File Organization

Binnur Kurt
binnur.kurt@ieee.org

Istanbul Technical University
Computer Engineering Department

Version 0.1.9

About the Lecturer

- **BSc**
  İTÜ, Computer Engineering Department, 1995
- **MSc**
  İTÜ, Computer Engineering Department, 1997
- Areas of Interest
  - Digital Image and Video Analysis and Processing
  - Real-Time Computer Vision Systems
  - Multimedia: Indexing and Retrieval
  - Software Engineering
  - OO Analysis and Design

Welcome to the Course

- **Important Course Information**
  - Office Hours
    - 14:00-15:00 Tuesday
  - Course Web Page
    - http://www.cs.itu.edu.tr/~kurt/courses/blg341
  - E-mail
    - kurt@ce.itu.edu.tr

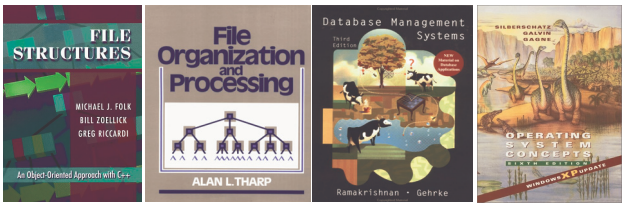
Grading Scheme

- 3 Projects (30%)
- A midterm exam (30%)
- A final exam (40%)
- You must follow the official Homework Guidelines (http://www.ce.itu.edu.tr/lisans/kilavuz.html).
- Academic dishonesty including but not limited to cheating, plagiarism, collaboration is unacceptable and subject to disciplinary actions. Any student found guilty will have grade F. Assignments are due in class on the due date. Late assignments will generally not be accepted. Any exception must be approved. Approved late assignments are subject to a grade penalty.

What we want to see in your programs

- All programs to be written in C/C++
- Self contained, well thought of, and well designed functions/classes
- Clean, well documented code, good programming style
- Modular design
- Do not write codes the way hackers do ☺

References



ITU Main Library
QA.76.73.C153.F65
QA.76.9.F5.T43
QA.76.76.O63.S55

This document is partially based on
http://www.site.uottawa.ca/~lucia/#Teaching

Tell me and I forget.
Show me and I remember.
Let me do and I understand.
—Chinese Proverb

Purpose of the Course

- Objective of Data Structures (BLG221) was to teach ways of efficiently organizing and manipulating data in *main memory*.
- In BLG341E, you will learn equivalent techniques for organization and manipulation of data in *secondary storage*.
- In the first part of the course, you will learn about "low level" aspects of file manipulation (basic file operations, secondary storage devices and system software).
- In the second part of the course, you will learn the most important high-level file structure tools (indexing, co-sequential processing, B trees, Hashing, etc).
- You will apply these concepts in the design of C programs for solving various file management problems

Course Outline

1. Introduction to file management.
2. Fundamental File Processing Operations.
3. Managing Files of Records: Sequential and direct access.
4. Secondary Storage, physical storage devices: disks, tapes and CD-ROM.
5. System software: I/O system, file system, buffering.
6. File compression: Huffman and Lempel-Ziv codes.
7. Reclaiming space in files: Internal sorting, binary searching, keysorting.
8. Introduction to Indexing.
9. Indexing

Course Outline

10. Cosequential processing and external sorting
11. Multilevel indexing and B trees
12. Indexed sequential files and B+ trees
13. Hashing
14. Extendible hashing

1 Introduction to File Management

Content

- Introduction to file structures
- History of file structure design

Introduction to File Organization

- Data processing from a computer science perspective:
  - Storage of data
  - Organization of data
  - Access to data
- This will be built on your knowledge of

Data Structures

Data Structures *vs.* File Structures

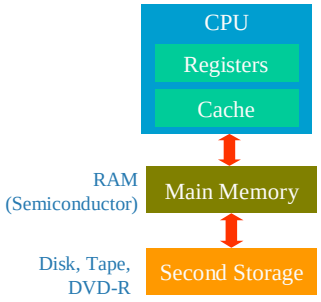
- Both involve:

  Representation of Data

  +

  Operations for accessing data
- Difference:
  - Data Structures deal with data in **main memory**
  - File Structures deal with data in **secondary storage device** (File).

Computer Architecture

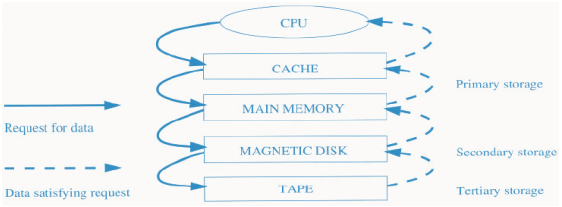


Differences

- Fast
- Small
- Expensive
- Volatile

- Slow
- Large
- Cheap
- Stable

Memory Hierarchy



Memory Hierarchy

- On systems with 32-bit addressing, only 2^{32} bytes can be directly referenced in main memory.
- The number of data objects may exceed this number!
- Data must be maintained across program executions. This requires storage devices that retain information when the computer is restarted.
  - We call such storage nonvolatile.
  - Primary storage is usually volatile, whereas secondary and tertiary storage are nonvolatile.

How Fast?

- Typical times for getting info
  - Main memory: ~120 nanoseconds = 120×10^{-9}
  - Magnetic Disks: ~30 milliseconds = 30×10^{-6}
- An analogy keeping same time proportion as above
  - Looking at the index of a book: 20 seconds

    versus
  - Going to the library: 58 days

Comparison

- Main Memory
  - Fast (since electronic)
  - Small (since expensive)
  - Volatile (information is lost when power failure occurs)
- Secondary Storage
  - Slow (since electronic and mechanical)
  - Large (since cheap)
  - Stable, persistent (information is preserved longer)

Goal of the Course

- Minimize number of trips to the disk in order to get desired information. Ideally get what we need in one disk access or get it with as few disk access as possible.
- Grouping related information so that we are likely to get everything we need with only one trip to the disk (e.g. name, address, phone number, account balance).

Locality of Reference in **Time** and **Space**

Good File Structure Design

- Fast access to great capacity
- Reduce the number of disk accesses
- By collecting data into buffers, blocks or buckets
- Manage growth by splitting these collections

History of File Structure Design

1. In the beginning… it was the tape
   - **Sequential access**
   - Access cost proportional to size of file

   [Analogy to sequential access to array data structure]
2. Disks became more common
   - **Direct access**

     [Analogy to access to position in array]
   - **Indexes** were invented
     - list of keys and points stored in small file
     - allows direct access to a large primary file

     Great if index fits into main memory.

     As file grows we have the same problem we had with a large primary file

History of File Structure Design

3. Tree structures emerged for main memory (1960`s)
   - **Binary search trees (BST`s)**
   - **Balanced**, self adjusting BST`s: e.g. AVL trees (1963)
4. A tree structure suitable for files was invented:

   B trees (1979) and **B+ trees**

   good for accessing millions of records with 3 or 4 disk accesses.
5. What about getting info with a single request?
   - Hashing Tables (Theory developed over 60's and 70's but still a research topic)

     good when files do not change too much in time.
   - Expandable, dynamic hashing (late 70's and 80's)

     one or two disk accesses even if file grows dramatically

2 Fundamental File Processing Operations

Content

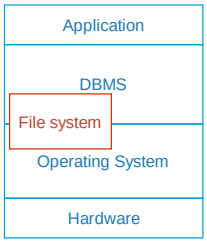
- Sample programs for file manipulation
- Physical files and logical files
- Opening and closing files
- Reading from files and writing into files
- How these operations are done in C and C++
- Standard input/output and redirection

What is a FILE?

A file is...

- A collection of data is placed under permanent or non-volatile storage
- Examples: anything that you can store in a disk, hard drive, tape, optical media, and any other medium which doesn't lose the information when the power is turned off.
- Notice that this is only an informal definition!

Where do File Structures fit in CS?



Physical Files & Logical Files

- Physical file: physically exists on secondary storage; known by the operating system; appears in its file directory
- Logical file, what your program actually uses, a 'pipe' though which information can be extracted, or sent.
- Operating system: get instruction from program or command line; link logical file with physical file or device
- Why is the distinction useful? Why not allow our programs to deal directly with physical files?

Basic File Operations

- **Opening a file** - basically, links a logical file to a physical file.
  - On open, the O/S performs a series operations that end in the program that is trying to open the file being assigned a **file descriptor**.
  - Additionally, the O/S will perform particular operations on the file at the request of the calling program, these operations are intended to 'initialize' the file for use by the program.
  - What happens when the O/S detects an error?

hFile: Logical File, "account.txt": Physical File

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

FILE

```
typedef struct  {
      unsigned char  *curp;       // Current active pointer
      unsigned char  *buffer;     // Data transfer buffer
      int            level;       // fill/empty level of buffer
      int            bsize;       // Buffer size
      unsigned short  istemp;     // Temporary file indicator
      unsigned short  flags;      // File status flags
      wchar_t        hold;        // Ungetc char if no buffer
      char           fd;          // File descriptor
      unsigned char   token;      // Used for validity checking
}     FILE;
```


C++ Counterpart

```
#include <fstream>
#include <iostream>
using namespace std ;
int main(){
    char c;
    fstream infile ;
    infile.open("account.txt",ios::in) ;
    infile.unsetf(ios::skipws) ;
    infile >> c ;
```



```
    while (! infile.fail()){
       cout << c ;
       infile >> c ;
    }
    infile.close() ;
    return 0;
}
```


Physical Files & Logical Files – Revisited # 1

- OS is responsible for associating a logical file in a program to a physical file in disk or tape. Writing to or reading from a file in a program is done through the OS.
- Note that from the program point of view, input devices (keyboard) and output devices (console, printer, etc) are treated as files – places where bytes come from or sent to
- There may be thousands of physical files on a disk, but a program only have a limited number of logical files open at the same time.
- The physical file has a name, for instance “account.txt”
- The logical file has a logical name used for referring to the file inside the program. The logical name is a variable inside the program, for instance “infile”

Physical Files & Logical Files – Revisited # 2

- In C PL, this variable is declared as.

  FILE *infile ;
- In C++ PL, the logical name is the name of an object of the class **fstream**:

  fstream infile ;
- In both languages, the logical name infile will be associated to the physical file “account.txt” at the time of opening the file.

More on Opening Files

- Two options for opening a file:
  - Open an **existing** file
  - Create a **new** file

How to do in C

FILE *outfile;

outfile = fopen(“account.txt”, “w”) ;

- The 1st argument indicates the physical name of the file
- The 2nd one determines the “mode”

  —the way the file is opened

The Mode

- “r”: open an existing file for reading
- “w”: create a new file, or truncate existing one, for writing
- “a”: open a new file, or append an existing one for writing
- “r+”: open an existing file for reading and writing
- “w+”: create a new file, or truncate an existing one for reading and writing
- “a+”: create a new file, or append an existing one for reading and writing

How to do in C++

fstream outfile;

outfile.open(“account.txt”, ios::out) ;

- The 1st argument indicates the physical name of the file
- The 2nd argument is an integer indicating the mode defined in the class **ios**.

The Mode

- ios::in open for reading
- ios::out open for writing
- ios::app seek to the end of file before each write
- ios::trunc always create a new file
- ios::nocreate fail if file does not exist
- ios::binary open in binary mode

Basic File Operations

- **Closing a file** - cuts the link between physical and logical files
  - Upon closing, the OS takes care of ‘synchronizing’ the contents of the file, e.g. often a buffer is used, need to write buffer content to file.
  - In general, files are automatically closed when the program ends.
  - So, why do we need to worry about closing files?
  - In C: **fclose(outfile)**
  - In C++: **outfile.close()**

Basic File Operations

- **Reading and Writing** – basic I/O operations.
  - Usually require three parameters: **a logical file**, **an address**, and the **amount of data** that is to be read or written.
  - What is the use of the **address** parameter?

Reading in C

```
char c ; // a character
char a[100] ; // an array with 100 characters
FILE * infile ;
:
infile = fopen("myfile.txt", "r") ;
fread(&c,1,1,infile) ; // reads one character
fread(a,1,10,infile) ; // reads 10 characters
```


fread()

```
fread(&c,1,1,infile) ; // reads one character
fread(a,1,10,infile) ; // reads 10 characters
```

- ▶ 1st argument: destination address
- ▶ 2nd argument: element size in bytes
- ▶ 3rd argument: number of elements
- ▶ 4th argument: logical file name

Reading in C++

```
char c ; // a character
char a[100] ; // an array with 100 characters
fstream infile ;
infile.open("myfile.txt", ios::in) ;
infile >> c; // reads one character
infile.read(&c,1) ;
infile.read(a,10); // reads 10 bytes
```

- ▶ Note that thanks to operator overloading in C++, operator >> gets the same info at a higher level

Writing in C

```
char c ; // a character
char a[100] ; // an array with 100 characters
FILE * outfile ;
outfile = fopen("myfile.txt", "w") ;
fwrite(&c,1,1,outfile) ; // writes one character
fwrite(a,1,10,outfile) ; // writes 10 characters
```


Writing in C++

```
char c ; // a character
char a[100] ; // an array with 100 characters
fstream outfile ;
outfile.open("myfile.txt", ios::out) ;
outfile << c; // writes one character
outfile.write(&c,1) ;
outfile.write(a,10); // writes 10 bytes
```


Additional File Operations

- ▶ Seeking: source file, offset.
- ▶ Detecting the end of a file
- ▶ Detecting I/O error

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

Seeking with C++ Stream Classes

A fstream has 2 file pointers: get pointer & put pointer
(for input) (for output)

file1.seekg (byte_offset, origin); //moves get pointer
file1.seekp (byte_offset, origin); //moves put pointer

origin can be ios::beg (beginning of file)
ios::cur (current position)
ios::end (end of file)

file1.seekg (373, ios::beg); // moves get pointer 373 bytes from
// the beginning of file

Calculating File Size

```
int main(int argc, char* argv[]) {
   FILE *hFile=fopen(argv[1],"r");
   fseek(hFile, 0L, SEEK_END);
   int fileLength = ftell(hFile);
   printf("\nFile size is %d",fileLength) ;
   fclose(hFile) ;
   return 0;
}
```


Detecting End of File

- In C: Check whether fread returned value 0

```
int i ;
i = fread(&c,1,1,infile) ; //attempt to read
if (i==0) // true if file has ended
```

- Alternatively, use the function feof(infile)
- In C++: Check whether infile.fail() returns true

```
infile >> c ;
if (infile.fail()) // true if file has ended
```

- Alternatively, use the function infile.eof()
- Also note that fail() indicates that an operation is unsuccessful, so it is more general than just checking for end of file

Logical File Names Associated to Std IO Devices

Purpose	Default Meaning	Logical Name	
		in C	in C++
Standard Output	Console/Screen	stdout	cout
Standard Input	Keyboard	stdin	cin
Standard Error	Console/Screen	stderr	cerr

- These streams do not need to be open or closed in the program

Redirection

- Some OS allow the default meanings to be changed through a mechanism called redirection
- Example in Unix
  - Suppose that "prog" is the executable program
  - Input redirection (standard input becomes file in.txt)
    - prog < in.txt
  - Output redirection (standard output becomes file out.txt)
    - prog > out.txt
  - You can also do
    - prog < in.txt > out.txt

3 Managing Files of Records

Content

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

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;
}
```


"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).

05873507CARROLL19ALICE IN WONDERLAND
05038104FOLK15FILE STRUCTURES
05866830 5KNUTH15SURREAL NUMBERS
05183950 7TOLKIEN10THE 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 = $30 \times 101 = 3030$
- 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

- 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

- 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 primary key 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];
};
```

4 Secondary Storage Devices

Content

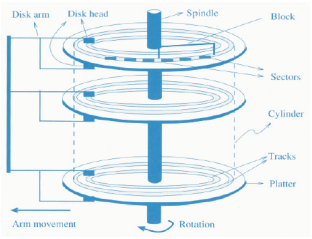
- Secondary storage devices
- Organization of disks
- Organizing tracks by sector
- Organizing tracks by blocks
- Non-data overhead
- The cost of a disk access
- Disk as a bottleneck

Secondary Storage Devices

- Since secondary storage is different from main memory we have to understand how it works in order to do good file designs.
- Two major types of storage devices
  - Direct Access Storage Devices (DASDs)
    - Magnetic Disks
      Hard Disks (high capacity, low cost per bit)
    - Optical Disks
      CD-ROM,DVD-ROM
      (Read-only/write-once, holds a lot of data, cheap)
  - Serial Devices
    - Magnetic Tapes

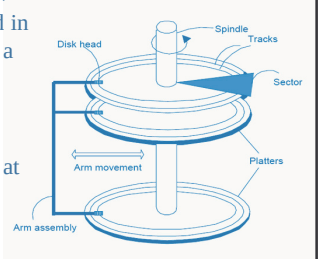
Magnetic Disks

- Magnetic disks support direct access to a desired location
- Simplified structure of a disk
  - Disk blocks
  - Tracks
  - Platters
  - Cylinder
  - Sectors
  - Disk heads
  - Disk Controller
  - Seek Time
  - Rotational delay



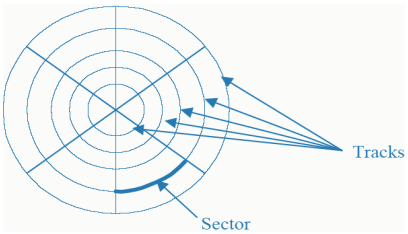
Components of a Disk

- The platters spin (7200 rpm)
- The arm assembly is moved in or out to position a head on a desired track. Tracks under heads make a *cylinder* (imaginary!).
- Only one head reads/writes at any one time
- *Block size* is a multiple of *sector size* (which is fixed)



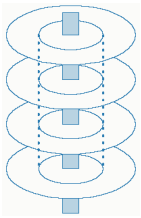
Looking at a Surface

- Disk contains concentric **tracks**
- **Tracks** are divided into **sectors**
- A sector is the smallest addressable unit in disk



Cylinder

- Cylinder: the set of tracks on a disk that are directly above/below each other
- All the information on a cylinder can be accessed without moving the read/write arm (seeking)



The Bottleneck

- When a program reads a byte from the disk, the operating system locates the surface, track and sector containing that byte, and reads the entire sector into a special area in main memory called buffer.
- The bottleneck of a disk access is moving the read/write arm. So, it makes sense to store a file in tracks that are below/above each other in different surfaces, rather than in several tracks in the same surface.

How to Calculate Disk Capacity

- Number of cylinders = number of tracks in a surface
- Track capacity = number of sector per track
  ×
  bytes per sector
- Cylinder capacity = number of surfaces
  ×
  track capacity
- Drive capacity = number of cylinders
  ×
  cylinder capacity

An Example

- We have fixed-length records
- Number of records = 50.000 records
- Size of a record = 256 bytes
- Disk characteristics
  - Number of bytes per sector = 512
  - Number of sectors per track = 63
  - Number of tracks per cylinder = 16
  - Number of cylinders = 4092
- How many cylinders are needed?

Clusters, Extents and Fragmentation

- The file manager is the part of the operating system responsible for managing files
- The file manager maps the logical parts of the file into their physical location
- A **cluster** is a fixed number of contiguous sectors
- The file manager allocates an integer number of clusters to a file. An example: Sector size: 512 bytes, Cluster size: 2 sectors
  - If a file contains 10 bytes, a cluster is allocated (1024 bytes).
  - There may be unused space in the last cluster of a file. This unused space contributes to **internal fragmentation**

More on Clusters

- Clusters are good since they improve sequential access: reading bytes sequentially from a cluster can be done in one revolution, seeking only once.
- The file manager maintains a file allocation table (FAT) containing for each cluster in the file and its location in disk
- An extent is a group of contiguous clusters. If file is stored in a single extent then seeking is done only once.
- If there is not enough contiguous clusters to hold a file, the file is divided into 2 or more extents.

Fragmentation

- Due to records not fitting exactly in a sector
  - Example: Record size = 200 bytes, sector size = 512 bytes
  - to avoid that a record span 2 sectors we can only store 2 records in this sector (112 bytes go unused per sector
  - the alternative is to let a record span two sectors, but in this case two sectors must be read when we need to access this record)
- Due to the use of clusters
  - If the file size is not multiple of the cluster size, then the last cluster will be partially used.

How to Choose Cluster Size

- Some OS allow the system administrator to choose the cluster size.
- When to use **large cluster size**?
  - When disks contain large files likely to be processed sequentially.
  - Example: Updates in a master file of bank accounts (in batch mode)
- What about **small cluster size**?
  - When disks contain small files and/or files likely to be accessed randomly
  - Example : online updates for airline reservation

Organizing Tracks By Blocks

- Disk tracks may be divided into user-defined blocks rather than into sectors.
- The amount transferred in a single I/O operation can vary depending on the needs of the software designer
- A block is usually organized to contain an integral number of logical records.
- Blocking Factor = number of records stored in each block in a file
- No internal fragmentation, no record spanning two blocks

SubBlocks

- A block typically contains subblocks:
- Count subblock: contains the number of bytes in a block
- Key subblock (optional): contains the key for the last record in the data subblock (disk controller can search can search for key without loading it in main memory)
- Data subblock: contains the records in this block.

NonData Overhead

- Amount of space used for extra stuff other than data
- **Sector-Addressable Disks**
  - at the beginning of each sector some info is stored, such as sector address, track address, condition (if sector is defective);
  - there is some gap between sectors
- **Block-Organized Disks**
  - subblocks and interblock gaps is part of the extra stuff; more nondata overhead than with sector-addressing.

An Example

- Disk characteristics
  - Block-addressable Disk Drive
  - Size of track = 20.000 bytes
  - Nondata overhead per block = 300 bytes
- File Characteristics
  - Record size = 100 bytes
- How many records can be stored per track for the following blocking factors?
  1. Block factor = 10
  2. Block factor = 60

Solution for the Example

- Blocking factor is 10
- Size of data subblocks = 1000
- Number of blocks that can fit in a track =

$$\left\lfloor \frac{20000}{1300} \right\rfloor = \lfloor 15.38 \rfloor = 15$$

- Number of records per track = 150 records

Solution for the Example

- Blocking factor is 60
- Size of data subblocks = 6000
- Number of blocks that can fit in a track =

$$\left\lfloor \frac{20000}{6300} \right\rfloor = \lfloor 3.17 \rfloor = 3$$

- Number of records per track = 180 records

Accessing a Disk Page

- Time to access (read/write) a disk block
  - *seek time* (moving arms to position disk head on track)
  - *rotational delay* (waiting for block to rotate under head)
  - *transfer time* (actually moving data to/from disk surface)
- Seek time and rotational delay dominate
  - Seek time varies from 1 to 20 msec
  - Rotational delay varies from 1 to 10 msec
  - Transfer rate is about 1msec fro 4KB page
- Key to lower I/O cost: reduce seek/rotation delays: Hardware vs. Software solutions?

An Example of a Current Disk

- Model Seagate ST3200822A
- Capacity 200GB
- Transfer Rate
  - Maximum Internal 683Mbits/sec
  - Maximum External 100Mbytes/sec
- Discs/Heads 2/4
- Bytes Per Sector 512
- Spindle Speed 7200 rpm
- Average Seek 8.5 milliseconds
- Average Latency 4.16 milliseconds

What is the average time to read one Sector?

- Transfer time = revolution time / #sectors per track

$$\frac{\left(\frac{1}{7200}\right)}{170} = \frac{\left(\frac{60}{7200}\right)}{170} = \frac{6}{720 \times 170} = \frac{6}{122400} \cong 0.05 msec$$

- Average totaltime = average seek time +
  average rottional delay +
  transfer time

  8.5 + 4.16 + 0.05 = 12.71 msec

Disk as a Bottleneck

- Processes are often disk-bound
- network and CPU have to wait a long time for the disk to transmit data
- Various techniques to solve this problem
  1. **Multiprocessing**: (CPU works on other jobs while waiting for the disk)
  2. **Disk Striping**:
     - Putting different blocks of the file in different drives.
     - Independent processes accessing the same file may not interfere with each other (parallelism)
  3. **RAID** (Redundant Array of Independent Disks)
  4. **RAM** Disk (Memory Disk) Piece of main memory is used to simulate a disk (speed *vs.* volatility)

Disk as a Bottleneck (Con't)

- Various techniques to solve this problem
  5. **Disk Cache**:
     - Large block of memory configured to contain pages of data from a disk.
     - When data is requested from disk, first the cache is checked.
     - If data is not there (miss) the disk is accessed

RAID

- Disk Array: Arrangement of several disks that gives abstraction of a single, large disk.
- Goals: Increase performance and reliability.
- Two main techniques
  - Data striping: Data is partitioned; size of a partition is called the striping unit. Partitions are distributed over several disks.
  - Redundancy: More disks → more failures. Redundant information allows reconstruction of data if a disk fails.

RAID Levels

- Level 0: No redundancy
- Level 1: Mirrored (two identical copies)
  - Each disk has a mirror image (check disk)
  - Parallel reads, a write involves two disks
  - Maximum transfer rate=transfer rate of one disk
- Level 0+1: Striping and Mirroring
  - Parallel reads, a write involves two disks
  - Maximum transfer rate = aggregate bandwidth

RAID Levels

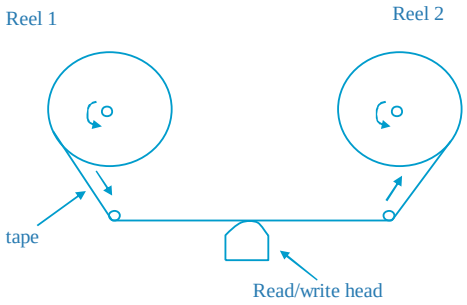
- Level 3: Bit-Interleaved Parity
  - Striping Unit: One bit. One Check Disk.
  - Each read and write request involves all disks; disk array can process one request at a time.
- Level 4: Block-Interleaved Parity
  - Striping Unit: One Disk Block. One Check Disk.
  - Parallel reads possible for small requests, large requests can utilize full bandwidth
  - Writes involve modified block and check disk
- Level 5: Block-Interleaved Distributed Parity
  - Similar to RAID Level 4, but parity blocks are distributed over all disks

Tapes

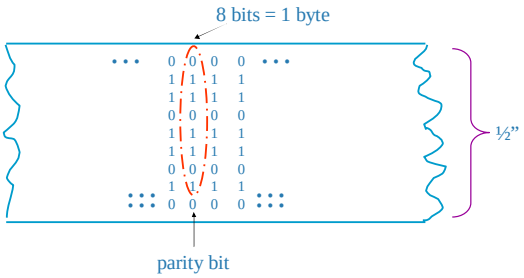
- Tapes
  - are relatively inexpensive
  - can store very large amounts of data
  - good choice for ***archival*** storage
    - we need to maintain data for a long period
    - we do not expect to access it very often
- The main drawback of tapes
  - they are sequential access devices
  - we must essentially step through all the data in order
  - cannot directly access a given location on tape
  - Mostly used to back up operational data periodically

Magnetic Tape

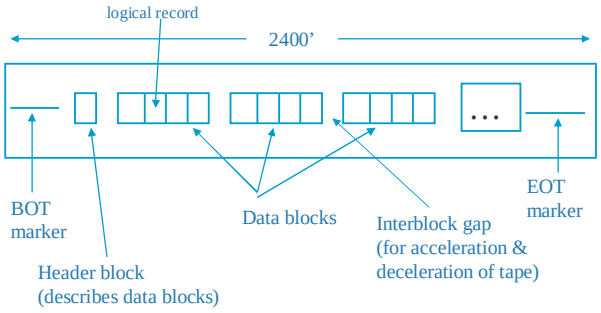
- A set of parallel tracks
- 9 tracks - parity bit
- Frame
  - one-bit-wide slice of tape
- Interblock gaps
  - permit stopping and starting



In detail



Tape Organization



Estimating Tape Length

- There is an interblock gap for each data block
- Space requirement **s**

$$s = n \times (b + g)$$

  - **b** is the physical length of a data block
  - **g** is the length of an interblock gap
  - **n** is the number of data blocks
- Tape density
- Tape speed
- Size of interblock gap

Estimating Tape Length (Con't)

- Example:
  - one million 100-byte records
  - 6,250 BPI tape
  - 0.3 inches of interblock gap
- How much tape is needed?
  - when blocking factor is between 1 and 50
- Nominal recording density
- Effective recording density:
  - number of byte per block / number of inches for block

Estimating Data Transmission Times

- Factors of data transmission rate
  - interblock gaps
  - effective recording density
  - nominal recording density
  - speed of r/w head
  - time to start/stop the tape

Disks *vs.* Tapes

- **Disk**
  - Random access
  - Immediate access
  - Expensive seek in sequential processing
- **Tape**
  - Sequential access
  - Long-term storage
  - No seek in sequential processing

- Decrease in cost of disk and RAM
- More RAM space is available in I/O buffers,
- so disk I/O decreases
- Tertiary storage for backup: CD-ROM, tape ...

Example: Quantum DLT 8000

- **Sustained Transfer Rate (MB/sec)**
  - **Native** 6
  - **Compressed** (up to) 121
- **Burst Transfer Rate (MB/sec)**
  - **Synchronous** 20
  - **Asynchronous** 12
- **Formatted Capacity (GB)**
  - **Native** 40
  - **Compressed** 80
- **Average File Access Time (sec)** 60
- **Interface** SCSI–2 Fast/Wide

Introduction to CD-ROM

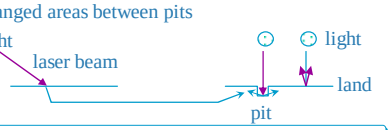
- CD-ROM: Compact Disc Read-Only Memory
  - Can hold over 600MB (200,000 pages)
  - Easy to replicate
  - Useful for publishing or distributing medium
  - But, not storing and retrieving data
- CD-ROM is a child of CD audio
- CD audio provides
  - High storage capacity
  - Moderate data transfer rate
  - But, against high seek performance
    - →Poor seek performance

History of CD-ROM

- CD-ROM
  - Philips and Sony developed CD-ROM in 1984 in order to store music on a disc
  - Use a digital data format
  - The development of CD-ROM as a licensing system results in widely acceptance in the industry
  - Promised to provide a standard physical format
  - Any CD-ROM drive can read any sector which they want
  - Computer applications store data in a file not in terms of sector, thus, file system standard should be needed
  - In early summer of 1986, an official standard for organizing files was worked out

Physical Organization of Master Disk

- **Master Disc**
  - Formed by using the digital data, 0 or 1
  - Made of glass and coated that is changed by the laser beam
- **Two part of CD-ROM**
  - **Pit**
    - The areas that is hit by the laser beam
    - Scatter the light
  - **Land**
    - Smooth, unchanged areas between pits
    - Reflect the light



Encoding Scheme of CD-ROM

- **Encoding scheme**
  - The alternating pattern of high- and low-density reflected light is the signal
  - 1 : transition from pit to land and back again
  - 0s : the amount of time between transitions
- **Constraint**

1_{10}	00000001_2	10000100000000_{EFM}

  - The limits of resolution of the optical pickup, there must be at least two 0's between any pair of 1's (no two adjacent 1s)
  - We cannot represent all bit patterns, thus, we need translation scheme
  - We need at least 14 bits to represent 8 bits under this constraint

Format of CD-ROM

- CD audio chose CLV format instead of CAV format
  - CD audio requires large storage space
  - CD audio is played from the beginning to the end sequentially

Format of CD-ROM

- Format of CD-ROM
  - CLV(Constant Linear Velocity)
  - A single spiral pattern
  - Same amount of space for each sector
  - Capability for writing all of sectors at the maximum density
  - Rotational speed is slower in reading outer edge than in inner edge
  - Finding the correct speed though trial and error
  - Characteristics
    - Poor seek performance
    - No straightforward way to jump to a specified location

Constant Angular Velocity Disk

- Magnetic disk usually uses CAV(Constant Angular Velocity)
  - Concentric tracks and pie-shaped sectors
  - Data density is higher in inner edge than in outer edge
  - Storage waste: total storage is less than a half of CLV
  - Spin the disc at the same speed for all positions
  - Easy to find a specific location on a disk → good seek performance

Addressing of CD-ROM

- Addressing
  - Magnetic disk: cylinder/track/sector approach
  - CD-ROM: a sector-addressing scheme
- Track density varies thus, each second of playing time on a CD is divided into 75 sectors
  - 75 sectors/sec, 2 Kbytes/sector
  - At least one-hour of playing time
  - Maximum capacity can be calculated: 600 Mbytes

    60 min * 60 sec/min * 75 sectors/sec = 270,000 sectors
- We address a given sector by referring minutes, second, and sector of play
  - 16:22:34 means 34th sector in the 22nd second in the 16th minutes of play

Fundamental Design of CD Disc

- Initially designed for delivering digital audio information
- Store audio data in digital form
- Wave patterns should be converted into digital form
- Measure of the height of the sound: 65,536 different gradation(16 bits)
- Sampling rate: 44.1 kHz, because of 2 times of 20,000 Hz upto which people can listen
- 16 bits sample, 44,100 times per second, and two channel for stereo sound, we should store 176,400 bytes per seconds
- Storage capacity of CD is 75 sectors per seconds, we have 2,352 bytes per sector
- CD-ROM divides this raw sector as shown in the following figure

Raw Sector

File Structure Problem of CD-ROM

- Strong and weak sides of CD-ROM
  - Strong aspects of CD-ROM
    - Data transfer rate: 75 sectors/sec
    - Storage capacity : over 600 Mbytes
    - Inexpensive to duplicate and durable
  - Weak aspects of CD-ROM
    - Poor seek performance (weak random access)
      - » Magnetic disk: 30 msec, CD-ROM : 500 msec
  - Comparison of access time of a large file from several media
    - RAM: 20 sec, Disk: 58 days, CD-ROM: 2.5 years
- We should have a good file structure avoiding seeks to an even greater extent that on magnetic disk

What is DVD?

- DVD
  - Digital Video disk (DVD-Video)
  - Digital Versatile disk (DVD-ROM)
- In September 1995
  - As a movie-playback format
  - As a computer-ROM format
- Next-Generation optical disc storage technology will replace audio-CD, videotape, laserdisk, CD-ROM, etc.

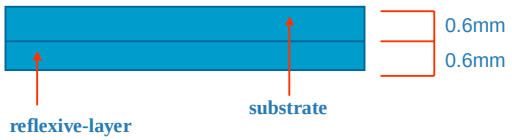
The History from CD to DVD

- 1980, Sony & Philips → CD-Audio
- 1985, Sony & Philips → CD-ROM
- 1989, Sony & Philips → CD-I
- 1990, Sony & Philips → CD-R
- 1995, → CD-E
- 1995, September → DVD

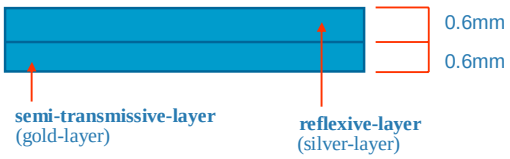
DVD Capacity

- Single-sided
  - DVD5 (4.7 GB/single-layer)
  - DVD9 (8.5 GB/dual-layer)
- Double-sided
  - DVD10 (9.4 = 4.7x2 GB/dual-layer)
  - DVD18 (17 = 8.5x2 GB/dual-layer)
- Write-Once
  - DVD-R (3.8 GB/side)
- Overwrite
  - DVD-RAM (more than 2.6 GB/side)

Single sided, single layer



Single sided, dual layer



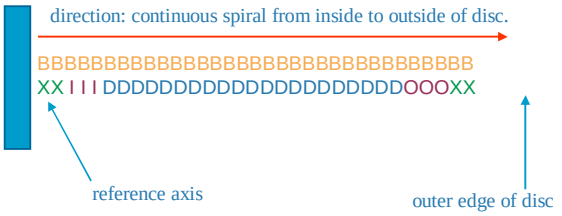
CD vs. DVD

- **Laser-Beam**
  - CD → infrared light (780nm)
  - DVD → red light (635-650nm)
- **Capacity**
  - CD → maximum 680MB
  - DVD → maximum 17GB (25 times of CD)
- **Reference Speed**
  - CD → 1.2m/sec. CLV
  - DVD → 4.0m/sec. CLV

Track Structure

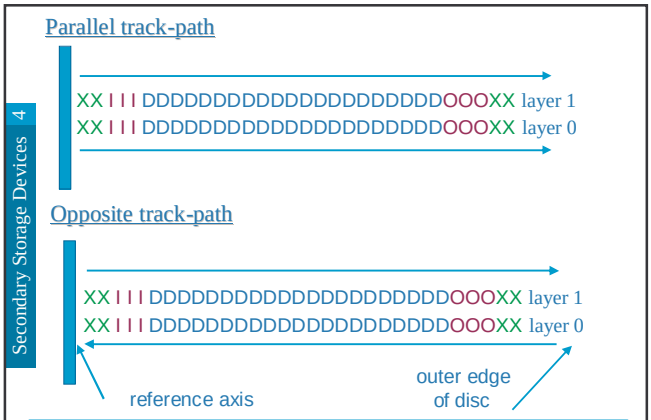
- **Legend**
  - **I** Lead-in area (leader space near edge of disc)
  - **D** Data area (contains actual data)
  - **O** Lead-out area(leader space near edge of disc)
  - **X** Unusable area (edge or donut hole)
  - **M** Middle area (interlayer lead-in/out)
  - **B** Dummy-bonded layer
    (to make disc 1.2mm thick instead of 0.6mm)

Single Layer Disc



Dual Layer Disc

(A) Parallel track path (for computer CD-ROM use)
Direction : same for both layers.
(B) Opposite track path (for movies)
Direction : opposite directions
(Since the reference beam and angular velocities are the same at the layer transition point, the delay comes from refocusing. This permits seamless transition for movie playback.)



Sector Structure

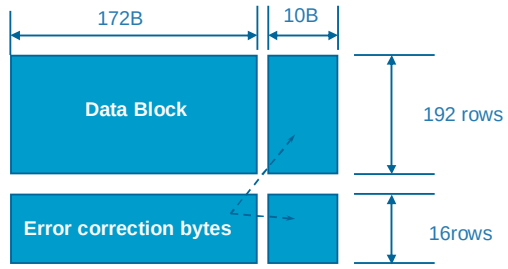
- **2064 bytes/sector**
  - organized into 12 rows, each with 172B
  - first row starts with 12B sector header
    (ID,IEC,Reserved bytes)
  - final row is punctuated with 4B (EDC bytes)

Row	Fields within row
0	ID(4B) IEC(2B) RESERVED(6B) Main data(160B : D[0]-D[159])
1	Main data(172B : D[160]-D[331])
2	Main data(172B : D[332]-D[503])
3	Main data(172B : D[504]-D[675])
4	Main data(172B : D[676]-D[847])
5	Main data(172B : D[848]-D[1019])
6	Main data(172B : D[1020]-D[1191])
7	Main data(172B : D[1192]-D[1363])
8	Main data(172B : D[1364]-D[1535])
9	Main data(172B : D[1536]-D[1707])
10	Main data(172B : D[1708]-D[1879])
11	Main data(168B : D[1880]-D[2047]) EDC(4B)

ID : Identification Data (32bit sector number)
IEC : ID Error Correction
EDC : Error Detection Code

Block Structure

- To combat burst error, 16 sectors are interleaved together
  (16 sectors * 12 rows/sector = 192 rows)
- Error correction byes are concatenated
  - 10bytes at the end of each row
  - 16 rows at the end of the block



$$\text{payload/block} = \frac{172 \times 192}{182 \times 208} \times 100 = 87\ \%$$

DVD Video Features

- Over 2 hours of high-quality digital video
- Support wide screen movies & standard or widescreen TVs (4:3 & 16:9 aspect ratios)
- Up to 8 tracks of digital audio
- Up to 32 subtitle/karaoke tracks
- Up to 9 camera angles
- Multilingual identifying text for title name, album name, song name, actors, etc.

DVD Video Encoding Data

- Encoding Video
  - MPEG-2 compression
    (developed by the Motion Pictures Experts Group)
  - High-Resolution (better than CD,LD
    3-times better than Video tape)
- Encoding Sound
  - Dolby Digital surround AC-3 sound compression
    (support five sound channel plus subwoofer channel
    => left, center, right, rear-left, rear-right channel)

5 Buffer Management

Content

- A journey of a byte
- Buffer Management

A journey of a byte

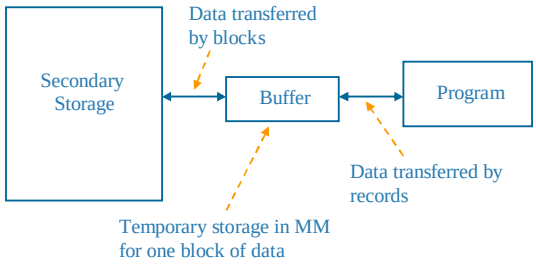
- Suppose in our program we wrote:
  ```
  outfile << c;
  ```
- This causes a call to the **file manager** (a part of O.S. responsible for I/O operations)
- The O/S (File manager) makes sure that the byte is written to the disk.
- Pieces of software/hardware involved in I/O:
  - Application Program
  - Operating System/ file manager
  - I/O Processor
  - Disk Controller

- **Application program**
  - Requests the I/O operation
- **Operating system / file manager**
  - Keeps tables for all opened files
  - Brings appropriate sector to buffer.
  - Writes byte to buffer
  - Gives instruction to I/O processor to write data from this buffer into correct place in disk.
  - Note: the buffer is an exact image of a cluster in disk.
- **I/O Processor**
  - a separate chip; runs independently of CPU
  - Find a time when drive is available to receive data and put data in proper format for the disk
  - Sends data to disk controller
- **Disk controller**
  - A separate chip; instructs the drive to move R/W head
  - Sends the byte to the surface when the proper sector comes under R/W head.

Buffer Management

- Buffering means working with large chunks of data in main memory so the number of accesses to secondary storage is reduced.
- Today, we'll discuss the System I/O buffers. These are beyond the control of application programs and are manipulated by the O.S.
- Note that the application program may implement its own "buffer" – i.e. a place in memory (variable, object) that accumulates large chunks of data to be later written to disk as a chunk.

System I/O Buffer



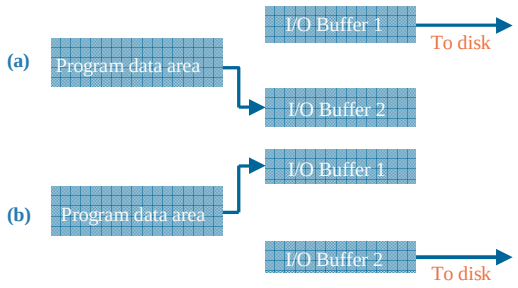
Buffer Bottlenecks

- Consider the following program segment:
  ```
  while (1) {
    infile >> ch;
     if (infile.fail()) break;
     outfile << ch;
  }
  ```
- What happens if the O.S. used only one I/O buffer?
  - ⇒ Buffer bottleneck
- Most O.S. have an input buffer and an output buffer.

Buffering Strategies

- **Double Buffering:** Two buffers can be used to allow processing and I/O to overlap.
  - Suppose that a program is only writing to a disk.
  - CPU wants to fill a buffer at the same time that I/O is being performed.
  - If two buffers are used and I/O-CPU overlapping is permitted, CPU can be filling one buffer while the other buffer is being transmitted to disk.
  - When both tasks are finished, the roles of the buffers can be exchanged.
- The actual management is done by the O.S.

Double Buffering



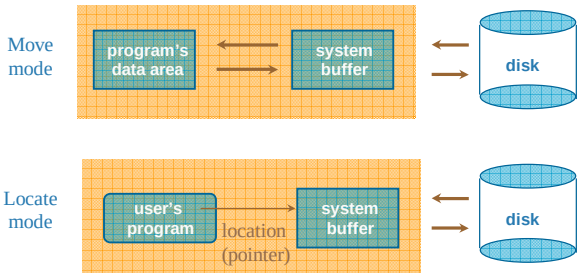
Other Buffering Strategies

- Multiple Buffering: instead of two buffers any number of buffers can be used to allow processing and I/O to overlap.
- Buffer pooling:
  - There is a pool of buffers.
  - When a request for a sector is received, O.S. first looks to see that sector is in some buffer.
  - If not there, it brings the sector to some free buffer. If no free buffer exists, it must choose an occupied buffer. (usually LRU strategy is used)

Buffering Strategies: Move & Locate mode

- Move mode (using both system buffer & program buffer)
  - moving data from one place in RAM to another before they can be accessed
  - sometimes, unnecessary data moves
- Locate mode (using system buffer only or program buffer only)
  - perform I/O directly between secondary storage and program buffer (program's data area)
  - system buffers handle all I/Os, but program uses *locations* through pointer variable

Move Mode and Location Mode



6 File Compression

Content

- Introduction to Compression
- Methods in Data Compression
  - Run-Length Coding
  - Huffman Coding

Data Compression

- Reasons for data compression
  - less storage
  - transmitting faster, decreasing access time
  - processing faster sequentially

File Compression 6

Data Compression

- Fixed-Length fields are good candidates
- Decrease the number of bits by finding a more compact notation
- *Cons.*
  - unreadable by human
  - cost in encoding time
  - decoding modules ⇒ increase the complexity of s/w
    ⇒ used for particular application

File Compression 6

Suppressing repeating sequences

- Run-length encoding algorithm
  - read through pixels, copying pixel values to file in sequence, except the same pixel value occurs more than once in succession
  - when the same value occurs more than once in succession, substitute the following three bytes
    - ✓special run-length code indicator(e.g. 0xFF)
    - ✓pixel value repeated
    - ✓the number of times that value is repeated
    - Example:
    - 22 23 *24 24 24 24 24 24* 25 26 26 26 26 26 26 25 24

      RL-coded stream: 22 23 *ff 24 07* 25 ff 26 06 25 24

File Compression 6

Suppressing Repeating Sequences

- Run-length encoding (cont'd)
  - example of redundancy reduction
  - cons.
    - not guarantee any particular amount of space savings
    - under some circumstances, *compressed image* is larger than original image
      - Why? Can you prevent this?

File Compression 6

Assigning Variable-Length Codes

- *Morse code*: oldest & most common scheme of variable-length code
- Some values occur more frequently than others
  - that value should take the least amount of space
- *Huffman coding*
  - base on probability of occurrence
    - determine probabilities of each value occurring
    - build binary tree with search path for each value
    - more frequently occurring values are given shorter search paths in tree

File Compression 6

Assigning Variable-Length Codes

- *Huffman coding*

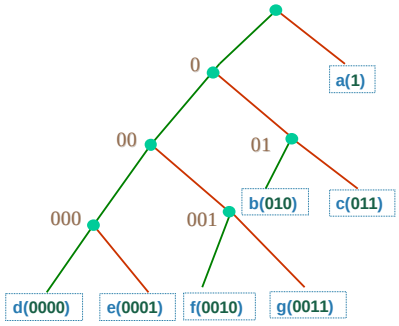
Letter:	*a*	*b*	*c*	*d*	*e*	*f*	*g*
Pr:	*0.4*	*0.1*	*0.1*	*0.1*	*0.1*	*0.1*	*0.1*
Code:	*1*	*010*	*011*	*0000*	*0001*	*0010*	*0011*

Example: the string "abde"

➔1010000000001

File Compression 6

Huffman Tree



Lempel-Ziv Codes

- There are several variations of Lempel-Ziv Codes.
- We will look at LZ78
- Commands **zip** and **unzip** and Unix **compress** and **uncompress** us Lempel-Ziv codes

Example

- Let us look at an example for an alphabet having only two letters:

aaababbbaaabaaaaaaabaabb

- Rule
  - Separate this stream of characters into pieces of text so that each piece is the shortest string of characters that we have not seen yet.

a|aa|b|ab|bb|aaa|ba|aaaa|aab|aabb

1. We see “a”
2. “a” has been seen, we now see “aa”
3. We see “b”
4. “a” has been seen, we now see “ab”
5. “b” has been seen, we now see “bb”
6. “aa” has been seen, we now see “aaa”
7. “b” has been seen, we now see “ba”
8. “aaa” has been seen, we now see “aaaa”
9. “aa” has been seen, we now see “aab”
10. “aab” has been seen, we now see “aabb”

Index

- We have index values from 1 to *n*
- For the previous example

1 2 3 4 5 6 7 8 9 10

a|aa|b|ab|bb|aaa|ba|aaaa|aab|aabb

- Encoding

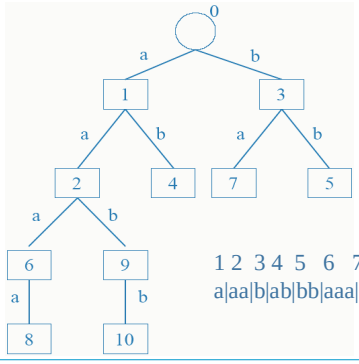
1 2 3 4 5 6 7 8 9 10

0a|1a|0b|1b|3b|2a|3a|6a|2b|9b

Lempel-Ziv Codes

- Since each piece is the concatenation of a piece already seen with a new character, the message can be encoded by a previous index plus a new character.
- A tree can be built when encoding

Encoding Tree



1 2 3 4 5 6 7 8 9 10
a|aa|b|ab|bb|aaa|ba|aaaa|aab|aabb

Exercise # 1

- encode the file containing the following characters, drawing the corresponding digital tree

"aaabbcbcdddeab"

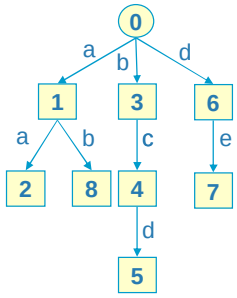
Solution

1 2 3 4 5 6 7 8

a|aa|b|bc|bcd|d|de|ab

0a|1a|0b|2c|4d|0d|6e|1b

Encoding Tree



1 2 3 4 5 6 7 8
a|aa|b|bc|bcd|d|de|ab
0a|1a|0b|2c|4d|0d|6e|1b

Exercise # 2

- Encode the file containing the following characters, drawing the corresponding digital tree

"I AM SAM. SAM I AM"

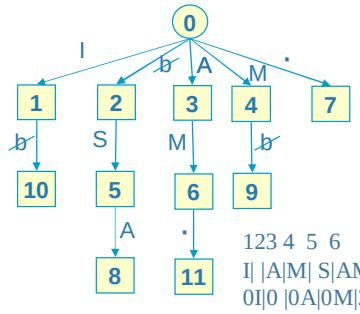
Solution

1 2 3 4 5 6 7 8 9 10 11

I| |A|M| S|AM|.| SA|M |I |AM.

0I|0 |0A|0M|2S|3M|0.|5A|4 |1 |6.

Encoding Tree



123 4 5 6 78 9 10 11
I| |A|M| S|AM|.| SA|M |I |AM.
0I|0 |0A|0M|2S|3M|0.|5A|4 |1 |6.

Lossy Compression Techniques

- Some information can be sacrificed
- Less common in data files
- Shrinking raster image
  - 400-by-400 pixels to 100-by-100 pixels
  - 1 pixel for every 16 pixels
- Speech compression
  - *voice coding (the lost information is of no little or no value)*

7 Reclaiming Spaces in Files

Motivation

- Let us consider a file of records (fixed length or variable length)
- We know how to create a file, how to add records to a file, modify the content of a record. These actions can be performed physically by using the various basic file operations we have seen (fopen, fclose, fseek, fread, fwrite)
- What happens if records need to be deleted?
- There is no basic operation that allows us to remove part of a file. Record deletion should be taken care by the program responsible for file organization

Strategies for Record Deletion

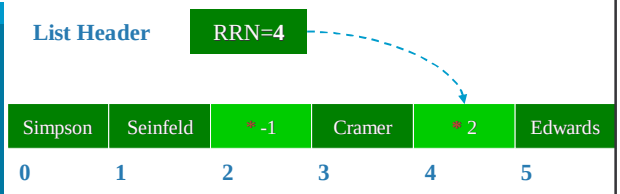
- How to delete records and reuse the unused space?

1. **Record Deletion and Storage Compaction**
   - Deletion can be done by marking a record as deleted
   - Note that the space for the record is not released, but the program that manipulates the file must include logic that checks if record is deleted or not.
   - After a lot of records have been deleted, a special program is used to squeeze the file-that is called **Storage Compaction**

Strategies for Record Deletion

2. **Deleting Fixed-Length Records and Reclaiming Space Dynamically**
   - How to use the space of deleted records for storing records that are added later?
   - Use an “AVAIL LIST”, a linked list of available records.
   - A header record stores the beginning of the AVAIL LIST
   - When a record is deleted, it is marked as deleted and inserted into the AVAIL LIST. The record space is in the same position as before, but it is logically placed into AVAIL LIST

Example

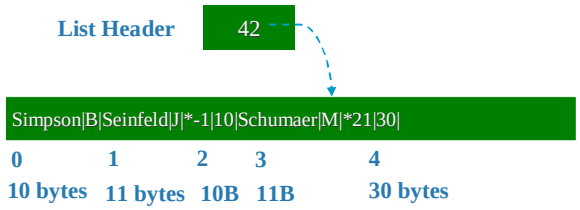


If we add a record, it can go to the first available spot in the AVAIL LIST where RRN=4.

Strategies for Record Deletion

3. **Deleting Variable-Length Records**
   - Use an AVAIL LIST as before, but take care of the variable-length difficulties
   - The records in AVAIL LIST must store its size as a field.
   - RRN can not be used, but exact byte offset must be used
   - Addition of records must find a large enough record in AVAIL LIST.

Example



Addition of records must find a large enough record in AVAIL LIST.

Placement Strategies for New Records

- There are several strategies for selecting a record from AVAIL LIST when adding a new record:

1. **First-Fit Strategy**
   - AVAIL LIST is not sorted by size.
   - First record large enough to hold new record is chosen.
- Example:
   - AVAIL LIST: size=10,size=50,size=22,size=60
   - record to be added: size=20
   - Which record from AVAIL LIST is used for the new record?

Placement Strategies for New Records

2. **Best-Fit Strategy**
   - AVAIL LIST is sorted by size.
   - Smallest record large enough to hold new record is chosen.
- Example:
   - AVAIL LIST: size=10,size=22,size=50,size=60
   - record to be added: size=20
   - Which record from AVAIL LIST is used for the new record?

Placement Strategies for New Records

3. **Worst-Fit Strategy**
   - AVAIL LIST is sorted by decreasing order of size.
   - Largest record is used for holding new record; unused space is placed again in AVAIL LIST.
- Example:
   - AVAIL LIST: size=60,size=50,size=22,size=10
   - record to be added: size=20
   - Which record from AVAIL LIST is used for the new record?

How to Choose Between Strategies

- We must consider two types of fragmentation within a file:
- **Internal Fragmentation**
  - wasted space within a record.
- **External Fragmentation**
  - space is available at AVAIL LIST, but it is so small that cannot be reused.

Study this!

- For each of the following approaches, which type of fragmentation arises, and which placement strategy is more suitable?
- When the added record is smaller than the item taken from AVAIL LIST:
- Leave the space unused within record
  - type of fragmentation: internal
  - suitable placement strategy: best-fit
- Return the unused space as a new available record to AVAIL LIST
  - type of fragmentation: external
  - suitable placement strategy: worst-fit

Ways of Combating External Fragmentation

- Coalescing the Holes
  - if two records in AVAIL LIST are adjacent, combine them into a larger record
- Minimize fragmentation by using one of the previously mentioned placement strategies
  - for example: worst-fit strategy is better than best-fit strategy in terms of external fragmentation when unused space is returned to AVAIL LIST

8 BINARY SEARCHING, KEYSORTING & INDEXING

Content

- Binary Searching
- Keysorting
- Introduction to Indexing

Binary Searching

- Let us consider fixed-length records that must be searched by a **key value**
- If we knew the RRN of the record identified by this key value, we could jump directly to the record (by using fseek function)
- In practice, we do not have this information and we must search for the record containing this key value
- If the file is not sorted by the key value we may have to look at every possible record before we find the desired record
- An alternative to this is to maintain the file sorted by **key value** and use **binary searching**

Binary Search Algorithm in C++

```
template <typename KeyType,typename RecordType>
bool BinarySearch(FILE *file,RecordType &rec, KeyType &key){
   int low=0,high=getFileLength(file)/sizeof(RecordType)-1 ;
   int guess ;
   while (low<=high){
      guess = (high+low)/2 ;
      readRecord(file,rec,guess) ;
      if (Equal (rec.key(),key)) return true ;
      if (Greater (rec.key(),key)) high = guess-1 ;
      else low = guess+1 ;
   }
   return false;
}
```


TBook

```
typedef struct TBook {
   char author[16] ;
   char title[24] ;
   char isbn[10] ;
   char *key(){return isbn;}
} SBook ;
```


Equal()

```
template <typename KeyType>
bool Equal(KeyType key1,KeyType key2){
   if (key1==key2) return true ;
   return false ;
}

bool Equal(char *key1,char *key2){
   return (strcmp(key1,key2)==0) ;
}
```


readRecord

```
template <typename RecordType>
void readRecord(FILE *file,RecordType &rec,int rnn){
   fseek(hFile,rnn*sizeof(RecordType),SEEK_SET) ;
   fread(&rec,sizeof(RecordType),1,hFile) ;
}
```


int main()

```
int main(int argc, char* argv[]) {
   FILE *hFile ;
   ...
   TBook book ;
   BinarySearch(hFile,book, "Da Vinci Code") ;
   cout << book.author ;
   return 0;
}
```

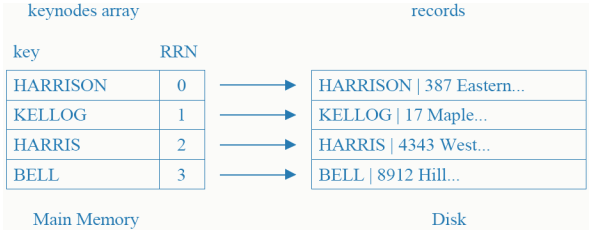

Binary Search *vs.* Sequential Search

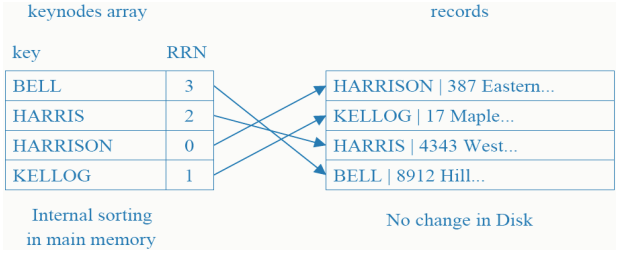
- Sequential Search: $O(n)$
- Binary Search: $O(\log_2 n)$
- If file size is doubled, sequential search time is doubled, while binary search time increases by 1

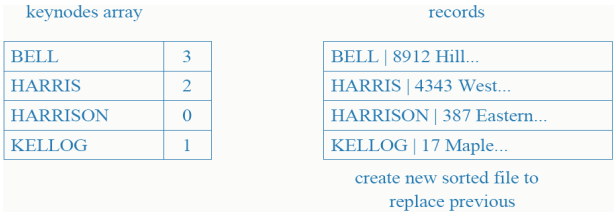
Keysorting

- Suppose a file needs to be sorted, but it is too big to fit into main memory.
- To sort the file, we only need the keys.
- Suppose that all the keys fit into main memory
- Idea
  - Bring the keys to main memory plus corresponding RRN
  - Do internal sorting of keys
  - Rewrite the file in sorted order

Example





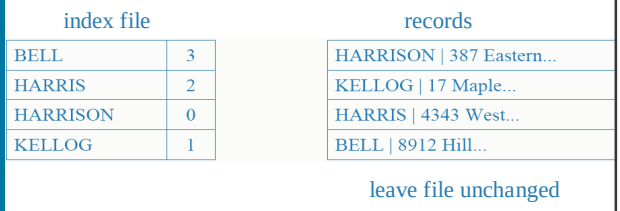


How much effort we must do?

- Read file sequentially once
- Go through each record in random order (seek)
- Write each record once (sequentially)

Why bother to write the file back?

- Use keynode array to create an index file instead.



this is called indexing!

Pinned Records

- Remember that in order to support deletions we used **AVAIL LIST**, a list of available records
- The **AVAIL LIST** contains info on the physical information of records. In such a file, a record is said to be **pinned**
- If we use an **index file** for sorting, the **AVAIL LIST** and positions of records remain unchanged.
- This is a good news ☺

Introduction to Indexing

- Simple indexes use simple arrays.
- An index lets us **impose order on a file** without rearranging the file.
- Indexes provide **multiple access paths** to a file – **multiple indexes** (like library catalog providing search for author, book and title)
- An index can provide keyed access to variable-length record files

A Simple Index for Entry-Sequenced File

- Records (Variable-length)

address of record

17	LON | 2312 | Symphony N.S | ...
62	RCA | 2626 | Quartet in C sharp | ...
117	WAR | 23699 | Adagio | ...
152	ANG | 3795 | Violin Concerto | ...

- Primary key = company label + record ID

index:

key	reference field
ANG3795	152
LON2312	17
RCA2626	62
WAR23699	117

Index

- Index is sorted (main memory)
- Records appear in file in the order they entered
- How to search for a recording with given LABEL ID?
  - Binary search (in main memory) in the index: find LABEL ID, which leads us to the referenced field
  - Seek for record in position given by the reference field

Some Issues

- How to make a persistent index
  - i.e. how to store the index into a file when it is not in main memory
- How to guarantee that the index is an accurate reflection of the contents of the file
  - This is tricky when there are lots of additions, deletions and updates

9 INDEXING

Indexing

- Operations in order to maintain an Indexed File
  1. Create the original empty and data files.
  2. Load the index file into memory before using it.
  3. Rewrite the index file from memory after using it.
  4. Add data records to the data file.
  5. Delete records from the data file.
  6. Update the index to reflect changes in the data file

Rewrite the Index File From Memory

- When the data file is closed, the index in memory needs to be written to the index file.
- An important issue to consider is what happens if the rewriting does not take place (power failures, turning the machine off, etc.)

Two Important Safeguards

- Keep an status flag stored in the header of the index file.
  - The status flag is "on" whenever the index file is not up-to-date.
  - When changes are performed in the index residing on main memory the status flag in the file is turned on.
  - Whenever the file is written from main memory the status flag is turned off.
- If the program detects the is index is out-of-date it calls a procedure that reconstruct the index from the data file

Record Addition

- This consists of appending the data file and inserting a new record in the index.
- The rearrangement of the index consists of "sliding down" the records with keys larger than the inserted key and then placing the new record in the opened space.
- Note that this rearrangement is done in main memory

Record Deletion

- This should use the techniques for reclaiming space in files when deleting from the data file
- We must delete the corresponding entry from the index:
  - Shift all records with keys larger than the key of the deleted record to the previous position (in main memory); or
  - Mark the index entry as deleted

Record Updating

- There are two cases to consider:
- The update changes the value of the key field:
  - Treat this as a deletion followed by an insertion
- The update does not affect the key field
  - If record size is unchanged, just modify the data record.
  - If record size changes treat this as a delete/insert sequence.

Indexes too Large to Fit into Main Memory

- The indexes that we have considered before could fit into main memory.
- If this is not the case, we have the following problems:
  - Binary searching of the index file is done on disk, involving several "fseek()" calls
  - Index rearrangement (record addition or deletion) requires shifting on disk

Two Main Alternatives

- Tree-structured index such as B-trees and B+ trees (Chapters 11-12)
- Hashed Organization (Chapters 13,14)

A Simple Index is still Useful, even in SSD

- It allows binary search to obtain a keyed access to a record in a variable-length record file.
- Sorting and maintaining an index is less costly than sorting and maintaining the data file, since the index is smaller
- We can rearrange keys, without moving the data records when there are pinned records

Indexing to Provide Access by Multiple Keys

- Suppose that you are looking at a collection of recordings with the following information about each of them:
  - Identification Number
  - Title
  - Composer or Composers
  - Artist or Artists
  - Label (publisher)

Data File

Address of Record	Actual data record
32	LON|2312|Romeo and Juliet|Prokofiev . . .
77	RCA|2626|Quarter in C Sharp Minor . . .
132	WAR|23699|Touchstone|Corea . . .
167	ANG|3795|Sympony No. 9|Beethoven . . .
211	COL|38358|Nebeaska|Springsteen . . .
256	DG|18807|Symphony No. 9|Beethoven . . .
300	MER|75016|Coq d'or Suite|Rimsky . . .
353	COL|31809|Symphony No. 9|Dvorak . . .
396	DG|139201|Violin Concerto|Beethoven . . .
442	FFI|245|Good News|Sweet Honey In The . . .

Indexing to Provide Access by Multiple Keys

- So far, our index only allows key access. i.e., you can retrieve record DG188807, but you cannot retrieve a recording of Beethoven's Symphony no. 9.
- We need to use secondary key fields consisting of album titles, composers, and artists.
- Although it would be possible to relate a secondary key to an actual byte offset, this is usually not done.
- Instead, we relate the secondary key to a primary key which then will point to the actual byte offset.

Example: Composer Index

- Composer Index

Secondary key	Primary key
Beethoven	ANG3795
Beethoven	DG139201
Beethoven	DG18807
Beethoven	RCA2626
Corea	WAR23699
Dvorak	COL31809
Prokofiev	LON2312

Record Addition

- When adding a record, an entry must also be added to the secondary key index.
- Store the field in Canonical Form
- There may be duplicates in secondary keys. Keep duplicates in sorted order of primary key

Record Deletion

- Deleting a record implies removing all the references to the record in the primary index and in all the secondary indexes.
- This is too much rearrangement, specially if indexes cannot fit into main memory

An Alternative to Record Deletion

- Delete the record from the data file and the primary index file reference to it. Do not modify the secondary index files.
- When accessing the file through a secondary key, the primary index file will be checked and a deleted record can be identified.
- This results in a lot of saving when there are many secondary keys
- The deleted record still occupy space in the secondary key indexes.
- If a lot of deletions occur, we can periodically cleanup these deleted records also from the secondary key indexes

Record Updating

- There are three types of updates

1. The update changes the secondary key
   - We have to rearrange the secondary key index to stay in sorted order.
2. The update changes the primary key
   - Update and reorder the primary key index
   - Update the references to primary key index in the secondary key indexes (it may involve some re-ordering of secondary indexes if secondary key occurs repeated in the file)

Record Updating (Con't)

3. Update confined to other fields
   - This will not affect secondary key indexes.
   - The primary key index may be affected if the location of record changes in the data file.

Retrieving Records using Combinations of Secondary Keys

- Secondary key indexes are useful in allowing the following kinds of queries:
  - Find all records with composer "BEETHOVEN"
  - Find all records with the title "Violin Concerto"
  - Find all records with composer "BEETHOVEN" and title "Symphony No.9"

Solution

- Use the matched list and primary key index to retrieve the two records from the file.

Matches from composer index	Matches from title index	Matched list (logical "and")
ANG3795	ANG3795	ANG3795
DG139201	COL31809	DG18807
DG18807	DG18807	
RCA2626		

Improving the Secondary Index Structure: Inverted Lists

- Two difficulties found in the proposed secondary index structures:
  - We have to rearrange the secondary index file even if the new record to be added in for an existing secondary key
  - If there are duplicates of secondary keys then the key field is repeated for each entry, wasting space

Array of References

- No need to rearrange
- Limited reference array
- Internal fragmentation

Revised composer index

Secondary key	Set of primary key references			
BEETHOVEN	ANG3795	DG139201	DG18807	RCA2626
COREA	WAR23699			
DVORAK	COL31809			
PROKOFIEV	LON2312			
RIMSKY-KORSAKOV	MER75016			
SPRINGSTEEN	COL38358			
SWEET HONEY IN THE R	FF245			

Inverted Lists

- Organize the secondary key index as an index containing one entry for each key and a pointer to a linked list of references.

Secondary Key Index File

0	Beethoven	3
1	Corea	2
2	Dvorak	5
3	Prokofiev	7

LABEL ID List File

0	LON2312	-1
1	RCA2626	-1
2	WAR23699	-1
3	ANG3795	6
4	DG18807	1
5	COL31809	-1
6	DG139201	4
7	ANG36193	0

- Beethoven is a secondary key that appears in records identified by the LABEL IDs: ANG3795, DG139201, DG18807 and RCA2626

Advantages

- Rearrangement of the secondary key index file is only done when a new composer's name is added or an existing composer's name is changed. Deleting or adding records for a composer only affects the LABEL ID List File. Deleting all records by a composer can be done by placing a "-1" in the reference field in the secondary index file.
- Rearrangement of the secondary index file is quicker since it is smaller
- Smaller need for rearrangement causes a smaller penalty associated with keeping the secondary index file in disk

Advantages (Con't)

- The LABEL ID List File never needs to be sorted since it is entry sequenced.
- We can easily reuse space from deleted records from the LABEL ID List File since its records have fixed-length.

Disadvantages

- Lost of "locality": labels of recordings with same secondary key are not contiguous in the LABEL ID List File (seeking).
- To improve this, keep the LABEL ID List File in main memory

Selective Indexes

- Selective Index: Index on a subset of records
- Selective index contains only some part of entire index
  - provide a selective view
  - useful when contents of a file fall into several categories
    - e.g. *20 < Age < 30 and $1000 < Salary*

Binding

- In our example of indexes, when does the binding of the index to the physical location of the record happens?
  - For the primary index, binding is at the time the file is constructed.
  - For the secondary index, it is at the time the secondary index is used.

Advantages of Postponing Binding

- We need small amount of reorganization when records are added or deleted.
- It is safer approach: important changes are done in one place rather than in many places.

Disadvantages

- It results in slower access times (Binary search in secondary index + Binary search in primary index)

When to use Tight Binding/Bind-at-retrieval

- Use Tight Binding
  - When data file is nearly static (little or no adding, deleting or updating of records)
  - When rapid retrieval performance is essential.
    - Example: Data stored in CD-ROM should use tight binding
- Use Bind-at-retrieval
  - When record additions, deletions and updates occur more often

10 COSEQUENTIAL PROCESSING (SORTING LARGE FILES)

Content

- Cosequential Processing and Multiway Merge
- Sorting Large Files (External Sorting)

Cosequential Processing & Multiway Merging

- K-way merge algorithm: merge K sorted input lists to create a single sorted output list
- Adapting 2-way merge algorithm
  - Instead of naming as List1 and List2 keep an array of lists: List[1], List[2],..., List[*K*]
  - Instead of naming as item(1) and item(2) keep an array of items: item[1], item[2],..., item[*K*]

2-way Merging Eliminating Repetitions

Synchronization

- Let item[1] be the current item from list[1] and item[2] be the current item from list[2].
- ***Rules:***
  - If item[1] < item[2], get the next item from list[1].
  - If item[1] > item[2], get the next item from list[2].
  - If item[1] = item[2], output the item and get the next items from the two lists.

K-way Merging Algorithm

- An array of *K* index values corresponding to the current element in each of the *K* lists, respectively.
- Main loop of the K-Way Merge algorithm:

```
1. minItem=index of minimum item in
   item[1],item[2],...,item[K]
2. output item[minItem] to output list
3. for i=1 to K do
4.     if item[i]=item[minItem] then
5.         get next item from List[i]
```

- If there are no repeated items among different lists, lines (3)-(5) can be simplified to

  get next item from List[*minItem*]

Implementation # 1

- The K-Way Merging Algorithm just described works well if $K<8$:
- Line(1) does a sequential search on item[1], item[2], ..., item[K]
  Running time: $O(K)$
- Line(5) just replaces item[i] with newly read item
  Running time: $O(1)$

Implementation # 2

- When the number of lists is large, store current items item[1], item[2], ..., item[K] into priority queue (heap).
- Line(1) does a min operation on the heap.
  Running time: $O(1)$
- Line(5) performs a **extract-min** operation on the heap:
  Running time: $O(\log_2 K)$
- and an **insert** on the heap
  Running time: $O(\log_2 K)$

Detailed Analysis of Both Algorithm

- Let N = Number of items in output list
  M = Number of items summing up all input lists
  (Note N≤M because of possible repetitions)
- Implementation # 1
  - Line(1): $K \times N$ steps
  - Line(5): counting all executions: $M \times 1$ steps
  - Total time: $O(K \times N + M) \subseteq O(K \times N)$
- Implementation # 2
  - Line(1): $1 \times N$ steps
  - Line(5): counting all executions: $M \times 2 \times \log_2 K$ steps
  - Total time: $O(N + M \times \log_2 K) \subseteq O(M \times \log_2 K)$

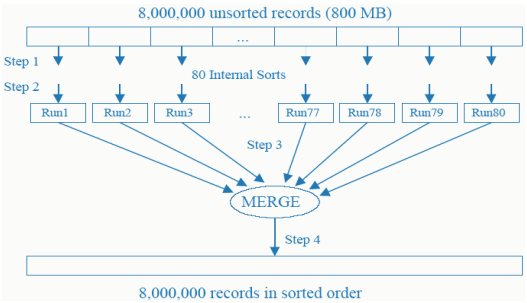
Merging as a Way of Sorting Large Files

- Characteristics of the file to be sorted
  8,000,000 records
  Size of a record = 100 Bytes
  Size of the key = 10 Bytes
- Memory available as a work area: 10 MB (Not counting memory used to hold program, OS, I/O buffers, etc.)
  Total file size = 800 MB
  Total number of bytes for all the keys = 80 MB
- So, we cannot do internal sorting

Solution

- Forming runs: bring as many records as possible to main memory, do internal sorting and save it into a small file. Repeat this procedure until we have read all the records from the original file
- Do a multiway merge of the sorted files
- In our example, what could be the size of a run?
  Available memory = 10 MB ≅ 10,000,000 bytes
  Record size = 100 bytes
  Number of records that can fit into available memory = 100,000 records
  Number of runs = 80 runs

80 Internal Sorts



Order of I/O Operations

- I/O operations are performed in the following times:

1. Reading each record into main memory for sorting and forming the runs
2. Writing sorted runs to disk

- The two steps above are done as follows:
  - Read a chunk of 10 MB; Write a chunk of 10 MB (Repeat this 80 times)
  - In terms of basic disk operations, we spend:
  - For reading: 80 seeks + transfer time for 800 MB

    Same for writing.

Order of I/O Operations (Con't)

3. Reading sorted runs into memory for merging. In order to minimize "seeks" read one chunk of each run, so 80 chunks. Since the memory available is 10 MB each chunk can have 10,000,000/80 bytes = 125,000 bytes = 1,250 records
   - How many chunks to be read for each run?
   - size of a run/size of a chunk = 10,000,000/125,000=80
   - Total number of basic "seeks" = Total number of chunks (counting all the runs) is

     80 runs × 80 chunks/run = 80^2 chunks

Order of I/O Operations (Con't)

4. When writing a sorted file to disk, the number of basic seeks depends on the size of the output buffer: bytes in file/ bytes in output buffer.
   - For example, if the output buffer contains 200 K, the number of basic seeks is 200,000,000/200,000 = 4,000

- From steps 1-4 as the number of records (*N*) grows, step 3 dominates the running time

Step 3 : The Bottleneck

- There are ways of reducing the time for the bottleneck step 3

1. Allocate more resource (e.g. disk drive, memory)
2. Perform the merge in more than one step – this reduces the order of each merge and increases the run sizes
3. Algorithmically increase the length of each run
4. Find ways to overlap I/O operations

11 B-Trees

Content

- Introduction to multilevel indexing and B-trees
- Insertion in B Trees
- `Search` and `Insert` Algorithms
- Deletion in B Trees

Introduction to Multilevel Indexing and B-Trees

- Problems with simple indexes that are kept in disk:

1. Seeking the index is still slow (binary searching):
   - We do not want more than 3/4 seeks for a search
   - So, here $\log_2(N+1)$ is still slow:

N	$\log_2(N+1)$
15 keys	4
1,000	~10
100,000	~17
1,000,000	~20

Introduction to Multilevel Indexing and B-Trees

- Problems with simple indexes that are kept in disk:

2. Insertions and deletions should be as fast as searches:
   - In simple indexes, insertion or deletion take $O(n)$ disk accesses (since index should be kept sorted)

Indexing with Binary Search Trees

- We could use balanced binary search trees:
- **AVL** Trees
  - Worst-case search is $1.44 \times \log_2(N+2)$
  - 1,000,000 keys → 29 Levels
  - Still prohibitive
- **Paged Binary** Trees
  - Place subtree of size K in a single page
  - Worst-case search is $\log_{K+1}(N+1)$
  - K=511, N=134,217,727
  - Binary trees: 27 seeks, Paged Binary tree: 3 seeks
  - This is good but there are lots of difficulties in maintaining (doing insertions and deletions) in a paged binary tree

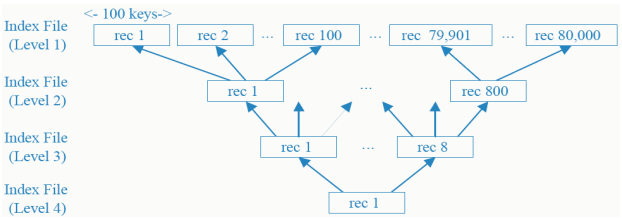
Multilevel Indexing

- Consider our 8,000,000 example with keysize =10B
- Index file size = 80 MB
- Each record in the index will contain 100 pairs (key,reference)
- A simple index would contain: 80,000 records:

  Too expensive to search (~16 seeks)

Multilevel Index

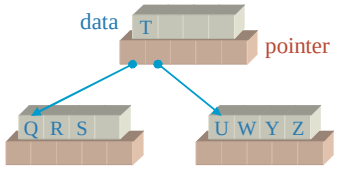
- Build an index of an index file
- How?
  - Build a simple index for the file, sorting keys using the method for external sorting previously studied
  - Build an index for this index
  - Build another index for the previous index, and so on
  - The index of an index stores the largest in the record it is pointing to.



B-Trees

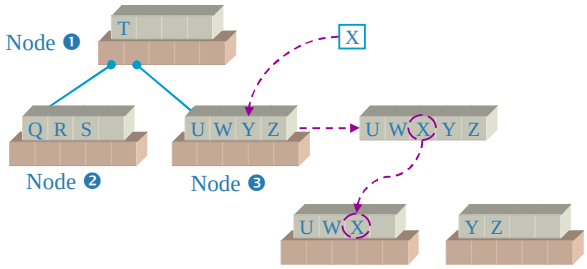
- Again an index record may contain 100 keys
- An index record may be half full (each index record may have from 50 to 100 keys)
- When insertion in an index record causes it to overfull
  - Split record in two
  - "Promote" the largest key in one of the records to the upper level

Example for Order = 4

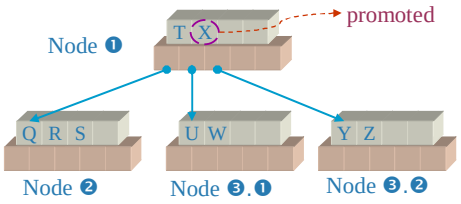


Inserting X

- X is between T and Z: insertion in node 3 splits it and generates a promotion of node X



Promotion

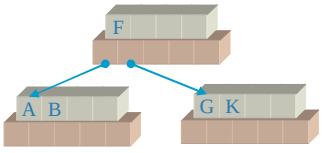


- Important: If Node 1 was full, this would generate a new split-promotion of Node 1. This could be propagated up to the root

Example of Insertions

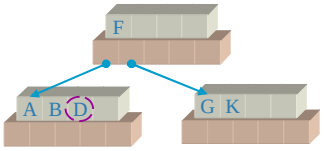
- Inserting keys: order = 4
- A,G,F,B,K,D,H,M,J,E,S,I,R,X,C,L,N,T,U,P

- Inserting K: Split and Promotion

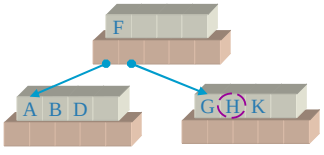


Insertion Example

- Inserting D



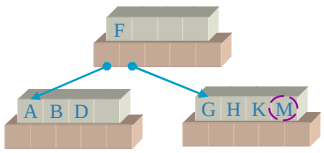
- Inserting H



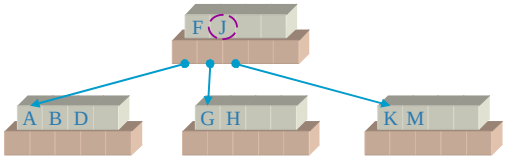
Insertion Example

B-Trees 11

▶ Inserting M



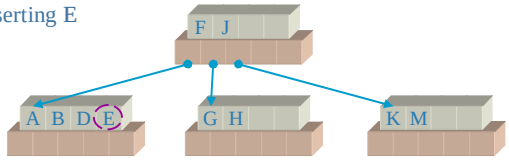
▶ Inserting J: Split and Promotion



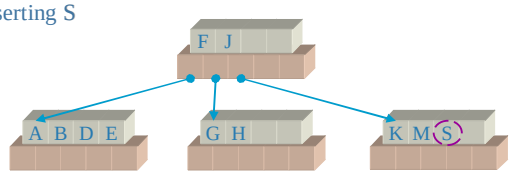
Insertion Example

B-Trees 11

▶ Inserting E



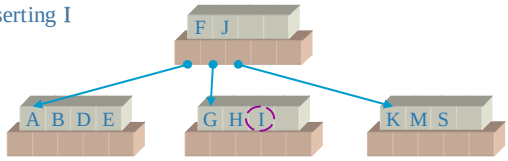
▶ Inserting S



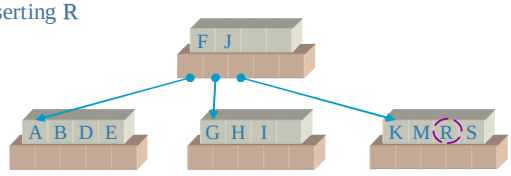
Insertion Example

B-Trees 11

▶ Inserting I



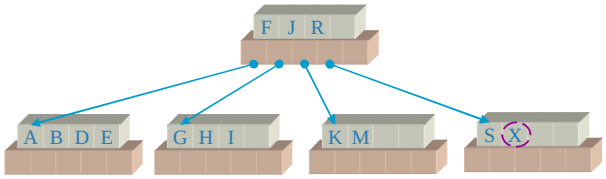
▶ Inserting R



Insertion Example

B-Trees 11

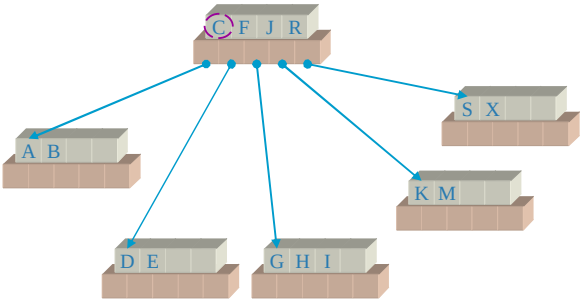
▶ Inserting X: Split and Promotion



Insertion Example

B-Trees 11

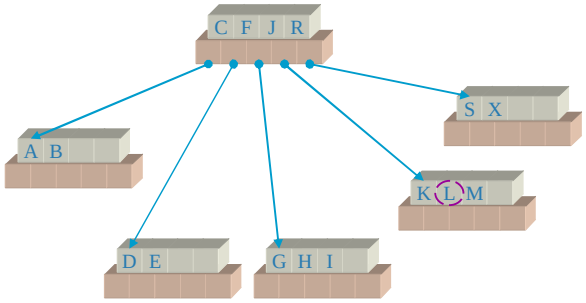
▶ Inserting C: Split and Promotion



Insertion Example

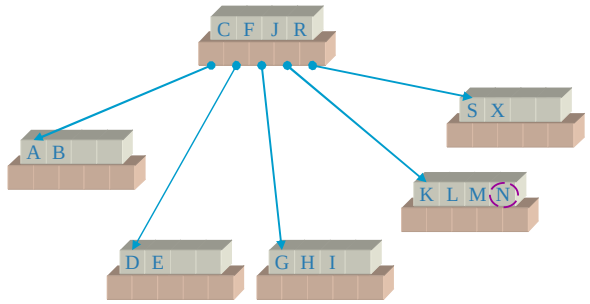
B-Trees 11

▶ Inserting L



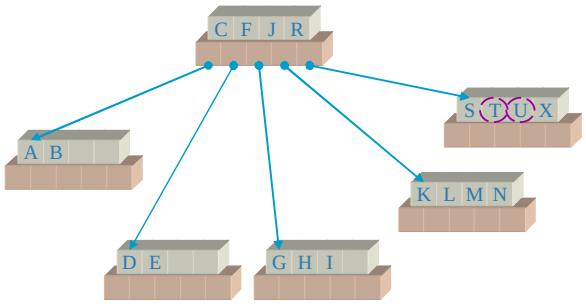
Insertion Example

- Inserting N

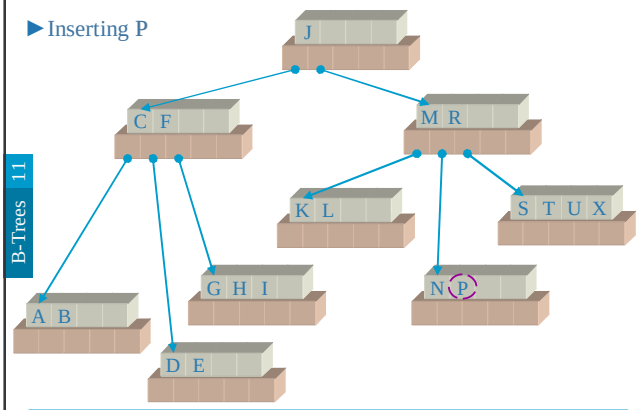


Insertion Example

- Inserting T,U



- Inserting P



B-Tree Properties

- Properties of a B-tree of order *m*:

1. Every node has a maximum of m children
2. Every node, except for the root and the leaves, has at least *m*/2 children
3. The root has at least two children (unless it is a leaf)
4. All the leafs appear on the same level
5. The leaf level forms a complete index of the associated data file

Worst-case Search Depth

- The worst-case depth occurs when every node has the minimum number of children

Level	Minimum number of keys (children)
1 (root)	2
2	$2 \cdot \lceil m/2 \rceil$
3	$2 \cdot \lceil m/2 \rceil \cdot \lceil m/2 \rceil = 2 \cdot \lceil m/2 \rceil^2$
4	$2 \cdot \lceil m/2 \rceil^3$
...	...
d	$2 \cdot \lceil m/2 \rceil^{d-1}$

Example

- Assume that we have *N* keys in the leaves

$$N \geq 2 \cdot (m/2)^{d-1}$$

So,

$$d \leq 1 + \log_{m/2}(N/2)$$

For *N*=1,000,000 and order *m*=512, we have

$$d \leq 1 + \log_{256}(N/2)$$

$$d \leq 3.37$$

- There is at most 3 levels in a B-tree of order 512 holding 1,000,000 keys.

Outline of Search Algorithm

- Search (KeyType key)

1. Find leaf: find the leaf that could contain key, loading all the nodes in the path from root to leaf into an array in main memory
2. Search for key in the leaf which was loaded into main memory

Deletions from B-Tree

- The rules for deleting a key *K* from a node *n* :

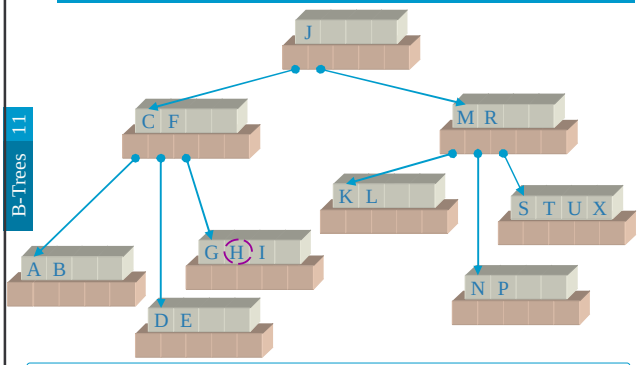
1. If *n* has more than the minimum number of keys and *K* is not the largest key in *n*, simply delete *K* from *n*.
2. If *n* has more than the minimum number of keys and *K* is the largest key in *n*, delete *K* from *n* and modify the higher level indexes to reflect the new largest key in *n*.
3. If *n* has exactly the minimum number of keys and one of the siblings has "few enough keys", **merge** *n* with its sibling and delete a key from the parent node

Deletions from B-Tree (Con't)

- The rules for deleting a key *K* from a node *n* :

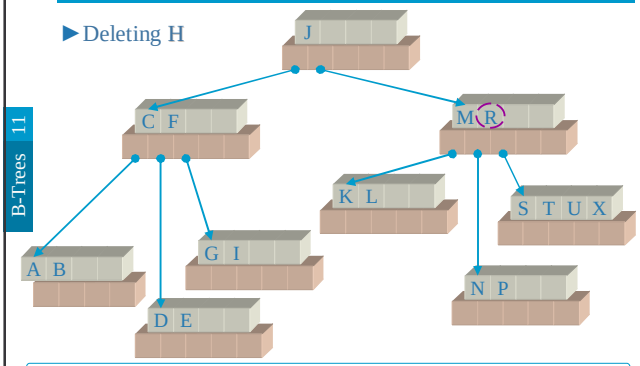
4. If *n* has exactly the minimum number of keys and one of the siblings has extra keys, **redistribute** by moving some keys from a sibling to *n*, and modify higher levels to reflect the new largest keys in the affected nodes

Example



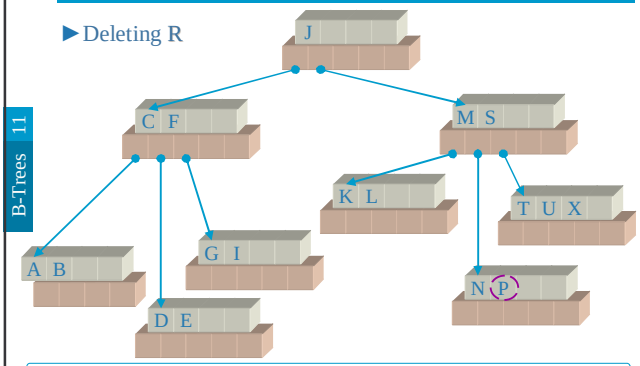
Example

- Deleting H



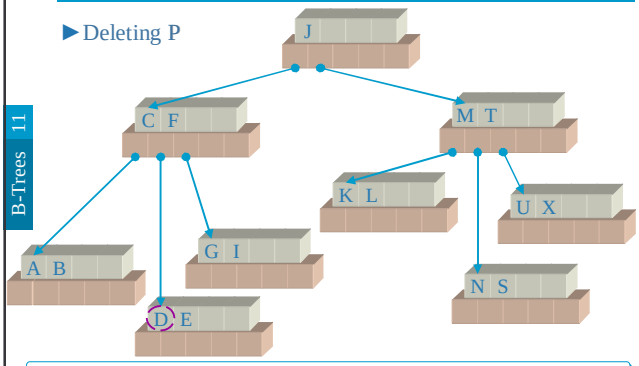
Example

- Deleting R



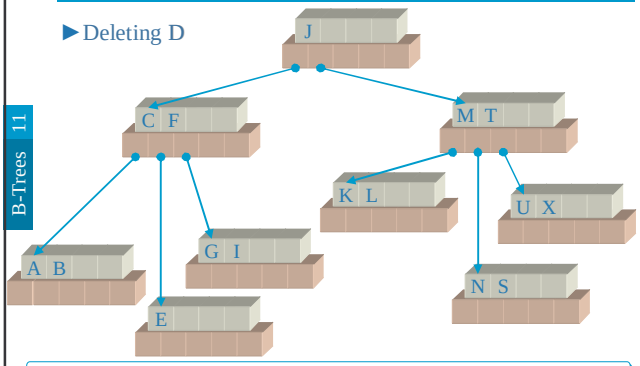
Example

▶ Deleting P

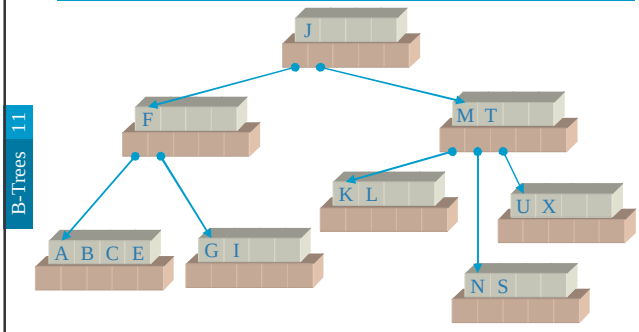


Example

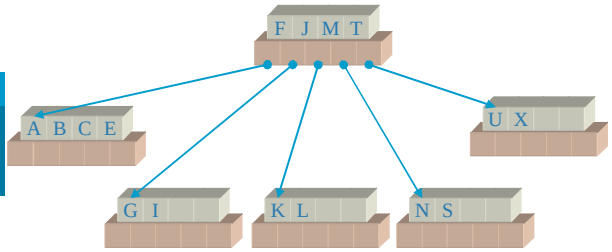
▶ Deleting D



Example

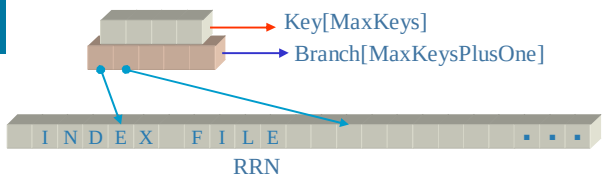


Example



Implementation

```
typedef struct {
   int Count; // Number of keys stored in the current
   ItemType Key[MaxKeys];
   long Branch[MaxKeysPlusOne];
} NodeType;
```



Implementation (Con't)

```
typedef struct {
   int Count; // Number of keys stored in the current
   ItemType Key[MaxKeys];
   long Branch[MaxKeysPlusOne];
   bool Search(KeyFieldType SearchKey,
               ItemType & Item);
} NodeType;
```

bool Search(KeyFieldType SearchKey,...)

```
{
   long CurrentRoot;
   int Location;
   bool Found;
  Found = false;
  CurrentRoot = Root;
  while ((CurrentRoot != -1L) && (!Found)) {
    fseek(hFile,CurrentRoot * NodeSize, SEEK_SET);
    fread((unsigned char *)(&CurrentNode),NodeSize,1,hFile);
    if (SearchNode(SearchKey, Location)) {
      Found = true;
      Item = CurrentNode.Key[Location];
    } else   CurrentRoot = CurrentNode.Branch[Location + 1];
  }
  return Found;
 }
```


bool SearchNode(KeyFieldType Target,...)

```
bool SearchNode(const KeyFieldType Target, int & Location) const {
  bool Found=false;
  if (strcmp(Target, CurrentNode.Key[0].KeyField)<0)  Location=-1L;
  else
  {
     Location = CurrentNode.Count - 1;
     while ((strcmp(Target, CurrentNode.Key[Location].KeyField) < 0)
              && (Location > 0))
        Location--;
        if (strcmp(Target, CurrentNode.Key[Location].KeyField) == 0)
           Found = true;
     }
  return Found;
}
```


12 B+Trees

Content

Motivation

- Some applications require two views of a file:

Indexed view :	**Sequential view :**
Records are indexed by a key	Records can be sequentially accessed in order by key
Direct, indexed access	Sequential access (physically contiguous records)
Interactive, random access	Batch processing (Ex: co-sequential processing)

Example of Applications

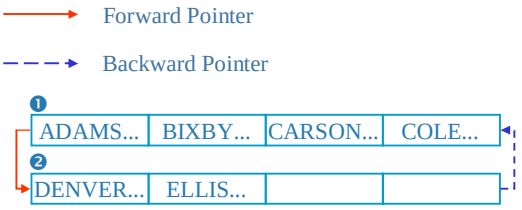
- Student record system in a university
  - Indexed view: access to individual records
  - Sequential view: batch processing when posting grades or when fees are paid
- Credit card system
  - Indexed view: interactive check of accounts
  - Sequential view: batch processing of payment slips
- We will look at the following two aspect of the problem:
  1. Maintaining a sequence set: keeping records in sequential order
  2. Adding an index set to the sequence set

Maintaining a Sequence Set

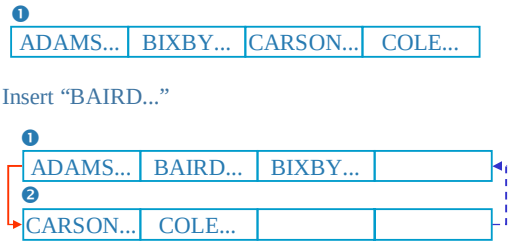
- Sorting and re-organizing after insertions and deletions is out of question
- We organize the sequence set in the following way
  - Records are grouped in **blocks**
  - Blocks should be **at least half full**
  - **Link fields** are used to point to the preceding block and the following block (similar to doubly linked list)
  - Changes (inserted/deletion) are localized into blocks by performing:
  1. **Block Splitting** when **insertion** causes overflow
  2. **Block Merging** or **Redistribution** when **deletion** causes underflow

Example

- Block Size = 4
- Key = Last Name

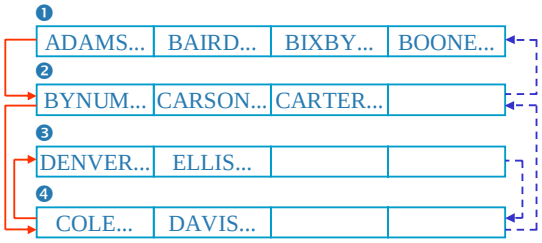


Insertion with Overflow



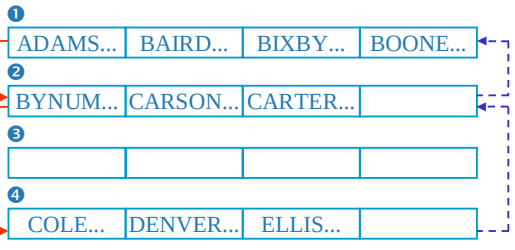
- Insert "BAIRD..."

Deletion with Merging



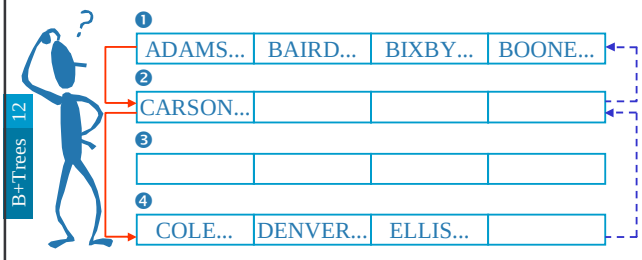
- Delete "DAVIS..."

Deletion with Merging (Con't)



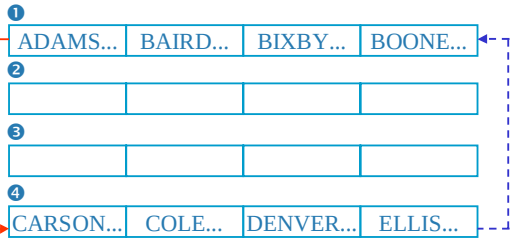
- Block ❸ is available for re-use

Delete "BYNUM", then Delete "CARTER"

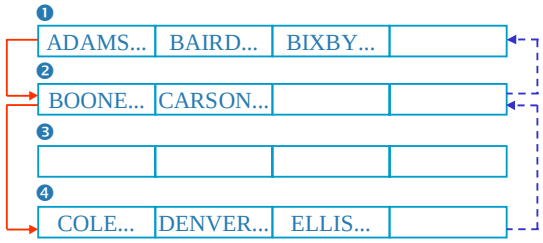


Solution # 1

- We can merge Block ❷ and ❹



Solution # 2: Deletion with Redistribution



Advantages and Disadvantages of Sequence Set

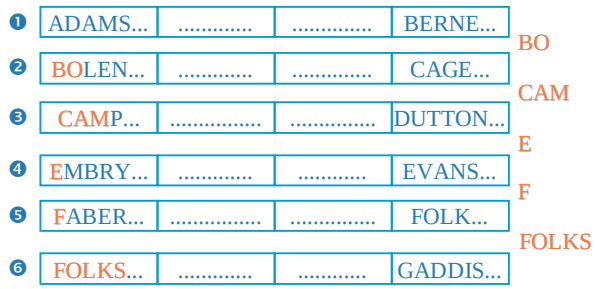
- Advantages
  - No need to re-organize the whole file after insertions/deletions
- Disadvantages
  - File takes more space than unblocked files (since blocks may be half full)
  - The order of the records is not necessarily **physically** sequential (we only guarantee physical sequentiality within a block)

Choosing Block Size

- Main memory constraints (must hold at least 2 blocks)
- Avoid seeking within a block (e.g., in sector formatted disks choose block size equal to cluster size).

Adding an Index Set to the Sequential Set

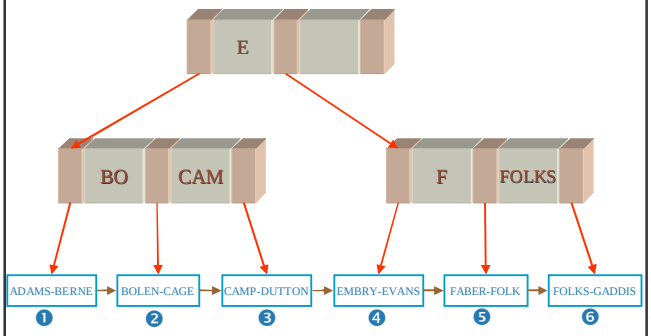
- Index will contain SEPERATORS instead of KEYS



The Simple Prefix B+ Tree

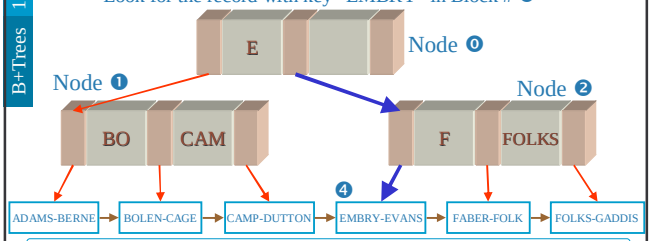
- The **simple prefix B+ tree** consists of
  - **Sequence Set**
  - **Index Set**: similar to a B-tree index, but storing the shortest separators for the sequence set.

Example: Order of the index set is 3



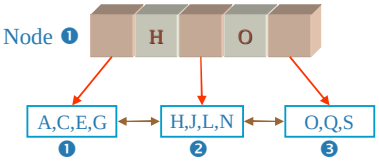
Search in a Simple Prefix B+ Tree

- Search for "EMBRY"
  - Retrieve Node ⓪ (Root)
  - Since "EMBRY"> "E", so go right, and retrieve Node ❷.
  - Since "EMBRY"< "F", so go left, and Block # ❹
  - Look for the record with key "EMBRY" in Block # ❹



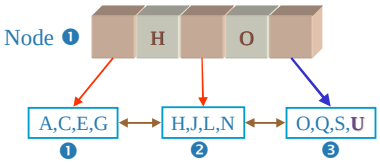
Simple Prefix B+ Tree Maintenance

- Example:
  - Sequence set has blocking factor 4
  - Index set is a B tree of order 3



Example (Cont'd)

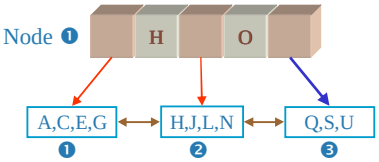
1. Changes which are local to single blocks in the sequence set
   - Insert "U"
     - Go to the root
     - Go to the right of "O"
     - Insert "U" to Block ❸



–There is no change in the index set

Example (Cont'd)

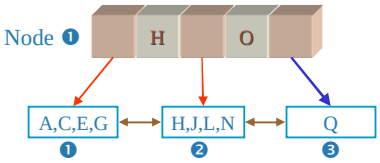
- Delete "O"
  - Go to the root
  - Go to the right of "O"
  - Delete "O" from Block ❸



–There is no change in the index set: "O" is still a perfect separator for Blocks ❷ and ❸

Example (Cont'd)

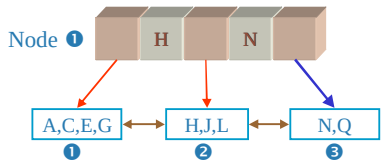
2. Changes involving multiple blocks in the sequence set
   - Delete "S" and "U"



–Now Block ❸ becomes less than ½ full (UNDERFLOW)

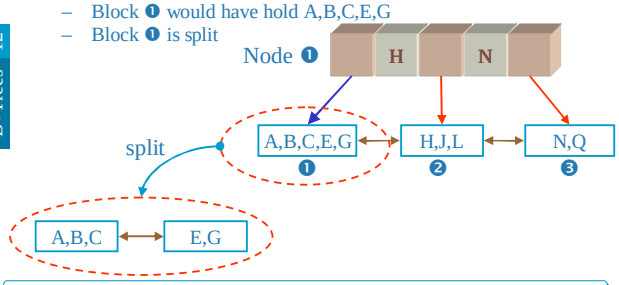
Example (Cont'd)

- Since Block ❷ is full, the only position is re-distribution bringing a key from Block ❷ to Block ❸
- We must update the separator "O" to "N"



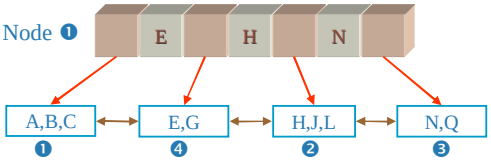
Example (Cont'd)

- Insert "B"
  - Go to the root
  - Go to the left of "H" to Block ❶
  - Block ❶ would have hold A,B,C,E,G
  - Block ❶ is split

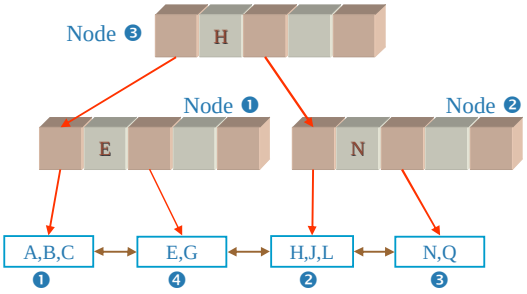


Example (Cont'd)

- So this causes Node ❶ to split

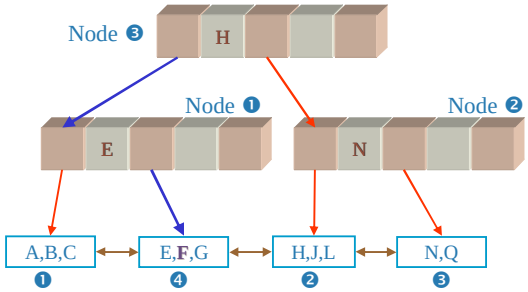


Example (Cont'd)



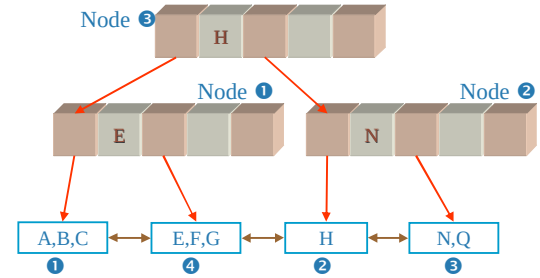
Example (Cont'd)

- Insert "F"



Example (Cont'd)

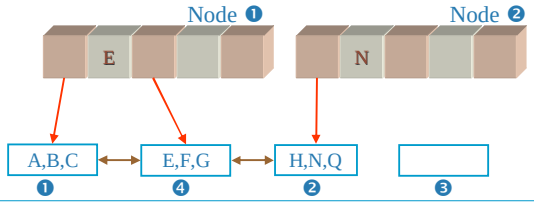
- Delete "J" and "L"



- This is an UNDERFLOW. You may think to redistribute Blocks ❷ and ❹: E,F,G,H becomes E,F and G,H.

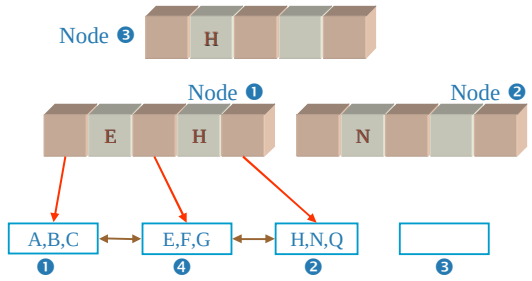
Example (Cont'd)

- Why this is not possible?
- Blocks ❷ and ❹ are not siblings! They are cousins.
- Merge Blocks ❷ and ❸
- Send Block ❸ to AVAIL LIST
- Remove the Link Between Node ❷ and Block ❸



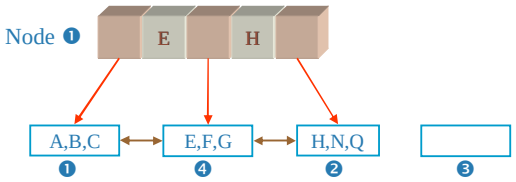
Example (Cont'd)

- Send Node ❷ and ❸ to AVAIL LIST



Example (Cont'd)

- Blocks were reunited as a big happy family again ☺



13 Hashing

Objectives

- Introduce the concept of hashing
- Examine the problem of choosing a good hashing algorithm
- Explore three approaches for reducing collisions
- Develop and use mathematical tools for analyzing performance differences resulting from the use of different hashing techniques
- Examine problems associated with file deterioration and discuss some solutions
- Examine effects of patterns of records access on performance

Content

- Introduction to Hashing
- Hash functions
- Distribution of records among addresses, synonyms and collisions
- Collision resolution by progressive overflow or linear probing

Motivation

- Hashing is a useful searching technique, which can be used for implementing indexes. The main motivation for Hashing is improving searching time.
- Below we show how the search time for Hashing compares to the one for other methods:
  - Simple Indexes (using binary search): $O(\log_2 N)$
  - B Trees and B+ trees: $O(\log_k N)$
  - Hashing: $O(1)$

What is Hashing?

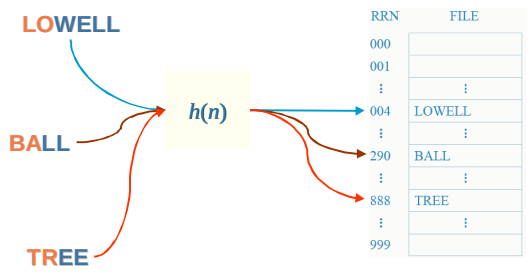
- The idea is to discover the location of a key by simply examining the key. For that we need to design a hash function.
- A Hash Function is a function $h(k)$ that transforms a key into an address
- An address space is chosen before hand. For example, we may decide the file will have 1,000 available addresses.
- If U is the set of all possible keys, the hash function is from U to {0,1,...,999}, that is

$$h : U \rightarrow \{0,1,...,999\}$$

Example

NAME	ASCII code for first two letters	PRODUCT	HOME ADDRESS
BALL	66 65	66×65=4290	290
LOWELL	76 79	76×79=6004	004
TREE	84 82	84×82=6888	888

What is Hashing?



What is Hashing?

- There is no obvious connection between the key and the location (randomizing)
- Two different keys may be sent to the same address generating a Collision
- Can you give an example of collision for the hash function in the previous example?

Answer

- LOWELL, LOCK, OLIVER, and any word with first two letters **L** and **O** will be mapped to the same address

$$h(\mathbf{LO}\text{WELL})=h(\mathbf{LO}\text{CK})=h(\mathbf{OL}\text{IVER})=\mathbf{004}$$

- These keys are called synonyms. The address "004" is said to be the home address of any of these keys.
- Avoiding collisions is extremely difficult
- Do you know the birthday paradox?
- So we need techniques for dealing with it.

Reducing Collisions

1. Spread out the records by choosing a good hash function
2. Use extra memory: increase the size of the address space
   - Example: reserve 5,000 available addresses rather than 1,000
3. Put more than one record at a single address: use of buckets

A Simple Hash Function

- To compute this hash function, apply 3 steps:
- **Step 1**: transform the key into a number.

LOWELL

L	O	W	E	L	L						

ASCII code

76	79	87	69	76	76	32	32	32	32	32	32

A Simple Hash Function (Con't)

- **Step 2**: fold and add (chop off pieces of the number and add them together) and take the mod by a prime number

76	79	87	69	76	76	32	32	32	32	32	32

7679	8769	7676	3232	3232	3232

7679+8769+7676+3232+3232+3232

33,820 mod 19937 = 13,883

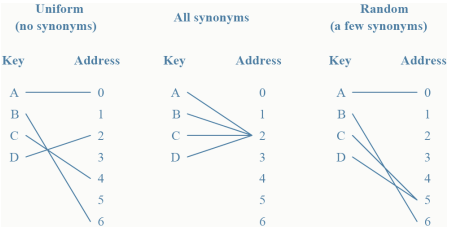
A Simple Hash Function (Con't)

- **Step 3**: divide by the size of the address space (preferably a prime number)

13,883 mod 101 = 46

Distribution of Records among Addresses

- There are 3 possibilities



- Uniform distributions are extremely rare
- Random distributions are acceptable and more easily obtainable.

Better than Random Distribution

- Examine keys for patterns
  - Example: Numerical keys that are spread out naturally such as keys are years between 1970 and 2004

    f(year)=(*year*-1970) mod (2004-1970+1)

    f(1970)=0, f(1971)=1,..., f(2004)=34
- Fold parts of the key.
  - Folding means extracting digits from a key and adding the parts together as in the previous example.
  - In some cases, this process may preserve the natural separation of keys, if there is a natural separation

Better than Random Distribution (Con't)

- Use prime number when dividing the key.
  - Dividing by a number is good when there are sequences of consecutive numbers.
  - If there are many different sequences of consecutive numbers, dividing by a number that has many small factors may result in lots of collisions. A prime number is a better choice.

Randomization

- When there is no natural separation between keys, try randomization.
- You can using the following Hash functions:

1. **Square the key and take the middle**

   Example: key=453 $453^2 = 205209$

   Extract the middle = 52.

   This address is between 00 and 99.

Randomization (Con't)

2. **Radix transformation:**

   Transform the number into another base and then divide by the maximum address

   Example: Addresses from 0 to 99

   key = 453 in base 11 = 382

   hash address = 382 mod 99 = 85.

Collision Resolution: Progressive Overflow

- Progressive overflow/linear probing works as follows:

1. **Insertion of key k**:
   - Go to the home address of k: $h(k)$
   - If free, place the key there
   - If occupied, try the next position until an empty position is found

     (the 'next' position for the last position is position 0, i.e. wrap around)

Example

key k	Home address - h(k)
COLE	20
BATES	21
ADAMS	21
DEAN	22
EVANS	20

Complete Table:

0	
1	
2	
⋮	⋮
19	
20	
21	
22	

Table size = 23

Progressive Overflow (Con't)

2. **Searching for key k**:
   - Go to the home address of k: $h(k)$
   - If k is in home address, we are done.
   - Otherwise try the next position until: key is found or empty space is found or home address is reached (in the last 2 cases, the key is not found)

Example

- A search for ‘EVANS’ probes places: 20,21,22,0,1, finding the record at position 1.
- Search for ‘MOURA’, if *h*(MOURA)=22, probes places 22,0,1,2 where it concludes ‘MOURA’ in not in the table.
- Search for ‘SMITH’, if *h*(SMITH)=19, probes 19, and concludes ‘SMITH’ in not in the table.

0	DEAN
1	EVANS
2	
⋮	⋮
19	
20	COLE
21	BATES
22	ADAMS

Advantages ✕ Disadvantages

- Advantage: Simplicity
- Disadvantage: If there are lots of collisions of records can form, as in the previous example

Search Length

- Number of accesses required to retrieve a record.

$$\textbf{average search length} = \frac{\textbf{sum of search lengths}}{\textbf{number of records}}$$

Example

0	DEAN
1	EVANS
2	
⋮	⋮
19	
20	COLE
21	BATES
22	ADAMS

key k	Home address - h(k)
COLE	20
BATES	21
ADAMS	21
DEAN	22
EVANS	20

key	Search Length
COLE	1
BATES	1
ADAMS	2
DEAN	2
EVANS	5

- Average search length (1+1+2+2+5)/5=2.2

Predicting Record Distribution

- We assume a random distribution for the hash function.
  - N = number of available addresses
  - r = number of records to be stored
- Let $p(x)$ be the probability that a given address will have x records assigned to it
- It is easy to see that

$$p(x) = \frac{r!}{(r-x)!x!}\left[1-\frac{1}{N}\right]^{r-x}\left[\frac{1}{N}\right]^{x}$$

Predicting Record Distribution (Con’t)

- For N and r large enough this can be approximated by

$$p(x) \approx \frac{(r/N)^{x} e^{-(r/N)}}{x!}$$

Example

- N=1000, r=1000

$$p(0) = \frac{(1)^0 e^{-1}}{0!} = 0.368$$

$$p(1) = \frac{(1)^1 e^{-1}}{1!} = 0.368$$

$$p(2) = \frac{(1)^2 e^{-1}}{2!} = 0.184$$

$$p(3) = \frac{(1)^3 e^{-1}}{3!} = 0.061$$

Predicting Record Distribution (Con't)

- For N addresses, the expected number of addresses with x records is

$$N \cdot p(x)$$

- N=1000, r=1000

$$1000 \times p(0) = 368$$

$$1000 \times p(1) = 368$$

$$1000 \times p(2) = 184$$

$$1000 \times p(3) = 61$$

Reducing Collision by using more Addresses

- Now, we see how to reduce collisions by increasing the number of available addresses.
- Definition: **`packing density`** = r/N
- Example:

  500 records to be spread over 1000 addresses result in **`packing density`** = 500/1000 = 0.5 = 50%

Questions

1. How many addresses go unused? More precisely: *What is the expected number of addresses with no key mapped to it*?

- $N \times p(0) = 1000 \times 0.607 = 607$

Questions (Con't)

2. How many addresses have no synonyms? More precisely: *What is the expected number of address with only one key mapped to it*?

- $N \times p(1) = 1000 \times 0.303 = 303$

Questions (Con't)

3. How many addresses contain 2 or more synonyms? More precisely: *What is the expected number of addresses with two or more keys mapped to it*?

- $N \times (p(2)+p(3)+\ldots) = N \times (1-p(0)-p(1)) = 1000 \times 0.09 = 90$

Questions (Con't)

4. Assuming that only one record can be assigned to an address. How many overflow records are expected?

   $1\times N\times p(2) + 2\times N\times p(3) + 3\times N\times p(4)+... =$

   $N\times(2\times p(2)+3\times p(4)+...) \approx 107$

- The justification for the above formula is that there is going to be (i-1) overflow records for all the table positions that have i records mapped to it, which are expected to be as many as $N\cdot p(i)$

A Simpler Formula

- Expected # of overflow records =

  #records – Expected # of non-overflow records

  $= r - (N \cdot p(1)+N \cdot p(2)+N \cdot p(3)+ \cdots)$

  $= r - (1 - p(0))$

  $= N \cdot p(0) - (N-r)$

Questions (Con't)

5. What is the expected percentage of overflow records?

   107/500 = 0.214 = 21.4%

- Note that using either formula, the percentage of overflow records depend only on the packing density ($PD = r/N$) and not on the individual values of N or r.
- The percentage of overflow records is

$$\frac{r - N\left(1 - p(0)\right)}{r} = 1 - \frac{1}{PD}\left(1 - p(0)\right)$$

- Poisson function that approximates $p(0)$ is a function of r/N which is equal to PD (for hashing without buckets).

Packing Density-Overflow Records

Packing Density %	Overflow Records %
10%	4.8%
20%	9.4%
30%	13.6%
40%	17.6%
50%	21.4%
60%	24.8%
70%	28.1%
80%	31.2%
90%	34.1%
100%	36.8%

Hashing with Buckets

- This is a variation of hashed files in which more than one record/key is stored per hash address.
- Bucket = block of records corresponding to one address in the hash table
- The hash function gives the **Bucket Address**
- Example:

Example

- For a bucket holding 3 records, insert the following keys

key	Home Address
LOYD	34
KING	33
LAND	33
MARX	33
NUTT	33
PLUM	34
REES	34

0	
⋮	⋮
33	KING
	LAND
	MARX
34	LOYD

Effects of Buckets on Performance

- We should slightly change some formulas

$$\textit{packing density} = \frac{r}{b \cdot N}$$

We will compare the following two alternatives

1. Storing 750 data records into a hashed file with 1000 addresses, each holding 1 record.
2. Storing 750 data records into a hashed file with 500 bucket addresses, each bucket holding 2 records

- In both cases the packing density is 0.75 or 75%.
- In the first case r/N=0.75.
- In the second case r/N=1.50

Effects of Buckets on Performance

- Estimating the probabilities as defined before:

	p(0)	p(1)	p(2)	p(3)	p(4)
1) r/N=0.75 (b=1)	0.472	0.354	0.133	0.033	0.006
2) r/N=1.50 (b=2)	0.223	0.335	0.251	0.126	0.047

Effects of Buckets on Performance

Calculating the number of overflow records in each case

1. b=1 (r/N=0.75):

$$\textit{Number of overflow records} =$$
$$= N \cdot \left[1 \cdot p(2) + 2 \cdot p(3) + 3 \cdot p(4) + \cdots\right]$$
$$= r - N \cdot \left(1 - p(0)\right)$$
$$= 750 - 1000 \cdot \left(1 - 0.472\right) = 750 - 528 = 222$$

This is about 29.6% overflow

Effects of Buckets on Performance

2. b=2 (r/N=1.5):

$$\textit{Number of overflow records} =$$
$$= N \cdot \left[1 \cdot p(3) + 2 \cdot p(4) + 3 \cdot p(5) + \cdots\right]$$
$$= r - N \cdot p(1) - 2 \cdot N \cdot \left[p(2) + p(3) + \cdots\right]$$
$$= r - N \cdot \left[p(1) + 2 \cdot \left(1 - p(0) - p(1)\right)\right]$$
$$= r - N \cdot \left[2 - 2 \cdot p(0) - p(1)\right]$$
$$= 750 - 500 \cdot \left(2 - 2 \cdot (0.223) - 0.335\right) = 140.5 \cong 140$$

This is about 18.7% overflow

Percentage of Collisions for Different Bucket Sizes

	Bucket Size				
Packing Density %	1	2	5	10	100
75%	29.6%	18.7%	8.6%	4.0%	0.0%

Implementation Issues

1. Bucket Structure

- A Bucket should contain a counter that keeps track of the number of records stored in it.
- Empty slots in a bucket may be marked '//.../'
- Example: Bucket of size 3 holding 2 records

2	JONES	//////////...///	ARNSWORTH

Implementation Issues

2. Initializing a file for hashing

- Decide on the Logical Size (number of available addresses) and on the number of buckets per address.
- Create a file of empty buckets before storing records. An empty bucket will look like

0	//////////..//	//////////..//	//////////..//

Implementation Issues

3. Loading a hash file

- When inserting a key, remember to:
- Be careful with infinite loops when hash file is full

Making Deletions

- Deletions in a hashed file have to be made with care

Record	ADAMS	JONES	MORRIS	SMITH
Home Address	5	6	6	5

Hashed File using Progressive Overflow

⋮	⋮
4	//////////
5	ADAMS
6	JONES
7	MORRIS
8	SMITH
⋮	⋮

Making Deletions: Delete 'MORRIS'

- If 'MORRIS' is simply erased, a search for 'SMITH' would be unsuccessful

⋮	⋮	
4	//////////	← Empty Slot
5	ADAMS	
6	JONES	
7	//////////	← Empty Slot
8	SMITH	**Problem**: you cannot find 'SMITH'
⋮	⋮	

- Search for 'SMITH' would go to home address (position 5) and when reached 7 it would conclude 'SMITH' is not in the file!

Solution

- Replace deleted records with a marker indicating that a record once lived there

⋮	⋮	
4	//////////	
5	ADAMS	
6	JONES	
7	######	← Deleted Slot
8	SMITH	you can find 'SMITH'
⋮	⋮	

- A search must continue when it finds a tombstone, but can stop whenever an empty slot is found

Be careful in Deleting and Adding a Record

- Only insert a tombstone when the next record is occupied or is a tombstone
- Insertions should be modified to work with tombstones: if either an empty slot or a tombstone is reached, place the new record there.

Effects of Deletions and Additions on Performance

- The presence of too many tombstones increases search length.
- Solutions to the problem of deteriorating average search lengths:

1. Deletion algorithm may try to move records that follow a tombstone backwards towards its home address
2. Complete reorganization: re-hashing
3. Use a different type of collision resolution technique

Other Collision Resolution Techniques

1. Double Hashing

- The first hash function determines the home address
- If the home address is occupied, apply a second hash function to get a number c (c relatively prime to N)
- c is added to the home address to produce an overflow addresses: if occupied, proceed by adding c to the overflow address, until an empty spot is found.

Example

k (key)	ADAMS	JONES	MORRIS	SMITH
$h_1(k)$ (home address)	5	6	6	5
$h_2(k) = c$	2	3	4	3

Hashed file using double hashing

2	
3	
4	
5	ADAMS
6	JONES
7	
8	SMITH
9	
10	MORRIS

A Question

- Suppose the above table is full, and that a key k has $h_1(k)$=6 and $h_2(k)$=3.
- What would be the order in which the addresses would be probed when trying to insert k?

Answer: 6, 9, 1, 4, 7, 10, 2, 5, 8, 0, 3

0	XXXXX
1	XXXXX
2	XXXXX
3	XXXXX
4	XXXXX
5	XXXXX
6	XXXXX
7	XXXXX
8	XXXXX
9	XXXXX
10	XXXXX

Other Collision Resolution Techniques (Con't)

2. Chained Progressive Overflow

- Similar to progressive overflow, except that synonyms are linked together with pointers.
- The objective is to reduce the search length for records within clusters.

Example

Key	Home	Progressive Overflow	Chained Progr. Overflow
ADAMS	20	1	1
BATES	21	1	1
COLES	20	3	2
DEAN	21	3	2
EVANS	24	1	1
FLINT	20	6	3
Average Search Length :		2.5	1.7

Example (Con't)

Progressive Overflow

	data
⋮	⋮
20	ADAMS
21	BATES
22	COLES
23	DEAN
24	EVANS
25	FLINT
⋮	⋮

Chained Progressive Overflow

	data	next
⋮	⋮	⋮
20	ADAMS	22
21	BATES	23
22	COLES	25
23	DEAN	-1
24	EVANS	-1
25	FLINT	-1
⋮	⋮	⋮

Other Collision Resolution Techniques (Con't)

3. **Chained with a Separate Overflow Area**
   - Move overflow records to a Separate Overflow Area
   - A linked list of synonyms start at their home address in the Primary data area, continuing in the separate overflow area
   - When the packing density is higher than 1 an overflow area is required

Example

Primary Data Area

20	ADAMS	0
21	BATES	1
22		
23		
24	EVANS	-1
25		

Overflow Area

0	COLES	2
1	DEAN	-1
2	FLINT	-1
3		
	⋮	⋮

Other Collision Resolution Techniques (Con't)

4. **Scatter Tables: Indexing Revisited**
   - Similar to chaining with separate overflow, but the hashed file contains no records, but only pointers to data records.

index (hashed)

⋮	
20	0
21	1
22	
23	
24	4
⋮	

datafile

	data	next
0	ADAMS	2
1	BATES	3
2	COLES	5
3	DEAN	-1
4	EVANS	-1
5	FLINT	-1

14 Extendible Hashing

Content

- What is extendible hashing?
- Insertions in extendible hashing
- Insertions a closer look at bucket splitting
- Deletions in extendible hashing
- Extendible hashing performance

What is Extendible Hashing?

- It is an approach that tries to make hashing dynamic, i.e. to allow insertions and deletions to occur without resulting in poor performance after many of these operations.
- Why this is not the case for ordinary hashing?
- Extendible hashing combines two ingredients:
  1. Hashing
  2. Tries
- Keys are placed into buckets, which are independent parts of a file in disk.
- Keys having a hashing address with the same prefix share the same bucket.
- A trie is used for fast access to the buckets. It uses a prefix of the hashing address in order to locate the desired bucket

Tries and Buckets

- Consider the following grouping of keys into buckets depending on the prefix of their hash addresses

15 Indexing Spatial Data

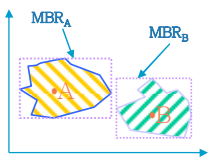
Content

Introduction

- Many applications(e.g., CAD, GIS) operate on *spatial data*, which include points, lines, polygons and so on
- Conventional DBMSs are unable to support spatial data processing efficiently
  - First, spatial data are large in quantity, complex in structures and relationships
  - Second, the retrieval process employs *complex spatial operators* like *intersection*, *adjacency*, and *containment*
  - Third, it is difficult to define a spatial ordering, so conventional techniques(e.g., sort-merge) cannot be employed for spatial operations
- Spatial indexes need!

Query Processing

- It is expensive to perform spatial operations (e.g., intersect, contain) on real spatial data
- Thus, simpler structure that *approximates* the objects are used: Minimum Bounding Rectangle or circle
- Example: intersection



- **Step1**: perform intersection operation between MBR_A and MBR_B
- **Step2**: if MBR_A intersects with MBR_B, then perform intersection operation between A and B

Query Processing (Con't)

- Multi-step spatial query processing
  1. The spatial index prunes the search space to a set of candidates
  2. Based on the approximations of candidates, some of the *false hits* can be further filtered away
  3. Finally, the actual objects are examined to identify those that match the query
  - The multi-step strategy can effectively reduce the number of pages accessed, the number of data to be fetched and tested and the computation time through the approximations
  - Types of spatial queries
    - Spatial selection: point query, range(window) query
    - Spatial join between two or more different entities sets

A Taxonomy of spatial indexes

- Classification of spatial indexes
  1. The transformation approach
     - Parameter space indexing
       - Objects with ***n*** vertices in a ***k***-dimensional space are mapped into points in a ***nk***-dimensional space
       - e.g.) two-dimensional rectangle described by the two corner (x_1,y_1) and (x_2, y_2) => a point in a four-dimensional space
     - Mapping to single attribute space
       - The data space is partitioned into grid cells of the same size, which are then numbered according to some *curve-filling methods*(e.g., hilbert curve, z-ordering, snake-curve)

A Taxonomy of spatial indexes (Con't)

- Classification of spatial indexes
  2. The non-overlapping native space indexing approach
     - Object duplication
     - Object clipping

A Taxonomy of spatial indexes (Con't)

- Classification of spatial indexes
  3. The overlapping native space indexing approach
     - Partitioning hierarchically the data space into a manageable number of smaller subspaces
     - Allowing the overlapping between bounding subspaces
     - The overlapping minimization is very important
     - e.g.)
       - binary-tree: kd-tree, LSD-tree, etc.
       - B-tree: k-d-b-tree, R-tree, R*-tree, TV-tree, X-tree, etc.
       - Hashing: Grid-files, BANG files, etc.
       - Space-Filling: Z-ordering, Filter-tree, etc.

Binary-tree based indexing

- The characteristics
  - A basic data structure for representing data items whose index values are ordered by some linear order
  - Repetitively partitioning a data space
- Types of binary search trees
  - kd-tree
  - K-D-B-tree
  - hB-tree
  - skd-tree
  - LSD-tree

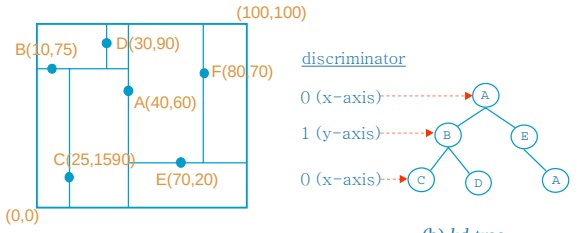
Binary-tree based indexing: The *kd*-tree

- The kd-tree
  - k-dimensional binary search tree to index multi-attribute data
  - A *node* in the tree serves both representation of a actual data point and direction of search
  - A *discriminator* is used to indicate the key on which branching decision depends
  - A node P has two children, a left son LOSON(P) and a right son HISON(P)
  - If discriminator is the j^{th} attribute, then the j^{th} attribute of any node in the LOSON(P) is less than the j^{th} attribute of node P, and the j^{th} attribute of any node in the HISON(P) is greater than or equal to that of node P

Binary-tree based indexing: The *kd*-tree (con't)

- The kd-tree
  - Complications arise when an internal node(Q) is deleted
    - One of the nodes in the subtree whose root is Q must replace Q
    - To reduce the cost of deletion, a non-homogeneous kd-tree was proposed
  - The kd-tree has been the subject of intensive research over the past decade: clustering, searching, storage efficiency and balancing

Binary-tree based indexing: The *kd*-tree (con't)



(a) data space

(b) *kd*-tree

Binary-tree based indexing: The *K-D-B*-tree

- The K-D-B-tree
  - is a combination of a *kd*-tree and B-tree
  - consists of a *region page* and a *point page*
    - region page: <region, page-ID> pairs
    - point page: <point, record-ID> pairs
  - is perfectly height-balanced
  - has poorer storage efficiency, nevertheless

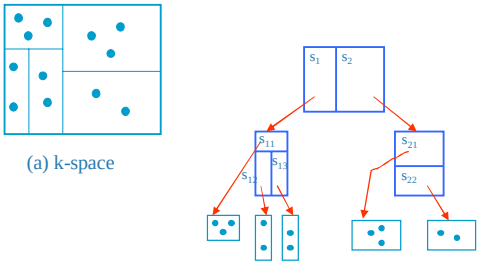
Binary-tree based indexing: The *K-D-B*-tree (Con't)

- Splitting
  - data page splitting
    - A split will occur during insertion of a new point into a *full* point page
    - The two resultant point pages will contain almost the same number of data points
    - The split of a point page may cause the parent region page to split as well, which may propagate upward

Binary-tree based indexing: The *K-D-B*-tree (Con't)

- Splitting
  - region page splitting
    - A split will occur when a region page is full
    - A region page is partitioned into two region pages such that both have almost the same number of entries
    - The split may propagate downward
    - The downward propagation may cause low storage utilization

Binary-tree based indexing: The *K-D-B*-tree (Con't)



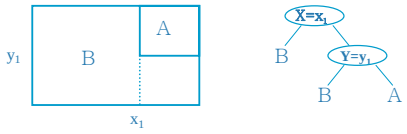
(a) k-space

(b) K-D-B Tree

Binary-tree based indexing: The *hB*-tree

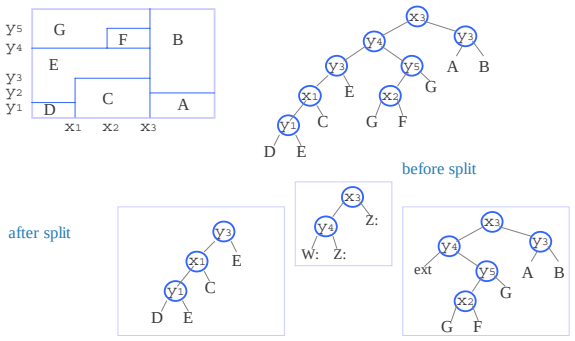
- The hB-tree
  - problem in the K-D-B-tree
    - The split of one index node can cause descendant nodes to be split as well. This may cause sparse index nodes to be created
  - To overcome this problem, the hB-tree (the holey brick B-tree) allows the data space to be holey
  - Based on the K-D-B-tree => height-balanced tree
  - Data nodes + Index nodes
  - Data space may be non-rectangular and kd-tree is used to space representation in internal nodes

Binary-tree based indexing: The *hB*-tree (Con't)



A holey brick is represented via a kd-tree. A holey brick is a brick from which a smaller brick hash been removed. Two leaves of the kd-tree are required to reference the holey brick region denoted by B.

Binary-tree based indexing: The *hB*-tree (Con't)



Binary-tree based indexing: The *hB*-tree (Con't)

- The advantages
  - Overcoming the problem of sparse nodes in the K-D-B-tree
  - The search time and the storage space are reduced because of the use of kd-tree
- The disadvantages
  - The cost of node splitting and node deletion are expensive
  - The multiple references to data nodes may cause a path to be traversed more than once

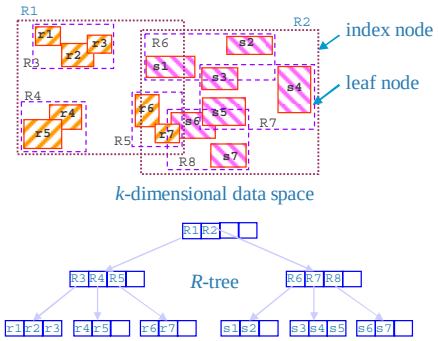
B-tree based indexing: The R-tree

- The R-tree
  - A multi-dimensional generalization of the B-tree
  - A height-balanced tree
  - Having received a great deal of attention due to its well defined structure
  - Like the B-tree, node splitting and merging are required

B-tree based indexing: The R-tree (Con't)

- The structure of the R-tree
  - Leaf node : a set of entries of <MBR, *object-ID*>
    - MBR: a bounding rectangle which bounds its data object
    - object-ID: an object identifier of the data object
  - Index node : a set of entries of <MBR, *child-pointer*>
    - MBR: a bounding rectangle which covers all MBRs in the lower node in the subtree
    - child-pointer: a pointer to a lower level node in the subtree

B-tree based indexing: The R-tree (Con't)



B-tree based indexing: The R-tree (Con't)

- Search
  - Query operations: intersect, contain, within, distance, etc.
  - Query rectangle: a rectangle represented by user
  - The search algorithm
    - Recursively traverse down the subtrees of MBR which intersect the query rectangle
    - When a leaf node is reached, MBRs are tested against the query rectangle and then their objects are tested if they insect the query rectangle
  - Primary performance factor: minimization of overlaps between MBRs of index nodes => determined by the splitting algorithm(different from other R-tree variants)

B-tree based indexing: The R-tree (Con't)

- Insertion
  - Criterion: least coverage
    - The rectangle that needs *least enlargement* to enclose the new object is selected, the one with the *smallest area* is chosen if more than on rectangle meets the first criterion
- Deletion
  - In case that the deletion causes the leaf node to underflow, the node is deleted and all the remaining entries of that node are reinserted from the root

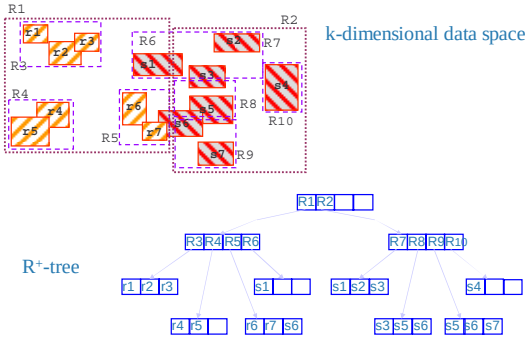
B-tree based indexing: The R*-tree

- The R*-tree
  - Minimization of both coverage and overlap is crucial to the performance of the R-tree. So the near optimal of both minimization was introduced by Beckmann et al.: The criterion that ensures the *quadratic* covering rectangles is used in the insertion and splitting algorithms
  - Dynamic hierarchical spatial indexes are sensitive to the order of the insertion of data: Beckmann proposed a *forced reinsertion* algorithm when a node overflows`

B-tree based indexing: The R$^+$-tree

- The R$^+$-tree
  - A compromise between the R-tree and the K-D-B-tree
  - Overcoming the problem of the overlapping of internal nodes of the R-tree
  - The R$^+$-tree differs from the R-tree:
    - Nodes of an R$^+$-tree are no guaranteed to be at least half filled
    - The entries of any internal node do not overlap
    - An object identifier may be stored in more than one leaf node
  - The disjoint MBRs avoid the multiple search paths for point queries

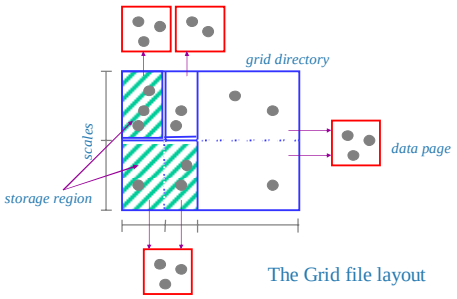
B-tree based indexing: The R$^+$-tree (Con't)



Cell methods based on dynamic hashing: The grid file

- The grid file
  - Based on dynamic hashing for multi-attribute point data
  - Two basic structures: *k linear scales* + *k-dimensional directory*
  - *grid directory*: k-dimensional array
  - Each grid need not contain at least *m* objects. So a data page is allowed to store objects from several grid cells as long as the union of these cells from a rectangular rectangle, which is known as the *storage region*

Cell methods based on dynamic hashing: The grid file (Con't)

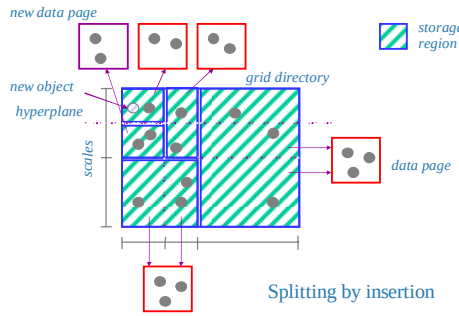


The Grid file layout

Cell methods based on dynamic hashing: The grid file (Con't)

- Splitting by insertion
  - In the case where the data page is *full*, a split is required
    - The split is simple if the storage region covers more than the grid cells
    - Otherwise a new (*k*-1)-dimensional hyperplane partitions the corresponding storage region into two subspaces
      - The corresponding storage region: partition into two regions and distribute objects into the existing page and a new data page
      - Other storage regions: unaffected

Cell methods based on dynamic hashing: The grid file (Con't)



Splitting by insertion

Cell methods based on dynamic hashing: The grid file (Con't)

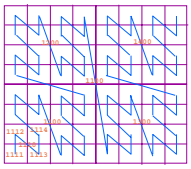
- Merging by deletion
  - Deletion may cause the occupancy of a storage region to fall below an acceptable level, which triggers merging operations
  - If the joint occupancy of two or more adjacent storage regions drops below a threshold, then the data pages are merged into one
  - Two merging approaches: *neighbor* system and *buddy* system

Spatial objects ordering

- The space-filling curves
  - Mapping multi-dimensional objects to one-dimensional values
    - Numbering each grid in a space according to mapping function (e.g., Peano-Hilbert curve, z-ordering, gray-ordering, etc.)
    - one-dimensional locational key is a number
  - A B^+-tree is used to index the objects based on locational keys

Spatial objects ordering (Con't)

e.g.) z-ordering



z-ordering

$$k = \begin{cases} k' + 5^{m-h} & \text{if k is the SW son of k'} \\ k' + 2*5^{m-h} & \text{if k is the NW son of k'} \\ k' + 3*5^{m-h} & \text{if k is the SE son of k'} \\ k' + 4*5^{m-h} & \text{if k is the NE son of k'} \end{cases}$$

mapping function