



23 Files & File Management

Motivation, Introduction, Files,
File Access Organizations
February 2 2009
Winter Term 2008/09
Gerd Liefländer



Recommended Reading

- Bacon, J.: Operating Systems (6)
- Silberschatz, A.: Operating System Concepts (10,11)
- Stallings, W.: Operating Systems (12)
- Tanenbaum, A.: Modern Operating Systems (6)
- Nehmer, J.: Systemsoftware (9)
- Solomon, D.A.: Inside Windows NT, 1998
- ... Distributed File-Systems related Papers



Roadmap of Today*

- Motivation, Introduction
 - File Management
 - File Access Implementation
 - Sequential
 - Index Sequential
 - Direct
 - Preview: Objectives
 - To explain the function of file systems
 - To describe the interfaces to file systems
 - To discuss file-system design tradeoffs, including access methods, file sharing, file locking, and directory structures
 - To explore file-system protection
- } Not in Tanenbaum's
Textbook



Motivation/Introduction

File ~ contiguous logical address space

File types:

- data
 - numeric
 - character
 - binary
- program



Tanenbaum's Motivation

Requirements motivating files & file systems

1. Enable the storing of large amount of data
2. Store data consistently & persistently
3. Look up "easily" previously stored data
4. But, some people claim that traditional FS are out and will be replaced sooner or later



Motivation

Do we still need files and a classical file system¹?

→ Save (huge) information sets for future use

Files are the third major OS-provided abstraction over HW resources

OS Abstraction	HW Resource
Processes, Threads	CPU
Address Space	Main Memory (RAM)
Files	Disk ² , CD, ...

¹see http://www.osnews.com/story.php?news_id=9228

²Our main example



Jakob Nielsen's Arguments

- Current FS are based on 3 assumptions:
 1. Information is partitioned into coherent and disjoint units, each of which is treated as a separate object (file). Users typically manipulate information using a file and are often restricted to be "in" one file at a time.
 2. Information objects are classified according to a single hierarchy: the subdirectory structure.
 3. Each information object is given a single, semi-unique name, which is fixed. This file name is the main way users access information inside the file object.
- This characteristics no longer holds in www-times



Single Unit Argument

- Web page consists of a text file + some image "objects" either
 - jpg file
 - gif file
- jpg and gif files can exist in the file system or are generated dynamically on demand from an underlying image representation with parameters as
 - Compression
 - Lossyness
 - Colordepth
 - ...
- Instead of one single executable file we might want to have the executable enhanced by adequate environments depending whether it should run on our
 - Office PC
 - PAD
 - Mobile



Single Hierarchy Argument

- Suppose you do regularly multiple presentations using
 - Screen shots
 - Measurement graphs
 - ...
- If you have to change or adapt a screen shot, you do not want to change or adapt this screen shot in each of your presentation file
- Find appropriate arguments against the current use of traditional **file names**
- Geoff Barell describes spotlight: Apple's Index in OS
<http://www.computerworld.com/hardwaretopics/hardware/desktops/story/0,10801,103518,00.html>



File

Collection of related information recorded on some form of media

Files can consist of very different kind of recorded information, e.g.

- Executable program
- Text files (e.g. Fender's catalogue of E-guitars)
- ...

A file has a set of **attributes**, i.e. its **meta data**

Attributes differ between OSes and FSs, e.g.:

- **Name, identifier**
- **Type** (needed for OS that support different types, e.g. .exe)
- **Location** (physical address of file on device)
- **Size** (# bytes or #blocks)
- **Protection** (who can access and how)
- others



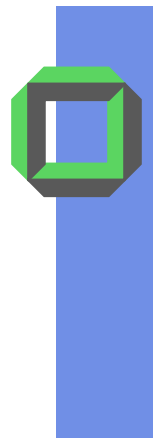
Typical File Attributes

Attribute	Meaning
Protection	Who can access the file and in what way
Password	Password needed to access the file
Creator	ID of the person who created the file
Owner	Current owner
Read-only flag	0 for read/write; 1 for read only
Hidden flag	0 for normal; 1 for do not display in listings
System flag	0 for normal files; 1 for system file
Archive flag	0 for has been backed up; 1 for needs to be backed up
ASCII/binary flag	0 for ASCII file; 1 for binary file
Random access flag	0 for sequential access only; 1 for random access
Temporary flag	0 for normal; 1 for delete file on process exit
Lock flags	0 for unlocked; nonzero for locked
Record length	Number of bytes in a record
Key position	Offset of the key within each record
Key length	Number of bytes in the key field
Creation time	Date and time the file was created
Time of last access	Date and time the file was last accessed
Time of last change	Date and time the file has last changed
Current size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to

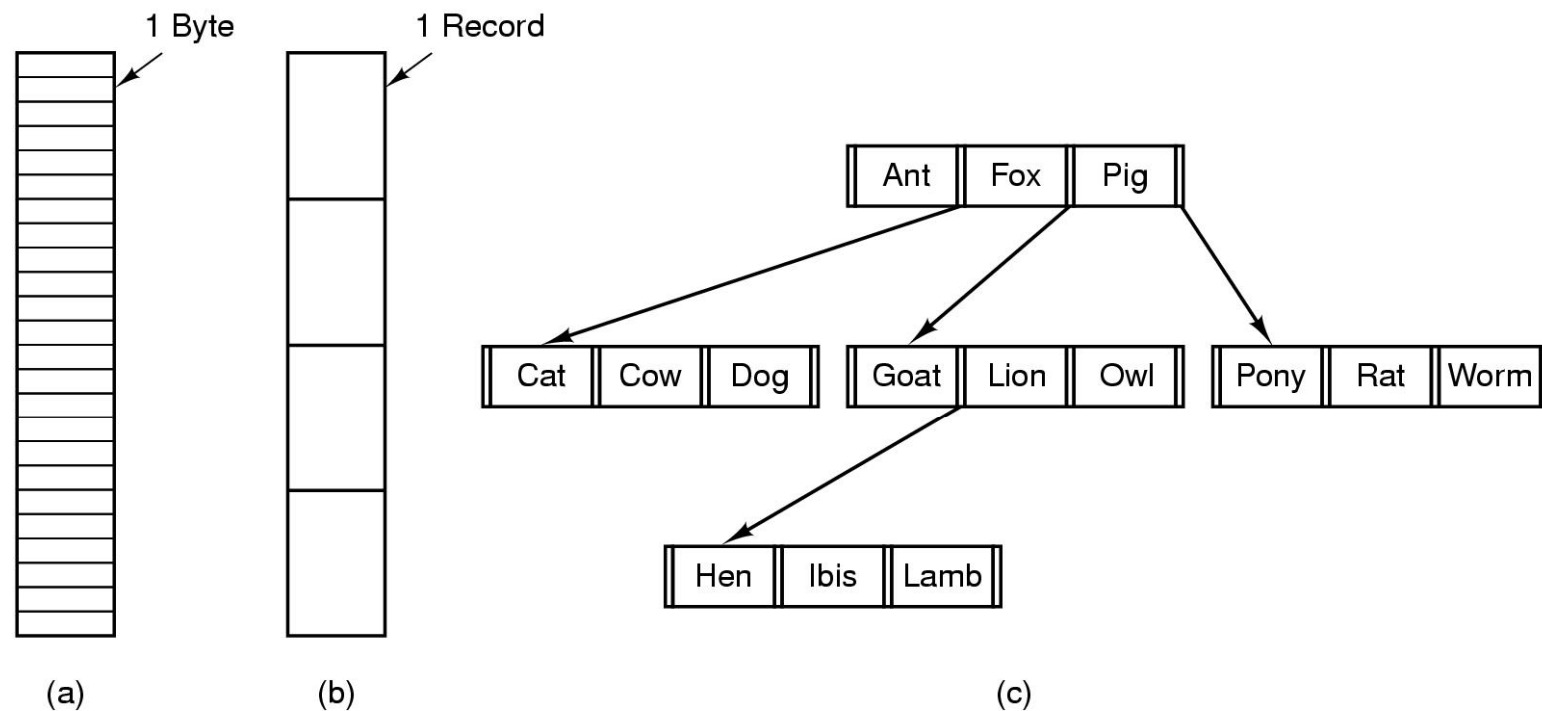


File Structure

- None - sequence of words, bytes
- Simple record structure
 - Lines
 - Fixed length
 - Variable length
- Complex Structures
 - Formatted document
 - Relocatable load file



File Structure (OS's Point of View)



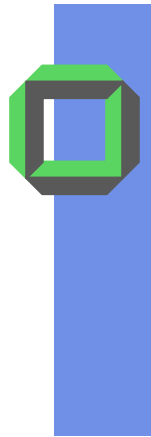
Three kinds of files:

- (a) byte sequence (provides maximal flexibility)
- (b) record sequence (often with fixed sized records)
- (c) Tree (sometimes with variable sized records)



File Types

- A file can have a type (e.g. `.exe`)
 - understood by the FS
 - block, character, device, portal, link, ...
 - understood by all parts of OS or runtime libraries
 - executable, dll, source, object, text, ...
- A file's type can be encoded in
 - its name, e.g. MS
 - `.com`, `.exe`, `.bat`, `.dll`, `.jpg` ...
 - its content, e.g. Unix/Linux file use
 - its inode, magic number or
 - an initial character (e.g. `#!` for shell scripts)



Example File Types

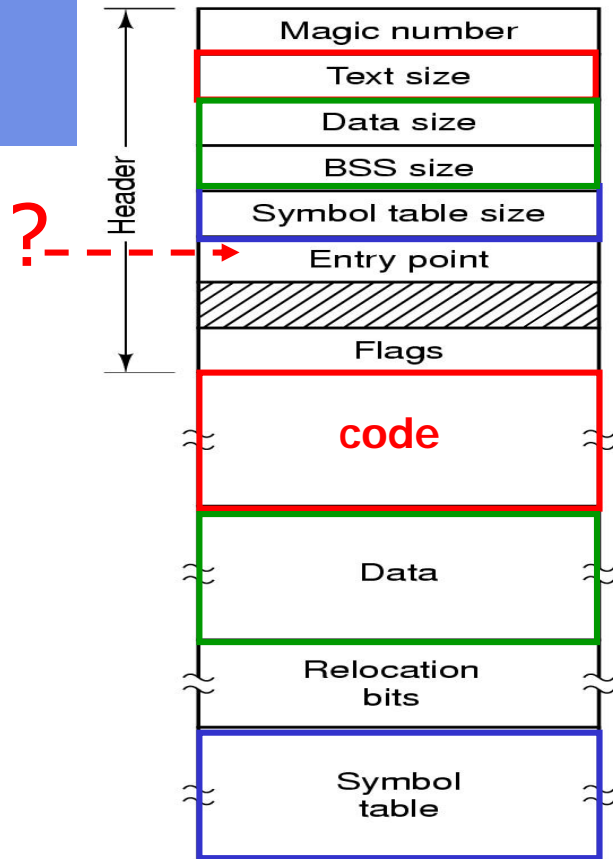
file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine- language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rtf, doc	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	ps, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes com- pressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information



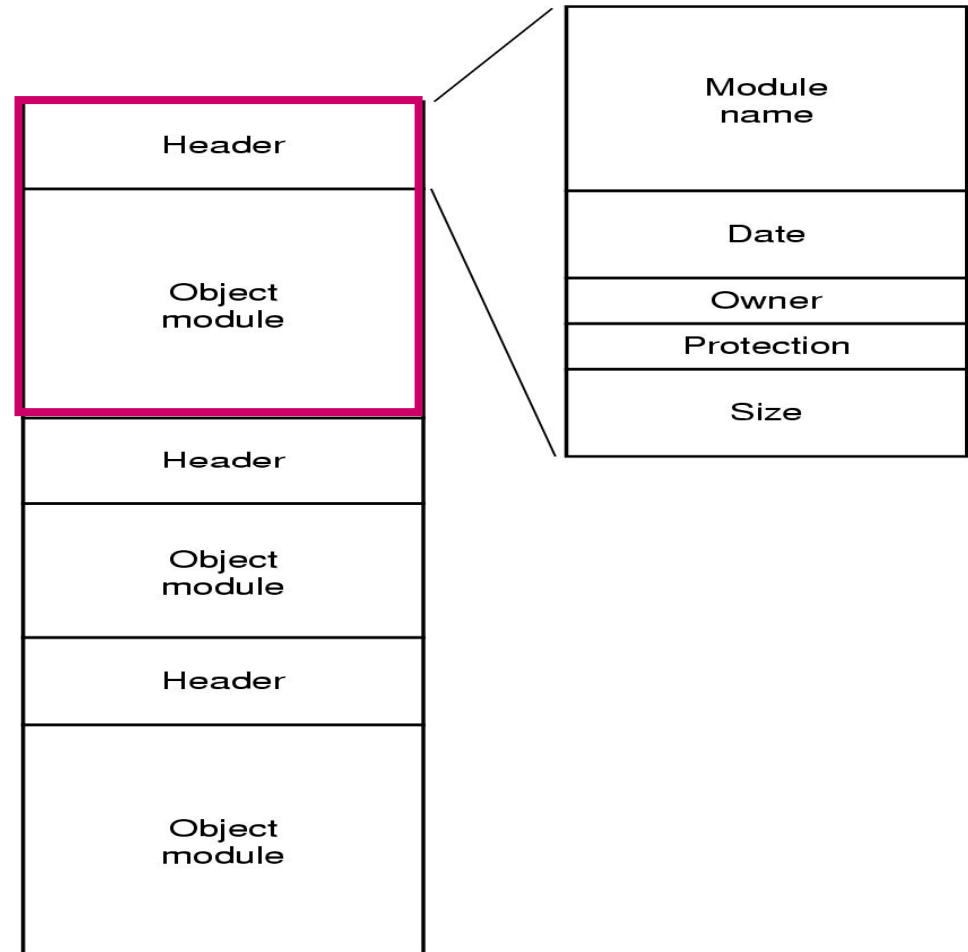
Unix File Types (1)

- Regular files
 - ASCII (→ unicode)
 - Binaries
 - Executable files
 - Archive
- Directories
- Device files
 - Character device
 - Block device

File Types (2)



(a) Executable file (e.g. ELF)



(b) Archive file



Typical File Applications

- Private data bases
 - Music-CDs, MP3
 - Movies, Photo Gallery, ...
- Commercial software in specific formats (e.g. .zip)
- OS itself as an executable file loaded at boot time
- Typical file contents:
 - Numeric data
 - Audio-/Visual information
 - Books, catalogues, lyrics & tabs, ...
 - Photos, drawings, ...
 - Binary programs

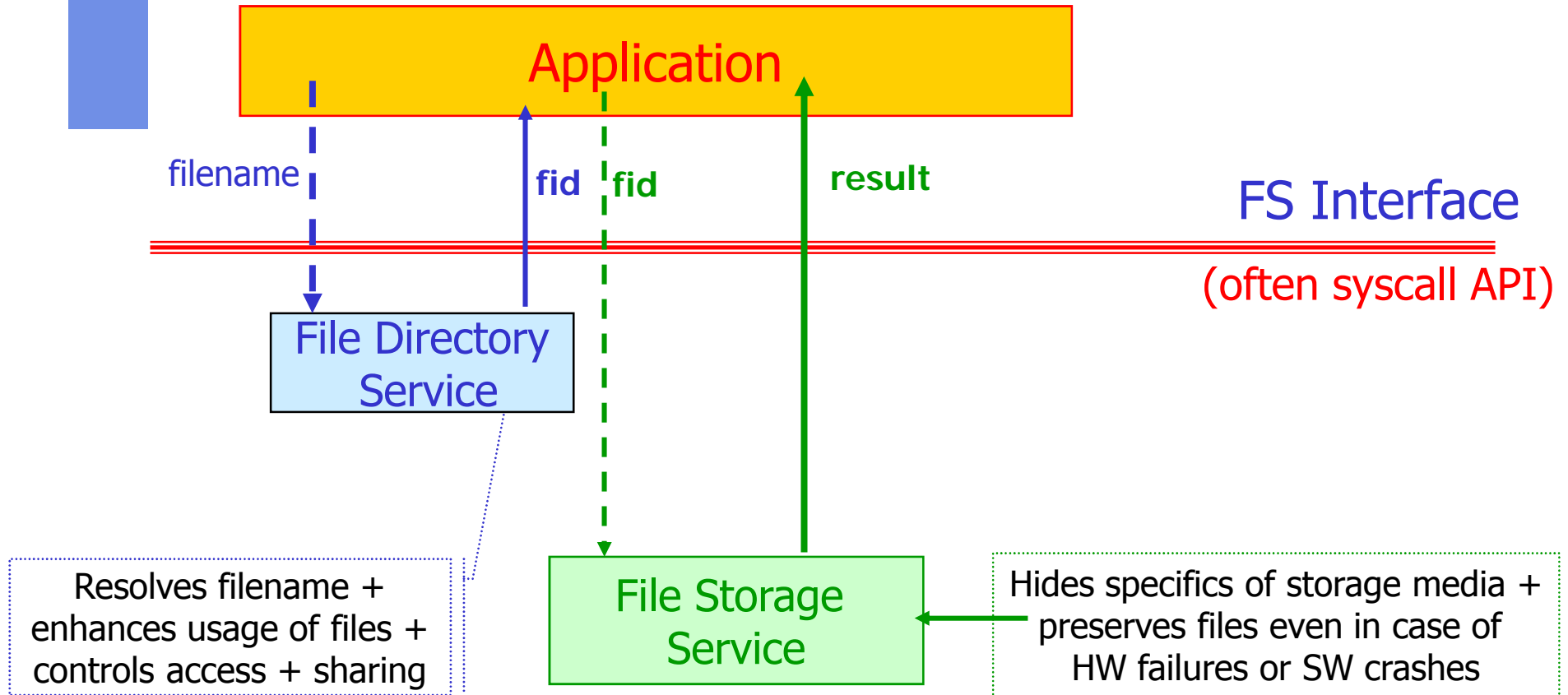


Abstract File Operations

A file is an abstract data type/object offering

- `create()`
- `write()`
- `read()`
- `reposition()` (within file)
- `delete()`
- `truncate()`
- `open(Fi)` – search the directory structure on disk for entry F_i, and move its content to memory
- `close(Fi)` – move content of entry F_i in memory to directory structure on disk

Interaction with a Classical FS





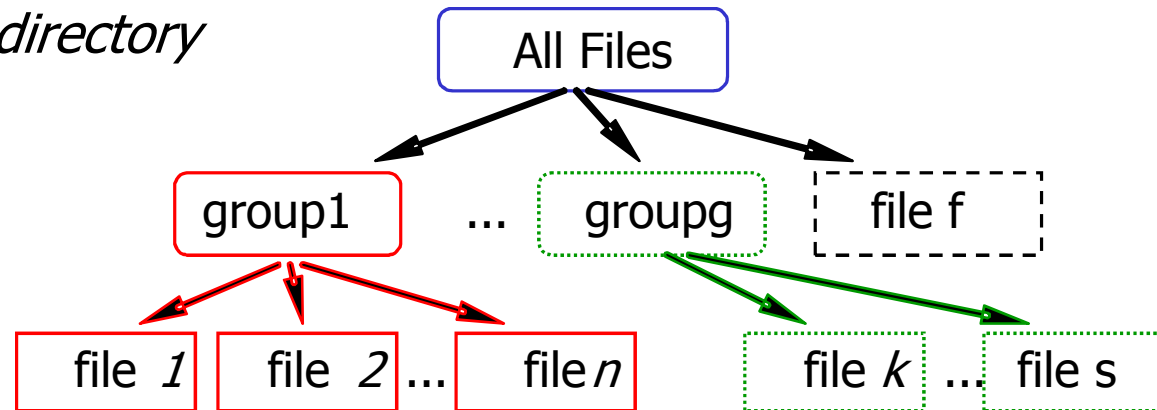
File Systems

- Most files are still located on disks which are really messy physical devices:
 - Errors, bad blocks, missed seeks, ...
- Job of an OS is to hide this mess from higher level software
 - Low-level device control (initiate a disk read, etc.)
 - High-level abstractions (read file)
- OS might provide different levels of disk access to different clients (applications)
 - Physical disk (surface, cylinder, sector)
 - Logical disk (disk block#)
 - Logical file (file block, record, byte#)

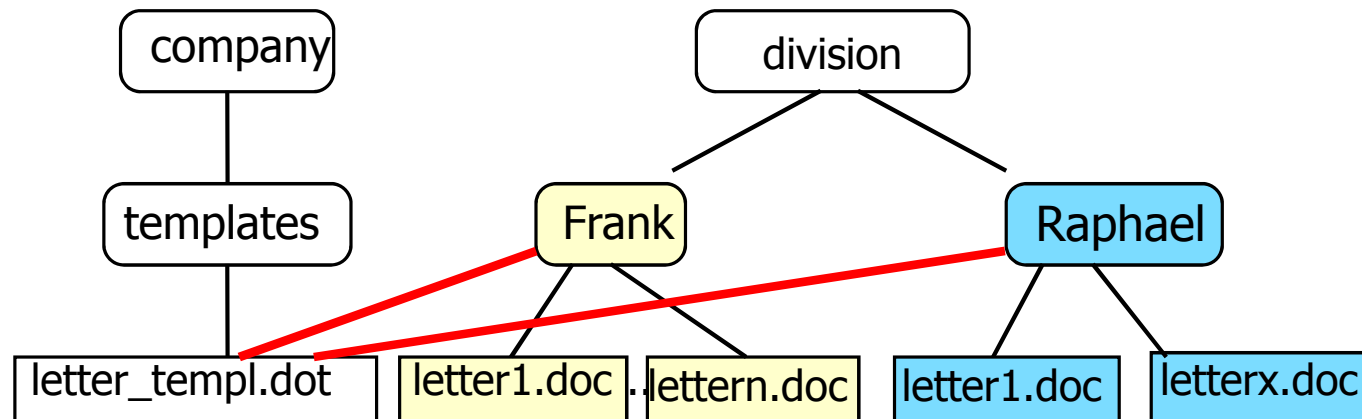


Overview File System

- Hierarchical FS as a rooted tree
 internal node = *directory*



- Additional links (*acyclic graph*)



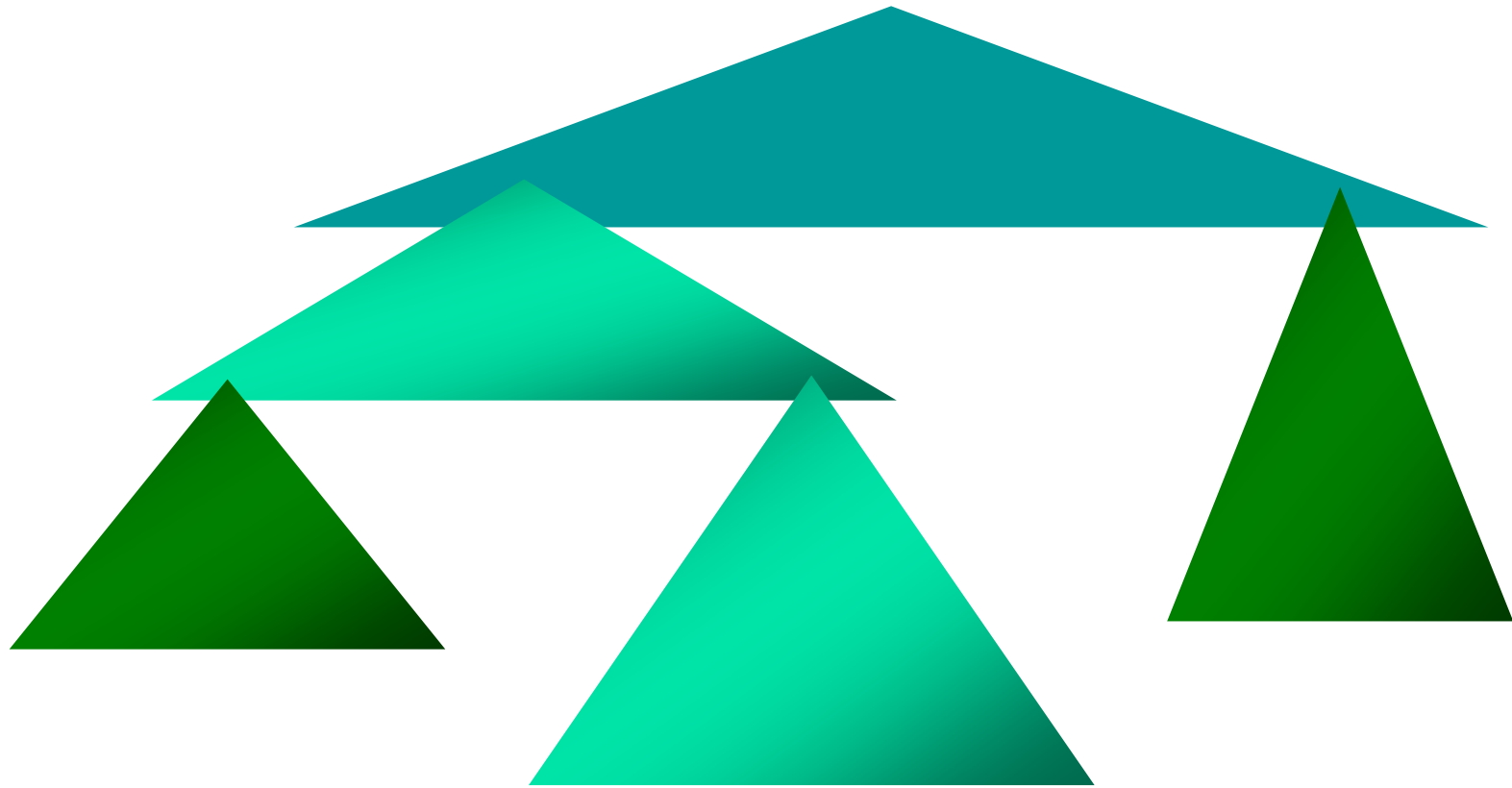


File System Basics

- OS may support multiple file systems
 - instances of the same FS type
 - Different FS types, e.g. EXT2 & Reiser
- All file systems are typically bound into a single **namespace**
 - Often hierarchical



Hierarchy of File Systems



- *Why hierarchical?*
- *Alternative ways of organizing a name space*
- *Why not a single file system?*



File Management

Names
Structure
Types
Attributes



File Management

- A file should only disappear if the **authorized instance** (subject) explicitly deletes it
- An authorized subject can be the
 - owner
 - system administrator
 - system security manager
- Design space of file organization depends on file
 - naming
 - structuring
 - accessing
 - protecting
 - implementing



Goals of File Management

- Provide a convenient naming scheme for files
- Provide uniform I/O support for a variety of storage device types
- Provide standardized set of I/O interface functions
- Minimize/eliminate lost or corrupted data
- Provide I/O support and access control for multiple users
- Enhance system administration (e.g. backup)
- Provide acceptable **performance**



File Names

- FS with a convenient **naming scheme**, e.g.
 - Textual names
 - Restricted alphabet, i.e.
 - Only certain characters (e.g. no '?' or '/')
 - Limited length
 - only certain formats, e.g.
 - DOS 8 character string.**xyz** character suffix
 - XP 255 character.**xyz** character suffix
 - Case (in)sensitive
 - Names must fulfill certain **convention**, extension **xyz.c** or **xyz.C** if **C(++)**-Compiler should run



Open Files

- Several meta data are needed to manage open files:
 - **file pointer**: pointer to last read/write location, per process that has the file open
 - **file-open count**: counter of number of times a file is open – to allow removal of data from open-file table when last processes closes it
 - **disk location**: cache of data access information
 - **access rights**: per-process/task access mode information, who is allowed to do what



File Access

- **Strictly sequential access (early systems)**
 - read all bytes/records from the beginning
 - cannot jump around, could only rewind
 - sufficient as long as storage was a tape
- **Random access (current systems)**
 - bytes/records read in any order
 - essential for database systems, e.g.
 - Airline reservation
 - DVD or Audio CD



Access Methods

- Sequential Access: `read next`
`write next`
`reset`
`no read after last write`
`append`
- Direct Access: `read n`
`write n`
`position to n`
`read next`
`write next`
`rewrite n`

`n = relative block number`



Example: File Operation (1)

Usage of the following program: **\$ copyfile abc xyz**

```
/* File copy program. Error checking and reporting is minimal. */  
  
#include <sys/types.h>           /* include necessary header files */  
#include <fcntl.h>  
#include <stdlib.h>  
#include <unistd.h>  
  
int main(int argc, char *argv[]); /* ANSI prototype */  
  
#define BUF_SIZE 4096           /* use a buffer size of 4096 bytes */  
#define OUTPUT_MODE 0700       /* protection bits for output file */  
  
int main(int argc, char *argv[])  
{  
    int in_fd, out_fd, rd_count, wt_count;  
    char buffer[BUF_SIZE];  
  
    if (argc != 3) exit(1);      /* syntax error if argc is not 3 */
```




Example: File Operation (2)

```
/* Open the input file and create the output file */
```

```
in_fd = open(argv[1], O_RDONLY); /* open the source file */
```

```
if (in_fd < 0) exit(2); /* if it cannot be opened, exit */
```

```
out_fd = creat(argv[2], OUTPUT_MODE); /* create the destination file */
```

```
if (out_fd < 0) exit(3); /* if it cannot be created, exit */
```

```
/* Copy loop */
```

```
while (TRUE) {
```

```
    rd_count = read(in_fd, buffer, BUF_SIZE); /* read a block of data */
```

```
if (rd_count <= 0) break; /* if end of file or error, exit loop */
```

```
    wt_count = write(out_fd, buffer, rd_count); /* write data */
```

```
    if (wt_count <= 0) exit(4); /* wt_count <= 0 is an error */
```

```
}
```

```
/* Close the files */
```

```
close(in_fd);
```

```
close(out_fd);
```

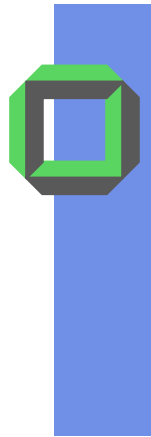
```
if (rd_count == 0) /* no error on last read */
```

```
    exit(0);
```

```
else
```

```
    exit(5); /* error on last read */
```

```
}
```



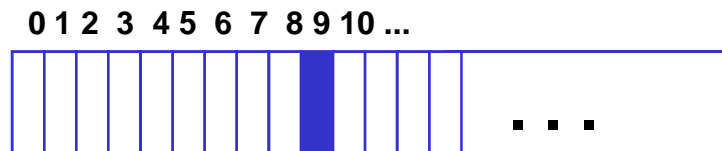
File Access Methods

- Plain (unstructured) file (generic file)
 - Entity: **byte** (sometimes: **block**)
 - If an application wants to structure a persistent data container it has to implement its internal structure
- Structured file
 - Entity: **record** (or user type objects...)

Remark: Since Unix, many OSes only offer plain files, applications and libraries can implement specific structured file types on top of this.

Plain File

Definition: A plain file is a sequence of bytes (gaps are possible). Typically located on a disk.



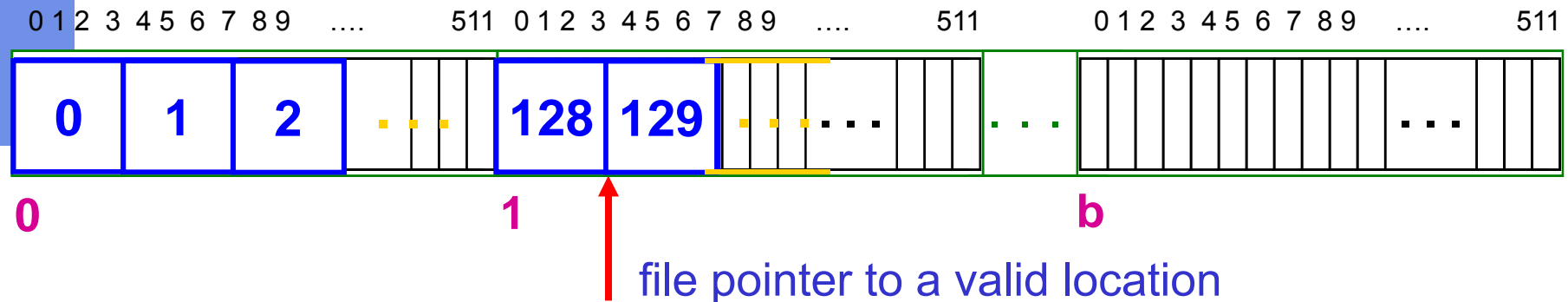
↑ file pointer to the current location within a file

Characteristic: You can randomly access any byte within an unstructured file if you have positioned its file pointer appropriately.

Problem: Disks cannot access bytes; only blocks.

Solution: Buffer file blocks (classical method) or entire files (**memory mapped files**) within main memory (see later paragraph "file buffering") or **virtual memory**.

Structured File



Records = logical entities tightly coupled to a specific application, e.g. record of an employee

Employee-file might contain all relevant information, e.g.

- employee number, family name, *Christian name*,
- employee position, department number,
- passport number, birth date, salary, etc.

Records of equal size or not (then additional length field is needed)

Records with special key field (\Rightarrow some ordering within the file)



File Organization and Access

- Possible access patterns:
 - Read the whole file
 - Read individual blocks of a file
 - Read blocks preceding/following the current one
 - Retrieve a subset of records
 - Write/update a complete file sequentially
 - Insert/delete/update one record in a file
 - Update blocks in a file



Operations on Unstructured Files

CreateFile(pathname)

DestroyFile(pathname)

OpenFile(pathname, read/write)

ReadFile(FID, byte-range, where to put
bytes into main memory)

WriteFile(FID, byte-range to write, where to find
bytes in main memory)

CloseFile(FID)

PositionPointer(FID, posi. for pointer in byt.)

Remark: "bytes in main memory" is data space within AS of
the calling client (e.g. global data segment or stack).



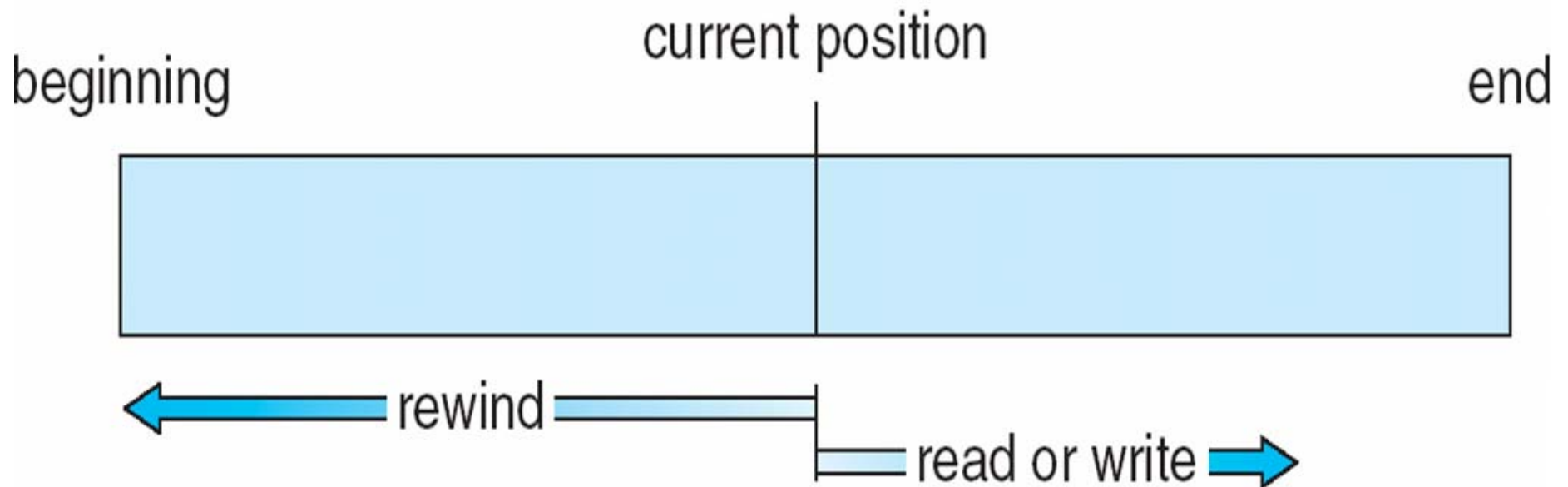
File Organization and Access

Programmer's perspective:

- Given an OS supporting only unstructured files, i.e. data containers with a **sequence of bytes** (see Linux, Unix, MSxyz ...)
- Programmer has full flexibility to impose any desired structure on a plain file, but it has also the burden to do so
- *How to organize the contents of these files?*



A Sequential Access to a File





Simulation of a Sequential-Access

- Given a direct (or random access) file
- *How to implement sequential access?*

sequential access	implementation for direct access
<i>reset</i>	<i>cp = 0;</i>
<i>read next</i>	<i>read cp;</i> <i>cp = cp + 1;</i>
<i>write next</i>	<i>write cp;</i> <i>cp = cp + 1;</i>



Structured Files

Performance Criteria

Sequential

Index- & Tree-Sequential

Extensible Hashed



Terms in Structured Files

- **Field**
 - basic element of data
 - contains one single value of a specific data type
 - characterized by its length and data type
- **Record**
 - collection of related fields
 - treated as a unit (e.g. employee record)
- **File**
 - collection of similar records
 - treated as a single entity
 - with restricted access
- **Database**
 - collection of related data (files)
 - relationships exist among elements



Operations on Structured Files

- **Retrieve_All, Retrieve_Few, Retrieve_One**
(e.g. sequential processing, all records satisfying some criteria)
- **Retrieve_Next, Retrieve_Previous**
(e.g. search within a file for a specific attribute)
- **Insert_One, Delete_One**
(e.g. hire a new employee or fire an old one)
- **Update_One, Update_All**
(e.g. decrease the salary or increase time of work of an employee)



Criteria for File Implementation (1)

- **Rapid access**
 - Needed when accessing a single record
 - Not needed for batch mode

- **Ease of update**
 - Some files may be updated periodically
 - File on a CD-ROM will never be updated

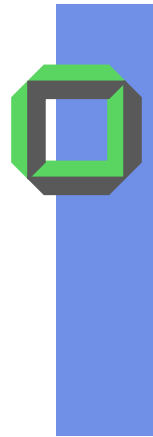


Criteria for File Implementation (2)

- **Economy of storage**
 - Should be minimal redundancy in the data
 - Redundancy can be used to speed up access (e.g. via index or keys)

- **Simple maintenance**

- **Reliability**
 - Redundancy and checkpoints can be used to improve robustness



Fundamental Access Methods

- \exists many ways to access the content of a structured file, e.g.:
 - sequential } Specialist
 - indexed sequential } Generalist
 - Allows sequential & direct access
 - direct or hashed } Specialist



Sequential Files



Sequential File

Sequential file (**fixed/variable** record length)

- basic operations: read/write next record/ append
- with optional key field(s):
 - one field is key field
 - uniquely identifies the record
 - records are stored in key sequence
 - new records are placed in a “log or transaction file”
 - when closing the file (or periodically), a batch update is done to merge the log file with the master file



Sequential File

Fixed-length Records

r1				
r2				
r3				
r4				
r5				
r6				

Remark:

File with 6 records of 4 fields, 1. field may serve as a key), e.g.
personnel number, name, department, salary



Analysis of Sequential Files

- Update (all records)
 - Same sized record, very good
 - Variable sized record, ok

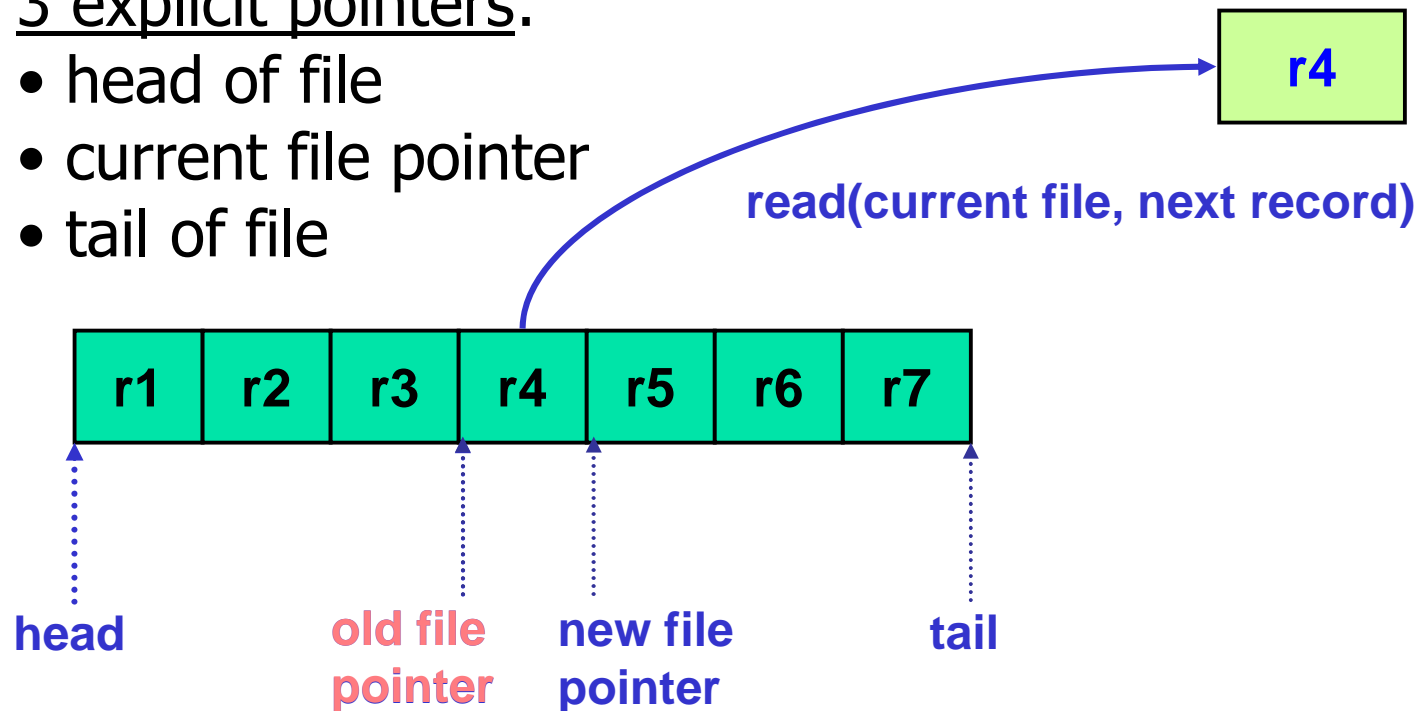
- Retrieval
 - Single record, poor
 - Subset, poor
 - Exhaustive, ok



Implementing Sequential Files (1)

3 explicit pointers:

- head of file
- current file pointer
- tail of file



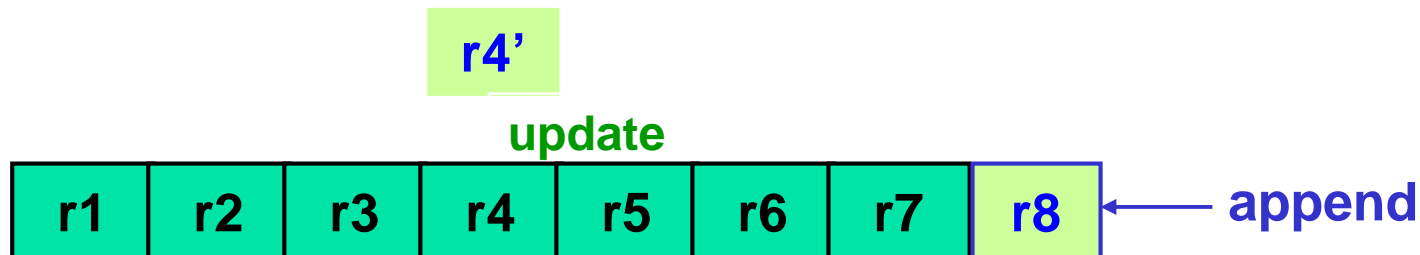
Remark:

Each file operation is related to the current position of the file pointer



Implementing Sequential Files (2)

In some cases you can only append to a sequential file. Updates in the midst of a sequential file are only possible if the updated record has the same size



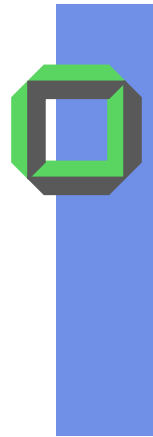
In general you have operations to manipulate the file pointer:

- next \cong place file pointer at the next record
- previous \cong place file pointer at the previous record
- reset \cong place file pointer at the head of the file
- arbitrary \cong any place within the "range" of a file



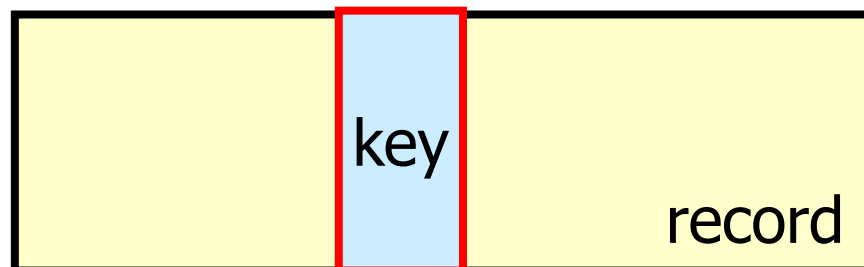
Index- Tree-Sequential

Classical Index-Sequential Files
Modern Tree-Index-Sequential Files

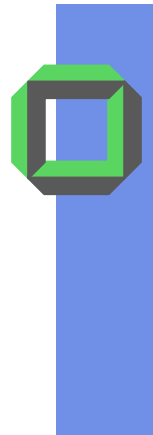


Index(ed) Sequential File

- Index sequential files support two different access methods for structured files
 - Lookup of a specific record \Rightarrow we need additional meta data
 - Sequential access to all records (sorted by key)



Offset (can be zero)



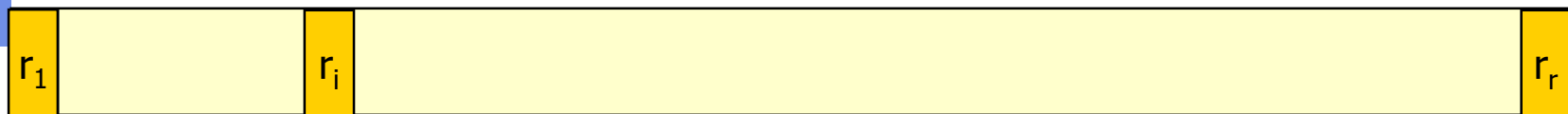
Index Sequential File

- Index key provides a quite fast lookup capability to find vicinity of desired record
 - contains a key field and a pointer to main file
 - index is searched to find highest key value that is equal or less than the desired key value
 - search continues in the main file at the location indicated by the pointer
- Additionally, records are stored sequentially

logically

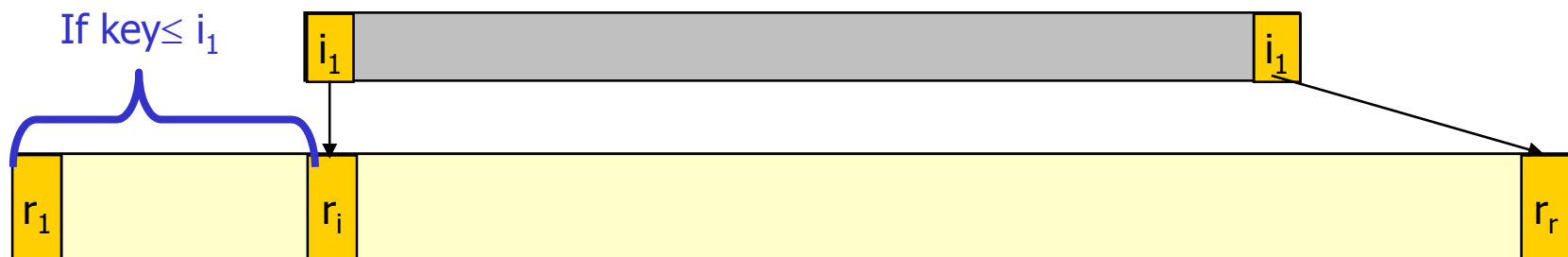
Sequential \leftrightarrow Ind. Sequential File

Sequential file contains $r = 1\,000\,000$ records



Average lookup costs $\sim 500\,000$ comparisons

Indexed sequential file with 1000 indices à 1000 records

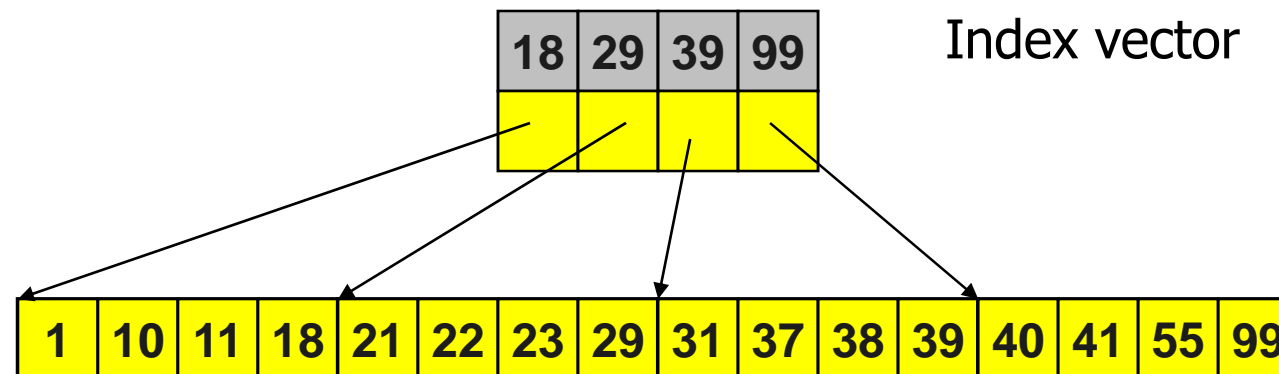


Average lookup costs $\sim 500 + 500 = 1000$ comparisons



Classical Ind. Sequential File

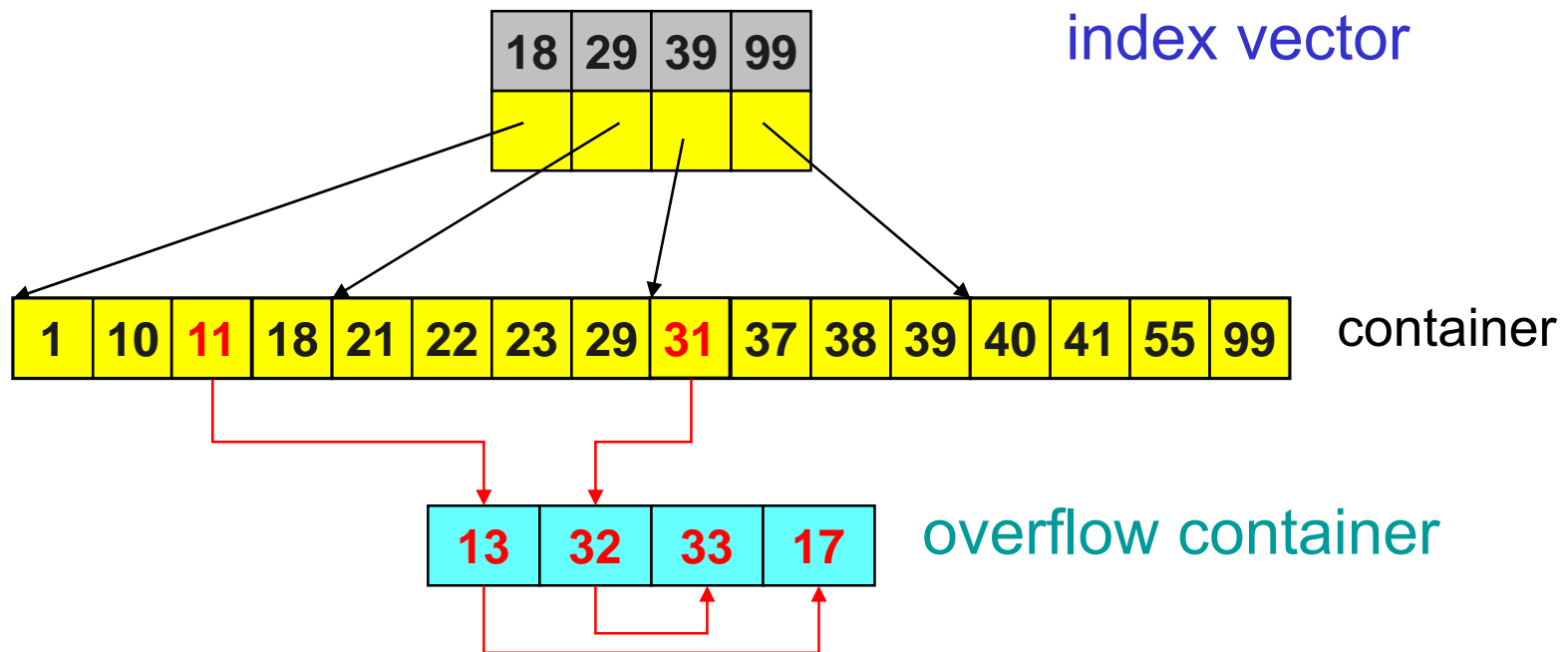
- If file full, a new record is added to overflow container
- Record that precedes the new record in file is updated to contain a pointer to the new record
- Overflow container is merged with main file during a batch update





Classical Ind. Sequential File (2)

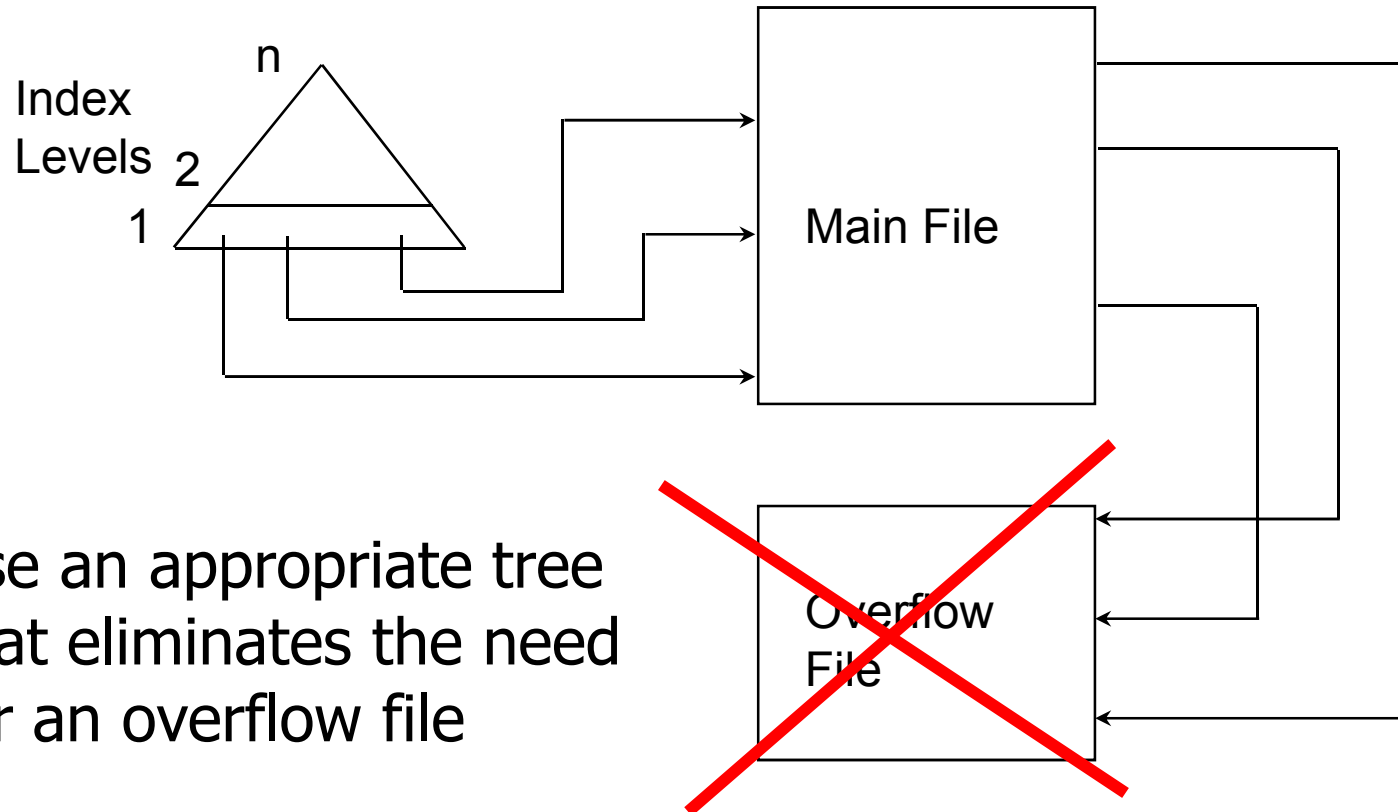
Adding data dynamically requires an overflow container.
Example: You want to add data with keys 13, 32, 33, 17.



Remark: The longer the overflow container \Rightarrow the worse the performance.
Merging file & overflow container may be costly if file is large.



Tree Ind. Sequential File



Use an appropriate tree that eliminates the need for an overflow file

Remark: With B*-trees (R. Bayer 1970) we establish more efficient indexed sequential files. Be *familiar* with B and *B*-trees!*



B*-Indexed Sequential Files

*What does the **B** in B*-tree mean?*

- B → Balanced tree
- B → Broad Trees
- B → Brushy Trees
- B → Bayer tree (due to one of its inventors)

B*-tree = a dynamically height-balanced, t-ary (maximal t successors per inner node), leaf-oriented (real data in the leaves).

The leaves of a B*-tree form a **double-linked list** to facilitate sequential accesses



B Tree, B⁺ Tree, B* Tree

- B Tree contains data in inner nodes
- B⁺ Tree is leaf-oriented
- B* Tree ~ B⁺ Tree + sequential ordering

Specific literature on B, B⁺ and B*-trees:

Knuth, D.: Vol. 3, Searching and Sorting, p. 476 – 481

Wirth, N.: Algorithms + Data Structures = Programs, p. 242 – 264

Bayer, R.: Organizing and Maintenance of Large Ordered Indexes, ACTA Informatica 1:3, p. 173 – 189, 1972

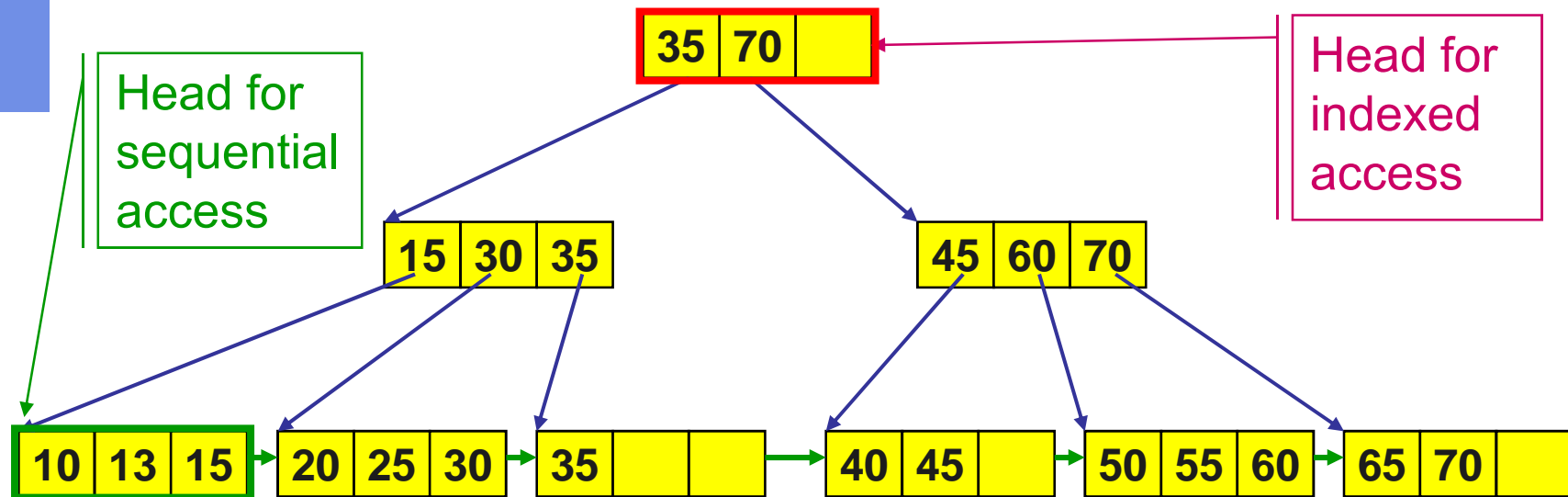
Comer, D.: The Ubiquitous B-Tree, ACM Computing Surveys, 11:2, p. 121 – 138, 1979



B*-Indexed Sequential Files

- Data containers (= leaves of a B*-tree) are linked together \Rightarrow accelerating sequential access
- Inner nodes contain up to *t elements* of lower level inner nodes or of leaves
- Height of a B*-tree only depends on
 - number of records
 - capacity of inner nodes and leaves

Example of a B*-ISF



Remark:

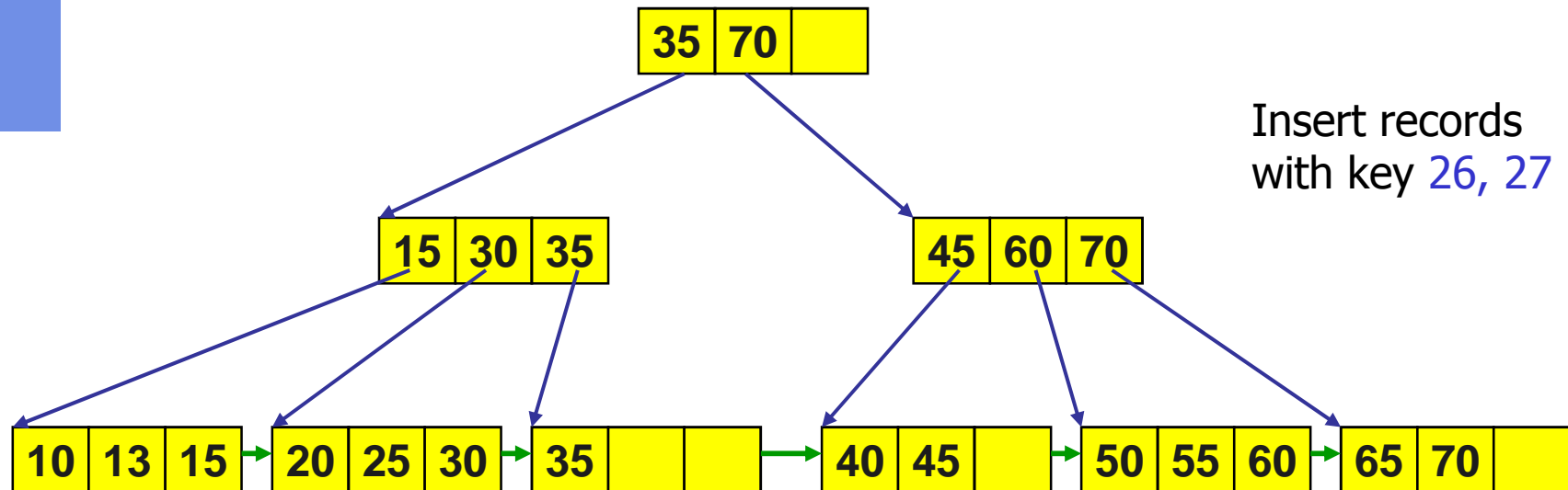
Analogous to an extensible hash file, *minor changes* in the file *might* affect only *neighboring containers* and its *next higher inner node*.

B*-trees try to remain balanced (more or less)

Reorganization of multiple levels are rare (however, possible).



Operation on a B*-File (1)



Insert records
with key **26, 27**

26

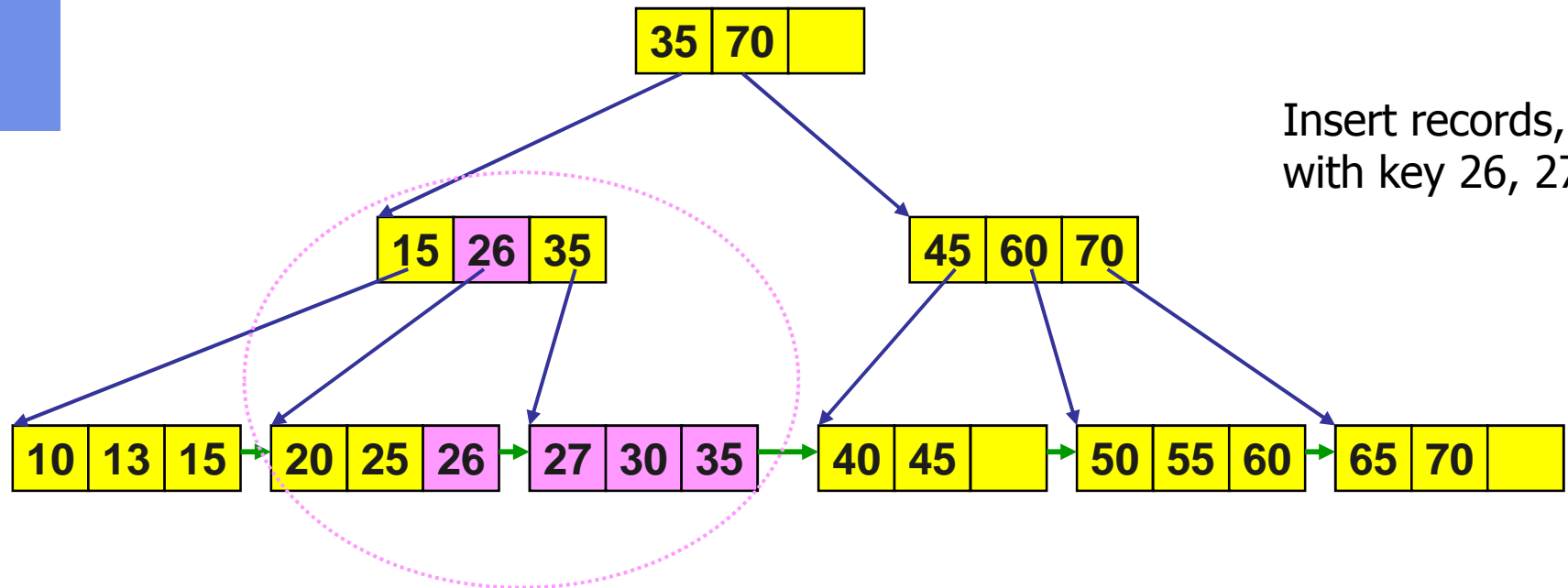
27

The appropriate container is already full,
but the neighbored container to the right isn't,
⇒ try to **move data** to the neighbored container.

*How would you **organize this move-operation**?*



Operation on a B*-File (2)

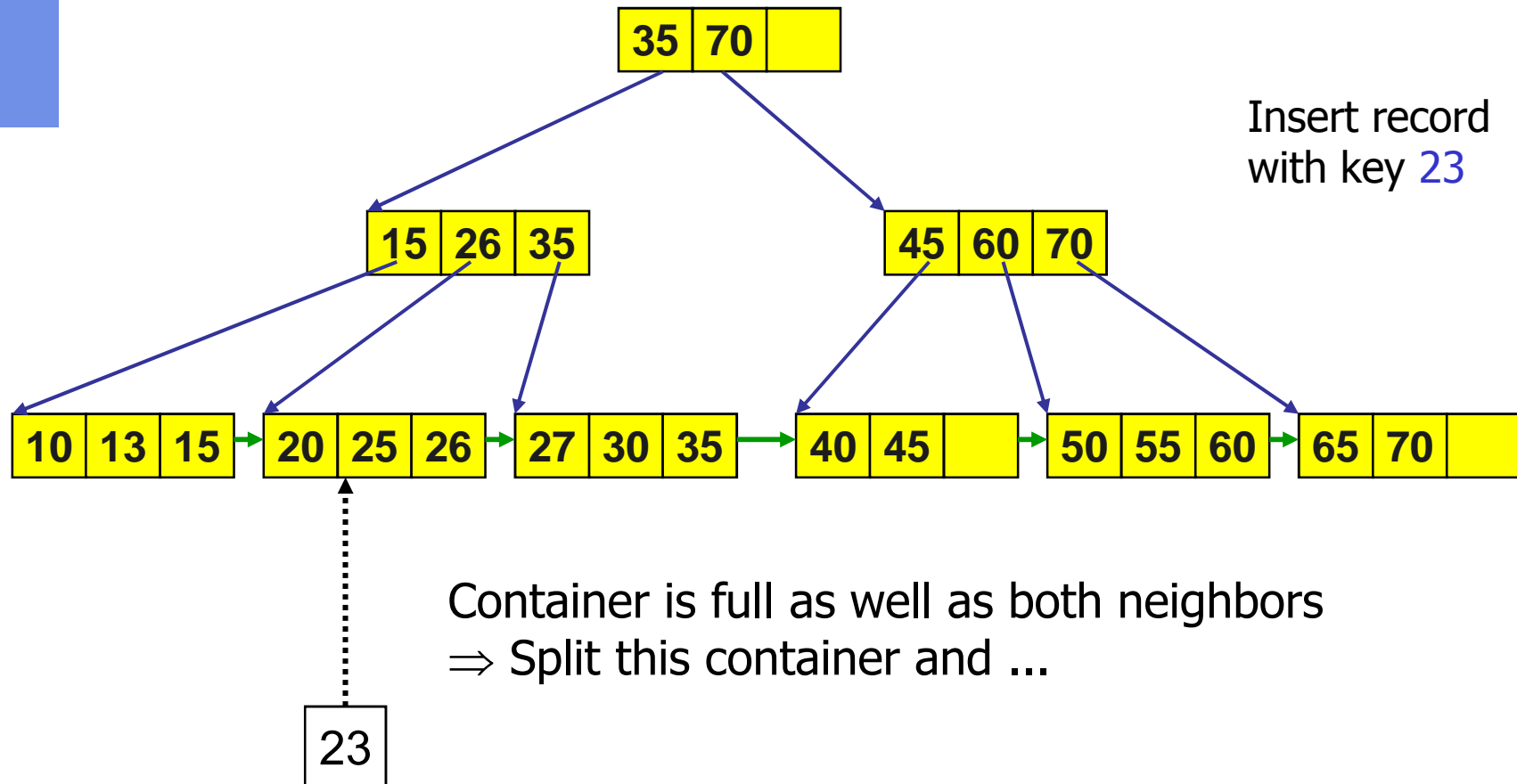


Result: Tree is still balanced (more or less),
modifications in 2 containers and 1 inner node

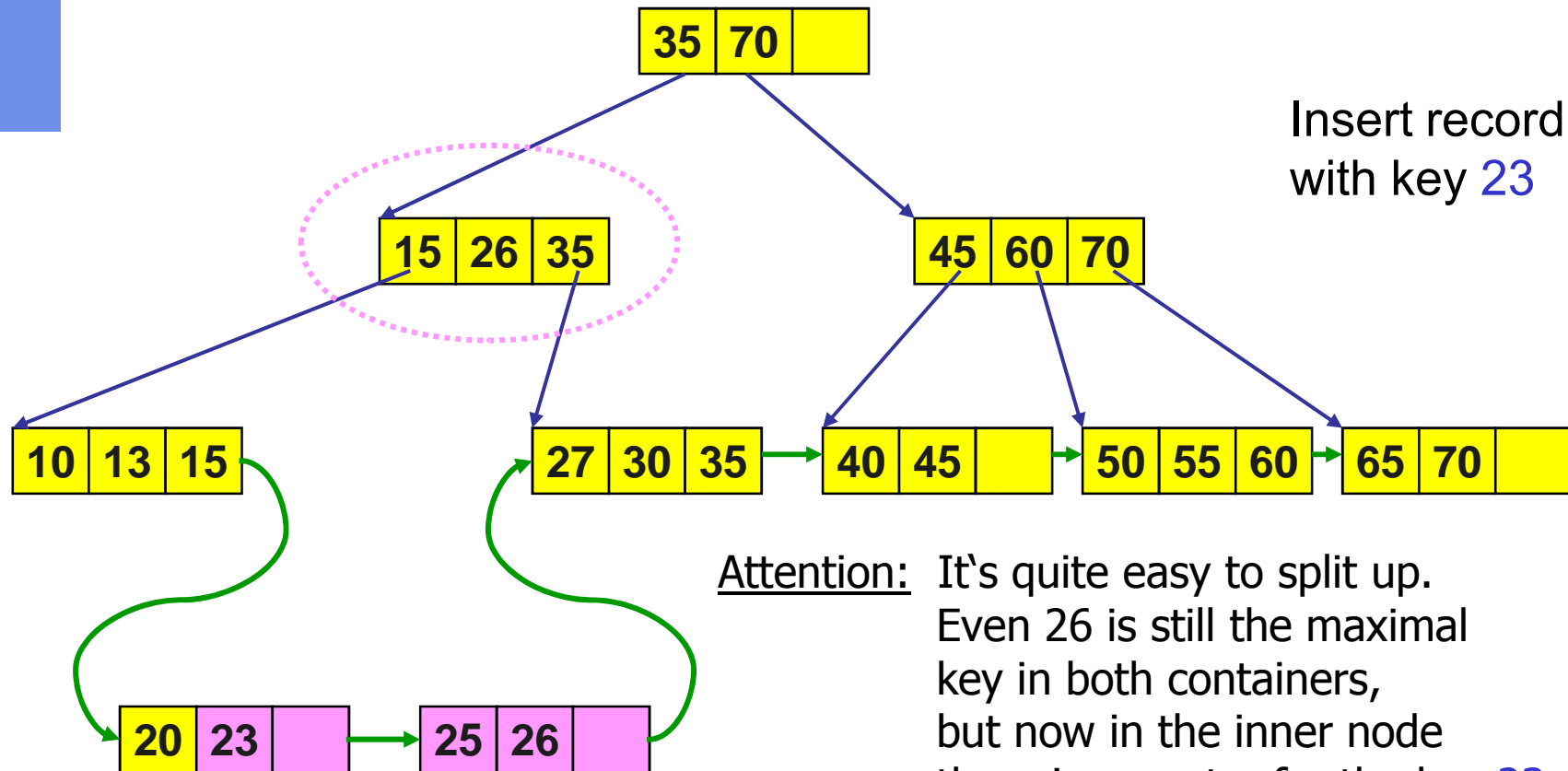
Analysis: Quite cheap, try to apply whenever possible without affecting
the main characteristic of a B*-tree structured index file



Operation on a B*-File (3)



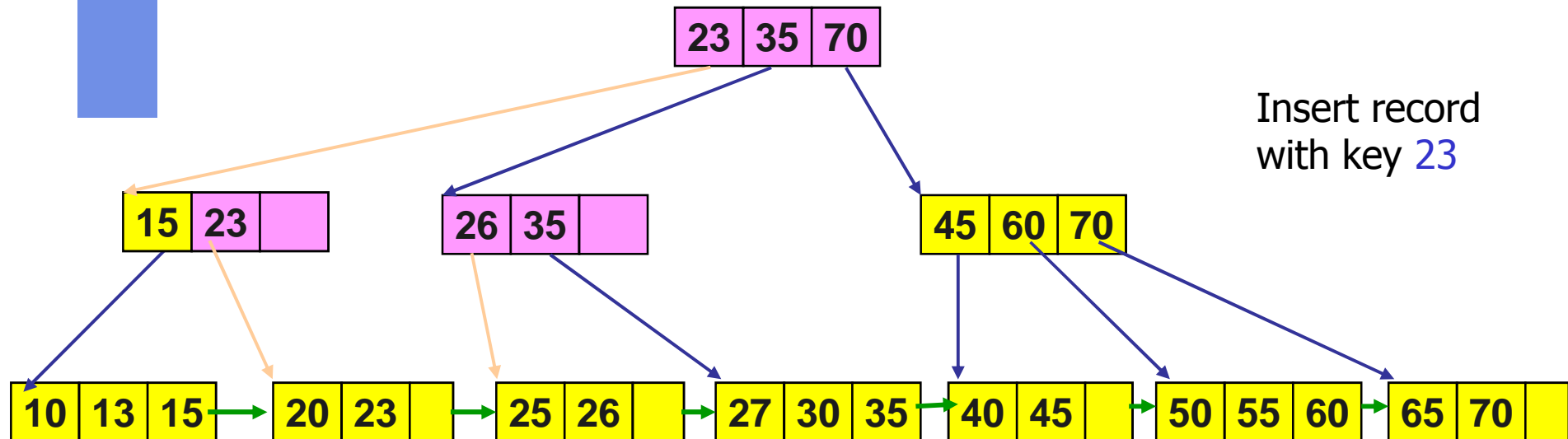
Operation on a B*-File (4)



Insert record with key 23

Attention: It's quite easy to split up. Even 26 is still the maximal key in both containers, but now in the inner node there is no entry for the key 23 => split also the inner node

Operation on a B*-File (5)



Remark:

In this case an update on a lower level may cause updates on higher levels, and might even require a complete new level within the B*-tree)



Multiple Indexed File*

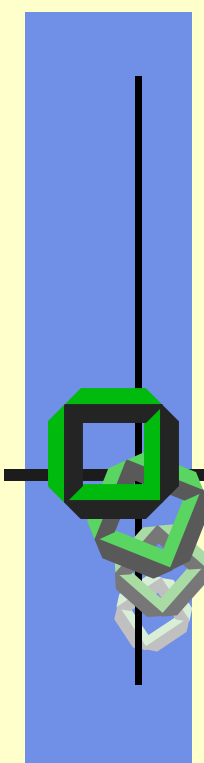
- Uses multiple indexes for different key fields, i.e. more than 1 "field" in the record may be a key
- Might contain an exhaustive index that contains one entry for every record in the main file
- Might contain a partial index, only

Example:

Databases for airline reservation system

⇒ ...

*see course: communication and databases next ST



Extensible Hashing



Direct or Hashed File

- **Key field** k_i required for *each record* r_i
- Key maps directly or via a hash function to an address within the file



Use a hash function $a_i = f(k_i)$, e.g. $a_i = k_i \bmod n$

a_i = address of the data container (\neq physical block number)
 \Rightarrow you need an additional mapping (on lower levels)

If \exists an overflow in the container you have to resolve this collision.



Extensible Hashing

Problem to solve:

Find a *hash function* enabling *incremental file extension*

Solution:

Indirect addressing of scattered containers via a **base vector**

Extension is done via

- doubling the size of this base vector or
- just splitting a container.

Extensible hash function \rightarrow *ind* (=index of base vector):

$$\text{ind} = k \bmod 2^{g_{\text{current_max}}}$$

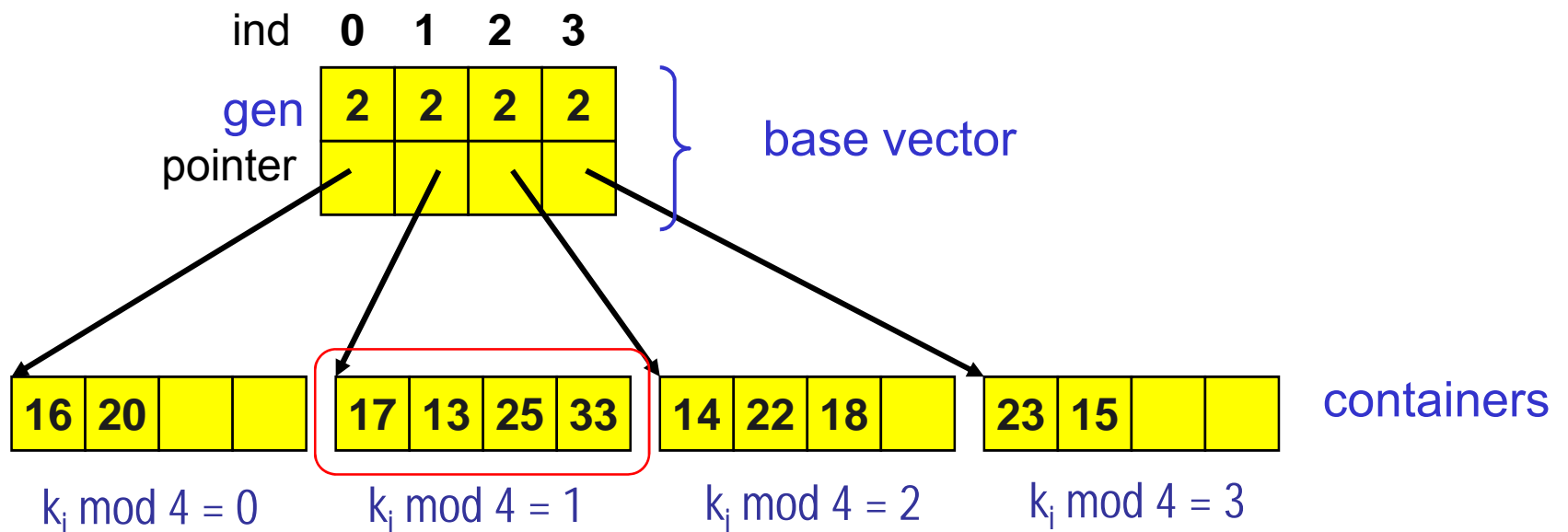
Entry of base vector:

- **generation number g**
- pointer to the target container



Extensible Hashing (2)

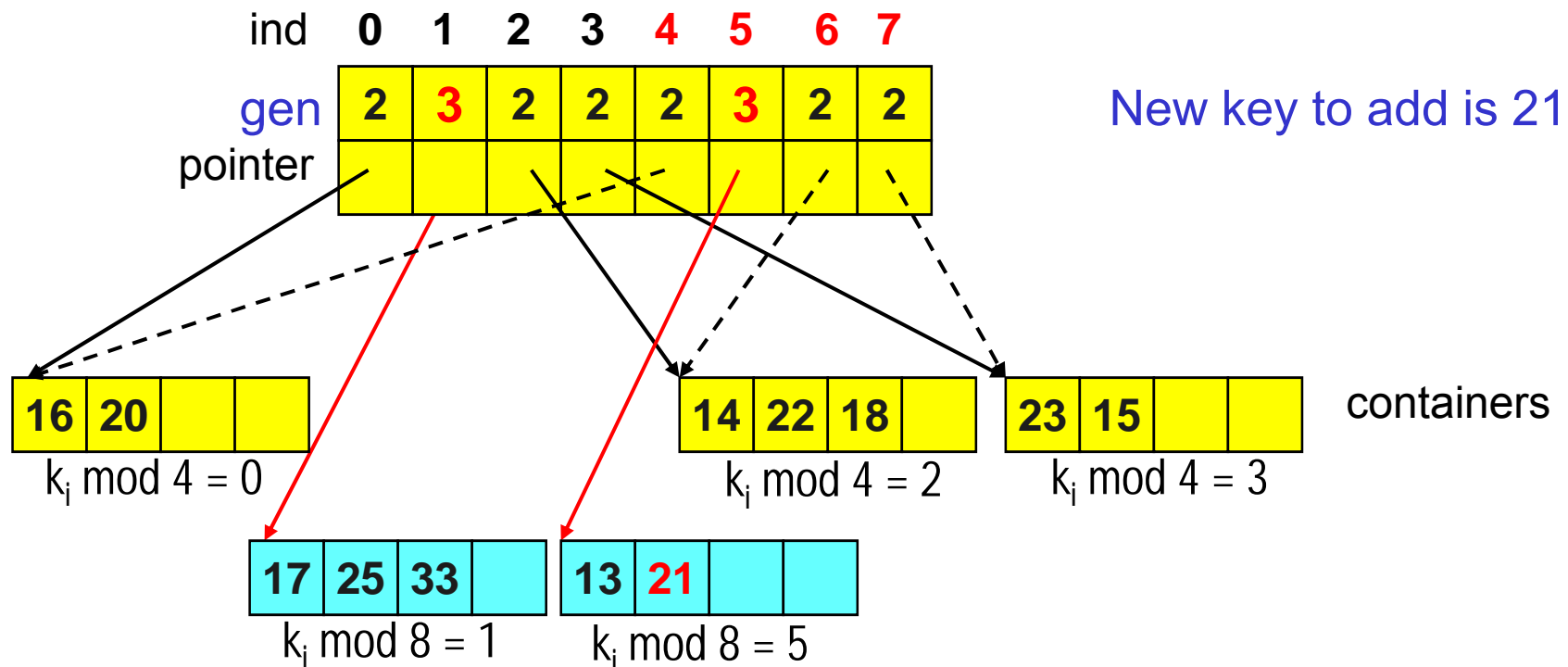
Assumption: current maximal generation number $g_{\max} = 2$
length of base vector = $2^2 = 4$ and \Rightarrow
hash function: $\text{ind} = k_i \bmod 4$
container capacity = 4



Problem: How to add a record with key **21**?

Extensible Hashing (3)

Double base vector \Rightarrow create new $g_{\max} = 3$, update generation.no for index 1 and 5, add another container, **rehash the old container**, insert new element 21, adjust the pointers with generation.no 2

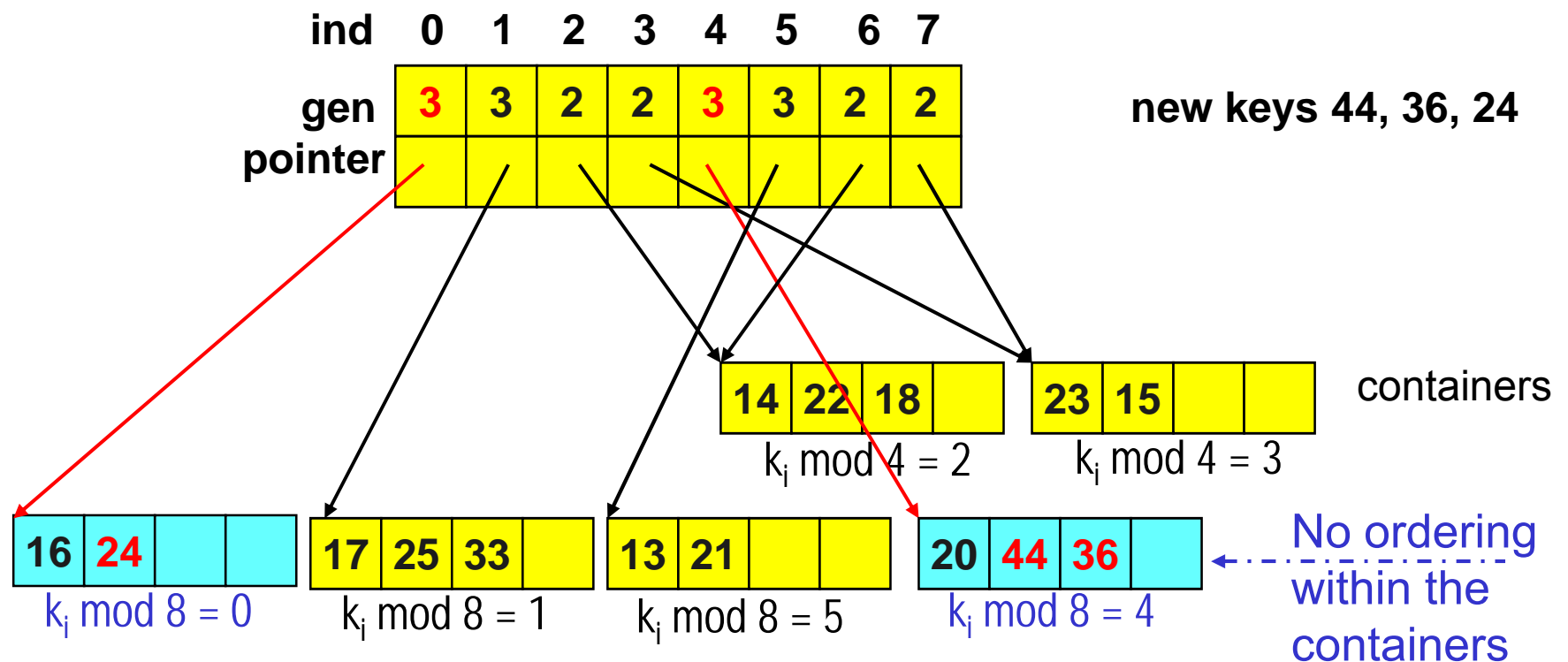


How to add data with keys 44, 36, 24 ?



Extensible Hashing (4)

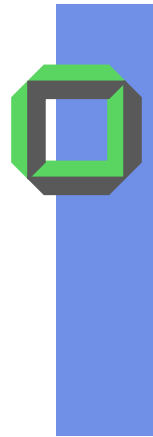
Add **44** and **36** to **container 0**, then only increase generation number for indices $\text{ind} = 0$ and 4 , add another container, rehash both containers, insert **24**, adjust the pointers.





Extensible Hashing (5)

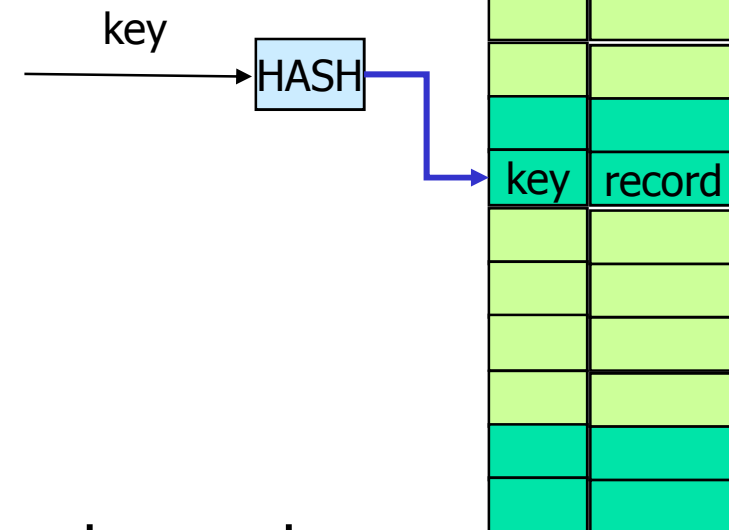
- Find criteria for how to shrink extensible hashed files
- Find criteria for determining length of a container
- Is it possible to get all records of a extensible hashed file?
- Up to now, we've assumed that the key field is a well-behaving bit pattern; *how to find such a pattern?*



Hashed File Performance

- Update
 - Same sized record, very good
 - Variable sized record, **poor**¹

- Retrieval
 - Single record excellent
 - Subset, poor
 - Exhaustive, poor



¹Classic direct files need fixed sized records
 Extensible Hashing allows variable sized records

Hashed File



Extens. Hashing \leftrightarrow B*-Sequential

File method	Usage of space	Update(file)	Retrieval (single record – file)	
Ext. Hashing	$N * C + BV$	$M * O(1+1)$	$O(1+1)$	$M * O(1+1)$
B*-Index-S.	$N * C + \text{Sum}(IN)$	$O(M/C + 1)$	$O(\ln(M)+1)$	$O(M/C + 1)$

N = # of Containers
 C = Capacity of container
 BV = Basis vector
 IN = Capacity of inner node
 M = # of different keys

Start at the sequential head



Summary: File Organization

- A FS should support the most common access methods of its files
- Files are typically used as follows:
 - Most files are very small, **only a few KB**
 - Files are
 - often read
 - sometimes written
 - rarely deleted
 - Sequential access is dominant
 - Files can be shared by multiple applications/users, but they are rarely accessed concurrently