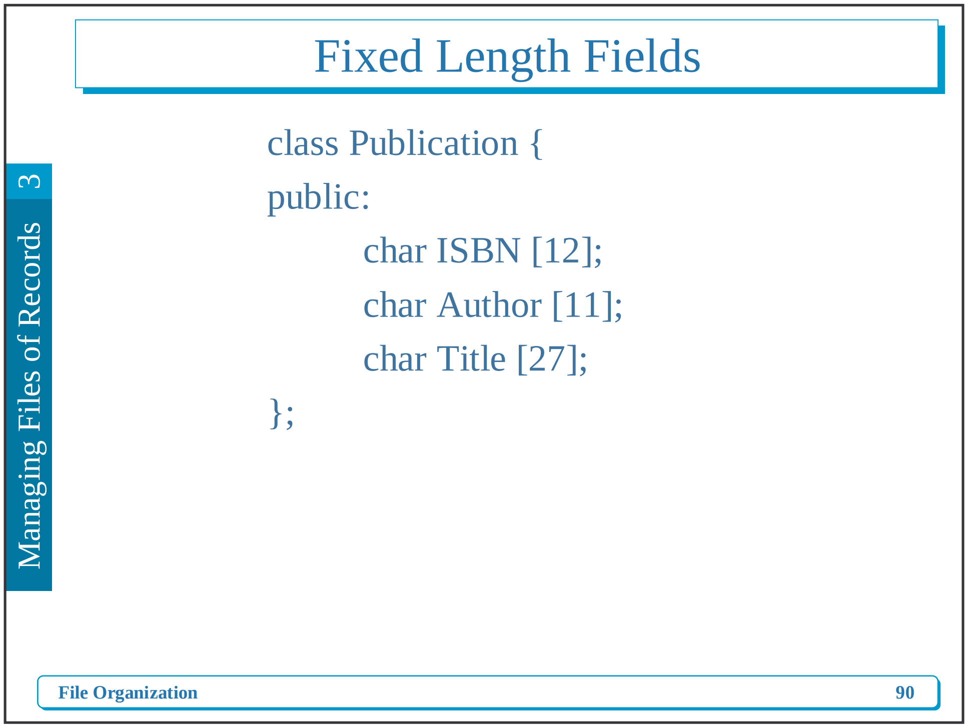
Inverted Lists

- Solve the problems associated with the variability in the number of references a secondary key can have
- Greatly reduces the need to reorganize / sort the secondary index
- Store primary keys in the order they are entered, do not need to be sorted
- The downside is that references for one secondary key are spread across the inverted list



Some Notes

- Even though it is preferred to store lists of primary keys, under certain circumstances it could be better to store pointers into the file.
  - When access speed is critical
  - When the file is static (does not suffer updates, or updates are very seldom)
- Consider also that there is a safety issue related to having to propagate updates to the file to several indexes, the updating algorithm must be robust to different types of failure.



Fixed Length Fields

```
class Publication {
public:
        char ISBN [12];
        char Author [11];
        char Title [27];
};
```