

# File Structure Notes

## Collected By Manjunath J

Visit: **<http://engginfo2002.tripod.com>** for more Info

For any assistance

Call : (080)6604878

E-mail: [manjunath\\_shiva@rediffmail.com](mailto:manjunath_shiva@rediffmail.com)

# From TextBook

## **File Structures, An Object-Oriented Approach with C++**

by

Michael J. Folk, Bill Zoellick  
and Greg Riccardi

Publisher: Addison Wesley

# Data and File Structures

Introduction to the Design and  
Specification of File Structures

# Outline

- What are File Structures?
- Why Study File Structure Design
- Overview of File Structure Design

# Definition

- A *File Structure* is a combination of *representations* for data in files and of *operations* for accessing the data.
- A File Structure allows applications to *read*, *write* and *modify* data. It might also support *finding* the data that matches some search criteria or *reading through* the data in some particular order.

# Why Study File Structure Design?

## I. Data Storage

- Computer Data can be stored in three kinds of locations:

- Primary Storage ==> Memory

- [Computer Memory]

- Secondary Storage [Online Disk/ Tape/ CDRom that can be accessed by the computer]

- Tertiary Storage ==> Archival Data

- [Offline Disk/Tape/ CDRom not directly available to the computer.]

Our  
Focus

# Why Study File Structure Design?

## II. Memory versus Secondary Storage

- Secondary storage such as disks can pack thousands of megabytes in a small physical location.
- Computer Memory (RAM) is limited.
- However, relative to Memory, access to secondary storage is extremely slow [E.g., getting information from slow RAM takes  $120 \cdot 10^{-9}$  seconds (= 120 nanoseconds) while getting information from Disk takes  $30 \cdot 10^{-3}$  seconds (= 30 milliseconds)]

# Why Study File Structure Design?

## II. How Can Secondary Storage Access Time be Improved?

***By improving the File Structure.***

Since the details of the representation of the data and the implementation of the operations determine the efficiency of the file structure for particular applications, improving these details can help improve secondary storage access time.

# Overview of File Structure Design

## I. General Goals

- Get the information we need with one access to the disk.
- If that's not possible, then get the information with as few accesses as possible.
- Group information so that we are likely to get everything we need with only one trip to the disk.

# Overview of File Structure Design

## II. Fixed versus Dynamic Files

- It is relatively easy to come up with file structure designs that meet the general goals when the files never change.
- When files grow or shrink when information is added and deleted, it is much more difficult.

# History of File Structures

## I. Early Work

- Early Work assumed that files were on tape.
- Access was sequential and the cost of access grew in direct proportion to the size of the file.

# History of File Structures

## II. The emergence of Disks and Indexes

- As files grew very large, unaided sequential access was not a good solution.
- Disks allowed for direct access.
- Indexes made it possible to keep a list of keys and pointers in a small file that could be searched very quickly.
- With the key and pointer, the user had direct access to the large, primary file.

# History of File Structures

## III. The emergence of Tree Structures

- As indexes also have a sequential flavour, when they grew too much, they also became difficult to manage.
- The idea of using tree structures to manage the index emerged in the early 60's.
- However, trees can grow very unevenly as records are added and deleted, resulting in long searches requiring many disk accesses to find a record.

# History of File Structures

## IV. Balanced Trees

- In 1963, researchers came up with the idea of AVL trees for data in memory.
- AVL trees, however, did not apply to files because they work well when tree nodes are composed of single records rather than dozens or hundreds of them.
- In the 1970's came the idea of B-Trees which require an  $O(\log_k N)$  access time where  $N$  is the number of entries in the file and  $k$ , the number of entries indexed in a single block of the B-Tree structure --> B-Trees can guarantee that one can find one file entry among millions of others with only 3 or 4 trips to the disk.

# History of File Structures

## V. Hash Tables

- Retrieving entries in 3 or 4 accesses is good, but it does not reach the goal of accessing data with a single request.
- From early on, *Hashing* was a good way to reach this goal with files that do not change size greatly over time.
- Recently, *Extendible Dynamic Hashing* guarantees one or at most two disk accesses no matter how big a file becomes.

# Data and File Structures

Basic File Processing Operations

# Outline

- Physical versus Logical Files
- Opening and Closing Files
- Reading, Writing and Seeking
- Special Characters in Files
- The Unix Directory Structure
- Physical Devices and Logical Files
- Unix File System Commands

# Physical versus Logical Files

- **Physical File**: A collection of bytes stored on a disk or tape.
- **Logical File**: A “Channel” (like a telephone line) that hides the details of the file’s location and physical format to the program.
- When a program wants to use a particular file, “data”, the operating system must find the physical file called “data” and make the hookup by assigning a logical file to it. This logical file has a logical name which is what is used inside the program.

# Opening Files

- Once we have a logical file identifier hooked up to a physical file or device, we need to declare what we intend to do with the file:
  - Open an existing file
  - Create a new file
- a** That makes the file ready to use by the program
- a** We are positioned at the beginning of the file and are ready to read or write.



# Closing Files

- Makes the logical file name available for another physical file (it's like hanging up the telephone after a call).
- Ensures that everything has been written to the file [since data is written to a buffer prior to the file].
- Files are usually closed automatically by the operating system (unless the program is abnormally interrupted).

# Reading

- Read(Source\_file, Destination\_addr, Size)
  - Source\_file = location the program reads from, i.e., its logical file name
  - Destination\_addr = first address of the memory block where we want to store the data.
  - Size = how much information is being brought in from the file (byte count).

# Writing

- Write(Destination\_file, Source\_addr, Size)
  - Destination\_file = the logical file name where the data will be written.
  - Source\_addr = first address of the memory block where the data to be written is stored.
  - Size = the number of bytes to be written.

# Seeking

- A program does not necessarily have to read through a file sequentially: It can jump to specific locations in the file or to the end of file so as to append to it.
- The action of moving directly to a certain position in a file is often called *seeking*.
- Seek(Source\_file, Offset)
  - Source\_file = the logical file name in which the seek will occur
  - Offset = the number of positions in the file the pointer is to be moved from the start of the file.

# Special Characters in Files I

- Sometimes, the operating system attempts to make “regular” user’s life easier by automatically adding or deleting characters for them.
- These modifications, however, make the life of programmers building sophisticated file structures (YOU) more complicated!

# Special Characters in Files II: Examples

- Control-Z is added at the end of all files (MS-DOS). This is to signal an end-of-file.
- <Carriage-Return> + <Line-Feed> are added to the end of each line (again, MS-DOS).
- <Carriage-Return> is removed and replaced by a character count on each line of text (VMS)

# The Unix Directory Structure I

- In any computer systems, there are many files (100's or 1000's). These files need to be organized using some method. In Unix, this is called the *File System*.
- The Unix File System is a tree-structured organization of directories. With the root of the tree represented by the character “/”.
- Each directory can contain regular files or other directories.
- The file name stored in a Unix directory corresponds to its *physical name*.

# The Unix Directory Structure II

- Any file can be uniquely identified by giving it its *absolute pathname*. E.g., /usr6/mydir/addr.
- The directory you are in is called your *current directory*.
- You can refer to a file by the path relative to the current directory.
- “.” stands for the current directory and “..” stands for the parent directory.

# Physical Devices and Logical Files

- Unix has a very general view of what a file is: it corresponds to a sequence of bytes with no worries about where the bytes are stored or where they come from.
- Magnetic disks or tapes can be thought of as files and so can the keyboard and the console.
- No matter what the physical form of a Unix file (real file or device), it is represented in the same way in Unix: by an integer.

# Stdout, Stdin, Stderr

- Stdout --> Console  
`fwrite(&ch, 1, 1, stdout);`
- Stdin --> Keyboard  
`fread(&ch, 1, 1, stdin);`
- Stderr --> Standard Error (again, Console)  
[When the compiler detects an error, the error message is written in this file]

# I/O Redirection and Pipes

- `< filename` [redirect stdin to “filename”]
- `> filename` [redirect stdout to “filename”]

E.g., `a.out < my-input > my-output`

- `program1 | program2` [take any stdout output from program1 and use it in place of any stdin input to program2.

E.g., `list | sort`

# Unix System Commands

- *cat* filenames --> Print the content of the named textfiles.
- *tail* filename --> Print the last 10 lines of the text file.
- *cp* file1 file2 --> Copy file1 to file2.
- *mv* file1 file2 --> Move (rename) file1 to file2.
- *rm* filenames --> Remove (delete) the named files.
- *chmod* mode filename --> Change the protection mode on the named file.
- *ls* --> List the contents of the directory.
- *mkdir* name --> Create a directory with the given name.
- *rmdir* name --> Remove the named directory.

# Data and File Structures

Secondary Storage and System  
Software: Magnetic Disks  
& Tapes

# Part I: Disks

# Outline

- The Organization of Disks
- Estimating Capacities and Space Needs
- Organizing Tracks by Sector
- Organizing Tracks by Block
- Non Data Overhead
- The Cost of a Disk Access
- Disk as Bottleneck

# General Overview

Having learned how to manipulate files, we now learn about the nature and limitations of the devices and systems used to store and retrieve files, so that we can design good file structures that arrange the data in ways that minimize access costs given the device used by the system.

# Disks: An Overview

- Disks belong to the category of *Direct Access Storage Devices* (DASDs) because they make it possible to access the data directly.
- This is in contrast to Serial Devices (e.g., Magnetic Tapes) which allows only serial access [all the data before the one we are interested in has to be read or written in order].
- Different Types of Disks:
  - Hard Disk: High Capacity + Low Cost per bit.
  - Floppy Disk: Cheap, but slow and holds little data. (zip disks: removable disk cartridges)
  - Optical Disk (CD-ROM): Read Only, but holds a lot of data and can be reproduced cheaply. However, slow.

# The Organization of Disks I

- The information stored on a disk is stored on the surface of one or more *platters*.
- The information is stored in successive *tracks* on the surface of the disk.
- Each track is often divided into a number of *sectors* which is the smallest addressable portion of a disk.

# The Organization of Disks II

- When a read statement calls for a particular byte from a disk file, the computer's operating system finds the correct platter, track and sector, reads the entire sector into a special area in memory called a *buffer*, and then finds the requested byte within that buffer.

# The Organization of Disks III

- Disk drives typically have a number of platters and the tracks that are directly above and below one another form a *cylinder*.
- All the info on a single cylinder can be accessed without moving the arm that holds the *read/write heads*.
- Moving this arm is called *seeking*. The arm movement is usually the *slowest* part of reading information from a disk.

# Estimating Capacities and Space Needs

- Track Capacity = number of sectors per track \* bytes per sector
- Cylinder Capacity = number of tracks per cylinder \* track capacity
- Drive Capacity = number of cylinders \* cylinder capacity

# Data Organization: I. Organizing Tracks per Sector

## The Physical Placement of Sectors

- The most practical logical organization of sectors on a track is that sectors are adjacent, fixed-sized segments of a track that happens to hold a file.
- Physically, however, this organization is not optimal: after reading the data, it takes the disk controller some time to process the received information before it is ready to accept more. If the sectors were physically adjacent, we would use the start of the next sector while processing the info just read in.

# Data Organization: I. Organizing Tracks per Sector (Cont'd)

- **Traditional Solution:** Interleave the sectors. Namely, leave an interval of several physical sectors between logically adjacent sectors.
- Nowadays, however, the controller's speed has improved so that no interleaving is necessary anymore.

# Data Organization:I. Organizing Tracks by Sectors (Cont'd)

- The file can also be viewed as a series of *clusters* of sectors which represent a fixed number of (logically) contiguous sectors.
- Once a cluster has been found on a disk, all sectors in that cluster can be accessed without requiring an additional seek.
- The *File Allocation Table* ties logical sectors to the physical clusters they belong to.

# Data Organization:I. Organizing Tracks by Sectors (Cont'd)

- If there is a lot of free room on a disk, it may be possible to make a file consist entirely of contiguous clusters.  $\implies$  the file consists of one extent.  $\implies$  the file can be processed with a minimum of seeking time.
- If one extent is not enough, then divide the file into more extents.
- As the number of extents in a file increases, the file becomes more spread out on the disk, and the amount of seeking necessary increases.

# Data Organization:I. Organizing Tracks by Sectors (Cont'd)

- There are 2 possible organizations for records (if the records are smaller than the sector size:
  1. Store 1 record per sector
  2. Store the records successively (i.e., one record may span two sectors

# Data Organization:I. Organizing Tracks by Sectors (Cont'd) Trade-Offs

- Advantage of 1: Each record can be retrieved from 1 sector.
- Disadvantage of 1: Loss of Space with each sector ==>  
Internal Fragmentation
- Advantage of 2: No internal fragmentation
- Disadvantage of 2: 2 sectors may need to be accessed to retrieve a single record.
- The use of clusters also leads to internal fragmentation.

# Data Organization: II. Organizing Tracks by Block

- Rather than being divided into sectors, the disk tracks may be divided into user-defined *blocks*.
- When the data on a track is organized by block, this usually means that the amount of data transferred in a single I/O operation can vary depending on the needs of the software designer (not the hardware).
- Blocks can normally be either fixed or variable in length, depending on the requirements of the file designer and the capabilities of the operating system.

## Data Organization: II. Organizing Tracks by Block (Cont'd)

- Blocks don't have the sector-spanning and fragmentation problem of sectors since they vary in size to fit the logical organization of the data.
- The blocking factor indicates the number of records that are to be stored in each block in a file.
- Each block is usually accompanied by subblocks: key-subblock or count-subblock.

# Non-Data Overhead I

- Whether using a block or a sector organization, some space on the disk is taken up by non-data overhead. i.e., information stored on the disk during pre-formatting.
- On sector-addressable disks, pre-formatting involves storing, at the beginning of each sector, sector address, track address and condition (usable or defective) + gaps and synchronization marks between fields of info to help the read/write mechanism distinguish between them.
- On Block-Organized disks, subblock + interblock gaps have to be provided with every block. The relative amount of non-data space necessary for a block scheme is higher than for a sector-scheme.

# Non-Data Overhead II

- The greater the block-size, the greater potential amount of internal track fragmentation.
- The flexibility introduced by the use of blocks rather than sectors can save time since it lets the programmer determine, to a large extent, how the data is to be organized physically on disk.
- Overhead for the programmer and Operating System.
- Can't synchronize I/O operation with movement of disk.

# The Cost of a disk Access

- *Seek Time* is the time required to move the access arm to the correct cylinder.
- *Rotational Delay* is the time it takes for the disk to rotate so the sector we want is under the read/write head.
- *Transfer Time* = (Number of Bytes Transferred/ Number of Bytes on a Track) \* Rotation Time

# Disk as Bottleneck I

- Processes are often *Disk-Bound*, i.e., the network and the CPU often have to wait inordinate lengths of time for the disk to transmit data.
- *Solution 1: Multiprogramming* (CPU works on other jobs while waiting for the disk)
- *Solution 2: Stripping*: splitting the parts of a file on several different drives, then letting the separate drives deliver parts of the file to the network simultaneously ==> Parallelism

# Disk as Bottleneck II

- **Solution 3: RAID**: Redundant Array of Independent Disks
- **Solution 4: RAM disk** ==> Simulate the behavior of the mechanical disk in memory.
- **Solution 5: Disk Cache**= large block of memory configured to contain pages of data from a disk. Check cache first. If not there, go to the disk and replace some page already in cache with page from disk containing the data.

# Data and File Structures

Secondary Storage and System  
Software: Magnetic Disks  
& Tapes

# Part II: Tape

## Outline

- Description of Tape Systems
- Organization of Data on Nine-Track Tapes
- Estimating Tape Length Requirements
- Estimating Data Transmission Times
- Disk versus Tape

# Description of Tape Systems

- No direct accessing facility, but very rapid sequential access.
- Compactness, resistance to rough environmental conditions, easy to store and transport, cheaper than disk
- Used to be used for application data
- Currently, tapes are primarily used as archival storage.

# Organization of Data on Nine-Track Tapes I

- On a tape, the logical position of a byte within a file corresponds directly to its physical position relative to the start of the file.
- The surface of a typical tape can be seen as a set of parallel tracks each of which is a sequence of bits. These bits correspond to 1 byte + a parity bit.
- One Byte = a one-bit-wide slice of tape called a *frame*.

# Organization of Data on Nine-Track Tapes II

- In *odd parity*, the bit is set to make the number of bits in the frame odd. This is done to check the validity of the data.
- Frames are organized into *data blocks* of variable size separated by *interblock gaps* (long enough to permit stopping and starting)

# Estimating Tape Length Requirements I

- Let  $b$  = the *physical length* of a data block
- Let  $g$  = the *length of an interblock gap*, and
- Let  $n$  = the *number* of data blocks.
- The *space requirement*,  $s$ , for storing the file is  $s = n * (b+g)$
- $b$  = blocksize (i.e., bytes per block)/ tape density (i.e., bytes per inch)

# Estimating Tape Length Requirements II

- The number of records stored in a physical block is called the *blocking factor*.
- *Effective Record Density*: a general measure of the effect of choosing different block sizes:  
(number of bytes per block)/ (number of inches required to store a block)
- $\implies$  Space utilization is sensitive to the relative sizes of data blocks and interblock gaps.

# Estimating Data Transmission Times

- *Normal Data Transmission Rate* = (Tape Density (bpi)) \* (Tape Speed (ips))
- Interblock gaps, however, must be taken into consideration ==> *Effective Transmission Rate* / ((Effective Recording Density) \* (Tape Speed))

# Disk versus Tape

- **In the past:** Both Disks and Tapes were used for secondary storage. Disks were preferred for random access and tape was better for sequential access.
- **Now (1):** Disks have taken over much of secondary storage ==> Because of the decreased cost of disk + memory storage
- **Now (2):** Tapes are used as Tertiary storage (Cheap, fast & easy to stream large files or sets of files between tape and disk)

# Data and File

Secondary Storage and System  
Software: CD-ROM & Issues in  
Data Management

# Overview

- CD-ROM (Compact Disk, Read-Only Memory)
- A Journey of a Byte
- Buffer Management
- I/O in Unix

# Introduction to CD-ROM

- A single disc can hold more than 600 megabytes of data (~ 400 books of the textbook's size)
- CD-ROM is read only. i.e., it is a publishing medium rather than a data storage and retrieval like magnetic disks.
- CD-ROM Strengths: High storage capacity, inexpensive price, durability.
- CD-ROM Weaknesses: extremely slow seek performance (between 1/2 a second to a second)  
==> Intelligent File Structures are critical.

# Physical Organization of CD-ROM I

- CD-ROM is a descendent of CD Audios. i.e., listening to music is sequential and does not require fast random access to data.
- *Reading Pits and Lands*: CD-ROMs are stamped from a glass master disk which has a coating that is changed by the laser beam. When the coating is developed, the areas hit by the laser beam turn into *pits* along the track followed by the beam. The smooth unchanged areas between the pits are called *lands*.

# Physical Organization of CD-ROM II

- When we read the stamped copy of the disc, we focus a beam of laser light on the track as it moves under the optical pickup. The pits scatter the light, but the lands reflect most of it back to the pickup. This alternating pattern of high- and low-intensity reflected light is the signal used to reconstruct the original digital information.
- 1's are represented by the transition from pit to land and back again. 0's are represented by the amount of time between transitions. The longer between transitions, the more 0s we have.

# Physical Organization of CD-ROM III

- Given this scheme, it is not possible to have 2 adjacent 1s: 1s are always separated by 0s. As a matter of fact, because of physical limitations, there must be at least two 0s between any pair of 1s.
- Raw patterns of 1s and 0s have to be translated to get the 8-bit patterns of 1s and 0s that form the bytes of the original data.
- EFM encoding (Eight to Fourteen Modulations) turns the original 8 bits of data into 14 expanded bits that can be represented in the pits and lands on the disk.
- Since 0s are represented by the length of time between transition, the disk must be rotated at a precise and constant speed. This affects the CD-ROM drive's ability to seek quickly.

# CLV instead of CAV I

- Data on a CD-ROM is stored in a single, spiral track. This allows the data to be packed as tightly as possible since all the sectors have the same size (whether in the center or at the edge).
- In the “regular arrangement”, the data is packed more densely in the center than in the edge ==> Space is lost in the edge.
- Since reading the data requires that it passes under the optical pick-up device at a constant rate, the disc has to spin more slowly when reading the outer edges than when reading towards the center.

# CLV instead of CAV II

- The CLV format is responsible, in large part, for the poor seeking performance of CD-ROM Drives: there is no straightforward way to jump to a location. Part of the problem is the need to change rotational speed.
- To read the address info that is stored on the disc along with the user's data, we need to be moving the data under the optical pick up at the correct speed. But to know how to adjust the speed, we need to be able to read the address info so we know where we are. How do we break this loop? By guessing and through trial and error ==> Slows down performance.

# Addressing

- Different from the “regular” disk method.
- Each second of playing time on a CD is divided into 75 sectors. Each sector holds 2 Kilobytes of data. Each CD-ROM contains at least one hour of playing time.
- $\implies$  The disc is capable of holding at least  $60 \text{ min} * 60 \text{ sec/min} * 75 \text{ sector/sec} * 2 \text{ Kilobytes/sector} = 540,000 \text{ KBytes}$
- Often, it is actually possible to store over 600,000 KBytes.
- Sectors are addressed by min:sec:sector e.g., 16:22:34

# CD-ROM Strengths & Weaknesses

- *Seek Performance*: very bad
- *Data Transfer Rate*: Not Terrible/Not Great
- *Storage Capacity*: Great
  - Benefit: enables us to build indexes and other support structures that can help overcome some of the limitations associated with CD-ROM's poor performance.
- *Read-Only Access*: There can't be any changes ==> File organization can be optimized.
- No need for *interaction* with the user (which requires a quick response)

# A Journey of A Byte: What happens when the program statement: `write(textfile, ch, 1)` is executed ?

## Part that takes place in memory:

- Statement calls the Operating System (OS) which oversees the operation
- File manager (Part of the OS that deals with I/O)
  - Checks whether the operation is permitted
  - Locates the physical location where the byte will be stored (Drive, Cylinder, Track & Sector)
  - Finds out whether the sector to locate the 'P' is already in memory (if not, call the I/O Buffer)
  - Puts 'P' in the I/O Buffer
  - Keep the sector in memory to see if more bytes will be going to the same sector in the file

A Journey of A Byte: What happens when the program statement: `write(textfile, ch, 1)` is executed (Cont'd) ?

**Part that takes place outside of memory:**

- **I/O Processor**: Wait for an external data path to become available (CPU is faster than data-paths ==> Delays)
- **Disk Controller**:
  - I/O Processor asks the disk controller if the disk drive is available for writing
  - Disk Controller instructs the disk drive to move its read/write head to the right track and sector.
  - Disk spins to right location and byte is written

# Buffer Management

- What happens to data travelling between a program's data area and secondary storage?
- *The use of Buffers*: Buffering involves working with a large chunk of data in memory so the number of accesses to secondary storage can be reduced.

# Buffer Bottlenecks

- Assume that the system has a single buffer and is performing both input and output on one character at a time, alternatively.
- In this case, the sector containing the character to be read is constantly over-written by the sector containing the spot where the character will be written, and vice-versa.
- In such a case, the system needs more than 1 buffer: at least, one for input and the other one for output.
- Moving data to or from disk is very slow and programs may become I/O Bound ==> Find better strategies to avoid this problem.

# Buffering Strategies

- Multiple Buffering
  - Double Buffering
  - Buffer Pooling
- Move Mode and Locate Mode
- Scatter/Gather I/O

# Data and File Structures

Fundamental File Structure  
Concepts & Managing Files of  
Records

# Outline I: Fundamental File

## Structure Concepts

- Stream Files
- Field Structures
- Reading a Stream of Fields
- Record Structures
- Record Structures that use a length indicator

# Outline II: Managing Files of Records

- Record Access
- More About Record Structures
- File Access and File Organization
- More Complex File Organization and Access
- Portability and Standardization

# Field and Record Organization: Overview

- The basic logical unit of data is the *field* which contains a single data value.
- Fields are organized into aggregates, either as many copies of a single field (an *array*) or as a list of different fields (a *record*).
- When a record is stored in memory, we refer to it as an *object* and refer to its fields as *members*.
- In this lecture, we will investigate the many ways that objects can be represented as records in files.

# Stream Files

- Mary Ames
  - 123 Maple
  - Stillwater, OK 74075
  - Alan Mason
  - 90 Eastgate
  - Ada, OK 74820
- In Stream Files, the information is written as a stream of bytes containing no added information:  
AmesMary123 MapleStillwaterOK74075MasonAlan90 EastgateAdaOK74820
  - **Problem:** There is no way to get the information back in the organized record format.

# Field Structures

- There are many ways of adding structure to files to maintain the identity of fields:
  - Force the field into a predictable length
  - Begin each field with a length indicator
  - Use a “keyword = value” expression to identify each field and its content.

# Reading a Stream of Fields

- A Program can easily read a stream of fields and output `====>`
- This time, we do preserve the notion of fields, but something is missing: Rather than a stream of fields, these should be two records

Last Name: 'Ames'  
First Name: 'Mary'  
Address: '123 Maple'  
City: 'Stillwater'  
State: 'OK'  
Zip Code: '74075'  
Last Name: 'Mason'  
First Name: 'Alan'  
Address: '90 Eastgate'  
City: 'Ada'  
State: 'OK'  
Zip Code: '74820'

# Record Structure I

- A *record* can be defined as a set of fields that belong together when the file is viewed in terms of a higher level of organization.
- Like the notion of a field, a record is another conceptual tool which needs not exist in the file in any physical sense.
- Yet, they are an important logical notion included in the file's structure.

# Record Structures II

- Methods for organizing the records of a file include:
  - Requiring that the records be a predictable number of bytes in length.
  - Requiring that the records be a predictable number of fields in length.
  - Beginning each record with a length indicator consisting of a count of the number of bytes that the record contains.
  - Using a second file to keep track of the beginning byte address for each record.
  - Placing a delimiter at the end of each record to separate it from the next record.

# Record Structures that Use a Length Indicator

- The notion of records that we implemented are lacking something: none of the variability in the length of records that was inherent in the initial stream file was conserved.
- Implementation:
  - Writing the variable-length records to the file
  - Representing the record length
  - Reading the variable-length record from the file.

# Record Access: Keys

- When looking for an individual record, it is convenient to identify the record with a key based on the record's content (e.g., the Ames record).
- Keys should uniquely define a record and be unchanging.
- Records can also be searched based on a secondary key. Those do not typically uniquely identify a record.

# Sequential Search

- Evaluating Performance of Sequential Search.
- Improving Sequential Search Performance with Record Blocking.
- When is Sequential Search Useful?

# Direct Access

- How do we know where the beginning of the required record is?
- ➔ It may be in an Index (discussed in a different lecture)
- ➔ We know the relative record number (RRN)
- RRN are not useful when working with variable length-records: the access is still sequential.
- With fixed-length records, however, they are useful.

# Record Structure

- Choosing a Record Structure and Record Length within a fixed-length record. 2 approaches:
  - Fixed-Length Fields in record (simple but problematic).
  - Varying Field boundaries within the fixed-length record.
- **Header Records** are often used at the beginning of the file to hold some general info about a file to assist in future use of the file.

# File Access and File Organization: A Summary

- File organization depends on what use you want to make of the file.
- Since using a file implies accessing it, file access and file organization are intimately linked.
- Example: though using fixed-length records makes direct access easier, if the documents have very variable lengths, fixed-length records is not a good solution: the application determines our choice of both access and organization.

# Beyond Record Structure

- Abstract Data Models for File Access
- Headers and Self-Describing File
- Metadata
- Color Raster Images
- Mixing Object Types in One File
- Representation-Independent File Access
- Extensibility

# Portability and Standardization

- Factors Affecting Portability
  - Differences among Operating Systems
  - Differences among Languages
  - Differences in Machine Architectures
- Achieving Portability
  - Agree on a Standard Physical Record Format and Stay with it
  - Agree on a Standard Binary Encoding for Data Elements
  - Number and Text Conversion
  - File Structure Conversion
  - File System Differences
  - Unix and Portability

# Data and File

Organizing Files for Performance

# Overview

- In this lecture, we continue to focus on file organization, but with a different motivation.
- This time we look at ways to organize or re-organize files in order to improve performance.

# Outline

- We will be looking at four different issues:
  - Data Compression: how to make files smaller
  - Reclaiming space in files that have undergone deletions and updates
  - Sorting Files in order to support binary searching ==> Internal Sorting
  - A better Sorting Method: KeySorting

# Data Compression I: An Overview

- **Question:** Why do we want to make files smaller?
- **Answer:**
  - To use less storage, i.e., saving costs
  - To transmit these files faster, decreasing access time or using the same access time, but with a lower and cheaper bandwidth
  - To process the file sequentially faster.

# Data Compression II: Using a Different Notation => Redundancy Compression

- In the previous lectures, when referring to the state field, we used 2 ASCII bytes=16 bits. Was that really necessary?
- Answer: Since there are only 50 states, we could encode them all with only 6 bits, thus saving 1 byte per state field.
- Disadvantages:
  - Not Human-Readable
  - Cost of Encoding/Decoding Time
  - Increased Software Complexity (Encoding/Decoding Module)

# Data Compression II: Suppressing Repeating Sequences ==> Redundancy Compression

- When the data is represented in a Sparse array, we can use a type of compression called: *run-length encoding*.
- Procedure:
  - Read through the array in sequence except where the same value occurs more than once in succession.
  - When the same value occurs more than once, substitute the following 3 bytes in order:
    - The special run-length code indicator
    - The values that is repeated; and
    - The number of times that the value is repeated.
- No guarantee that space will be saved!!!

# Data Compression III: Assigning Variable-Length Code

- **Principle:** Assign short codes to the most frequent occurring values and long ones to the least frequent ones.
- The code-size cannot be fully optimized as one wants codes to occur in succession, without delimiters between them, and still be recognized.
- This is the principle used in the Morse Code
- As well, it is used in Huffman Coding. ==> Used for compression in Unix (see slide 9).

# Data Compression IV: Irreversible Compression Techniques

- *Irreversible Compression* is based on the assumption that some information can be sacrificed. [Irreversible compression is also called *Entropy Reduction*].
- Example: Shrinking a raster image from 400-by-400 pixels to 100-by-100 pixels. The new image contains 1 pixel for every 16 pixels in the original image.
- There is usually no way to determine what the original pixels were from the one new pixel.
- In data files, irreversible compression is seldom used. However, it is used in image and speech processing.

# Data Compression V: Compression in Unix I:

## Huffman Coding (pack and unpack)

- Suppose messages are made of letters a, b, c, d, and e, which appear with probabilities .12, .4, .15, .08, and .25, respectively.
- We wish to encode each character into a sequence of 0's and 1's so that no code for a character is the *prefix* for another.
- Answer (using Huffman's algorithm given on the next slide): a=1111, b=0, c=110, d=1110, e=10.

# Constructing Huffman Codes

(A FOREST is a collection of TREES; each TREE has a root and a weight)

- While there is more than one TREE in the FOREST {
- $i =$  index of the TREE in FOREST with smallest weight;
  - $j =$  index of the TREE in FOREST with 2nd smallest weight;
  - Create a new node with left child FOREST( $i$ )--> root and right child FOREST( $j$ )--> root
  - Replace TREE  $i$  in FOREST by a tree whose root is the new node and whose weight is FOREST( $i$ )--> weight + FOREST( $j$ )--> weight
  - Delete TREE  $j$  from FOREST }

# Data Compression VI: Compression in Unix II: Lempel-Ziv (compress and uncompress)

- **Principle:** Compression of an arbitrary sequence of bits can be achieved by always coding a series of 0's and 1's as some previous such string (the prefix string) plus one new bit. Then the new string formed by adding the new bit to the previously used prefix string becomes a potential prefix string for future strings.
- **Example:** Encode 101011011010101011
- **Answer:** 00010000001000110101011110101101 (see procedure given on slide 12)
- If the initial string is short, the encoding may be longer as above, however, for long documents this encoding is close to optimal.

# Constructing Lempel-Ziv Codes

- **Step 1:** Parse the input string into comma separated phrases that represent strings that can be represented by a previous string as a **prefix** + 1 bit.
- **Step 2:** Encode the different phrases (except the last one) using a minimal binary representation. Start with the null phrase.
- **Step 3:** Write the string, listing 1) the code for the prefix phrase + the new bit needed to create the new phrase.

# Reclaiming Space in Files I: Record Deletion and Storage Compaction

- Recognizing Deleted Records
- Reusing the space from the record  $\implies$  *Storage Compaction*.
- Storage Compaction: After deleted records have accumulated for some time, a special program is used to reconstruct the file with all the deleted approaches.
- Storage Compaction can be used with both fixed- and variable-length records.

# Reclaiming Space in Files II: Deleting Fixed-Length Records for Reclaiming Space Dynamically

- In some applications, it is necessary to reclaim space immediately.
- To do so, we can:
  - Mark deleted records in some special ways
  - Find the space that deleted records once occupied so that we can reuse that space when we add records.
  - Come up with a way to know immediately if there are empty slots in the file and jump directly to them.
- Solution: Use an avail linked list in the form of a stack. Relative Record Numbers (RRNs) play the role of pointers.

# Reclaiming Space in Files III: Deleting Variable-Length Records for Reclaiming Space Dynamically

- Same ideas as for Fixed-Length Records, but a different implementation must be used.
- In particular, we must keep a byte count of each record and the links to the next records on the avail list cannot be the RRNs.
- As well, the data structure used for the avail list cannot be a stack since we have to make sure that when re-using a record it is of the right size.

# Reclaiming Space in Files IV: Storage Fragmentation

- Wasted Space within a record is called *internal Fragmentation*.
- Variable-Length records do not suffer from internal fragmentation. However, *external fragmentation* is not avoided.
- 3 ways to deal with external fragmentation:
  - Storage Compaction
  - Coalescing the holes
  - Use a clever placement strategy

# Reclaiming Space in Files V: Placement Strategies I

- **First Fit Strategy**: accept the first available record slot that can accommodate the new record.
- **Best Fit Strategy**: choose the first available smallest available record slot that can accommodate the new record.
- **Worst Fit Strategy**: choose the largest available record slot.

# Reclaiming Space in Files V:

## Placement Strategies II

- Some general remarks about placement strategies:
  - Placement strategies only apply to variable-length records
  - If space is lost due to internal fragmentation, the choice is first fit and best fit. A worst fit strategy truly makes internal fragmentation worse.
  - If the space is lost due to external fragmentation, one should give careful consideration to a worst-fit strategy.

# Finding Things Quickly I: Overview I

- The cost of Seeking is very high.
- This cost has to be taken into consideration when determining a strategy for searching a file for a particular piece of information.
- The same question also arises with respect to sorting, which often is the first step to searching efficiently.
- Rather than simply trying to sort and search, we concentrate on doing so in a way that minimizes the number of seeks.

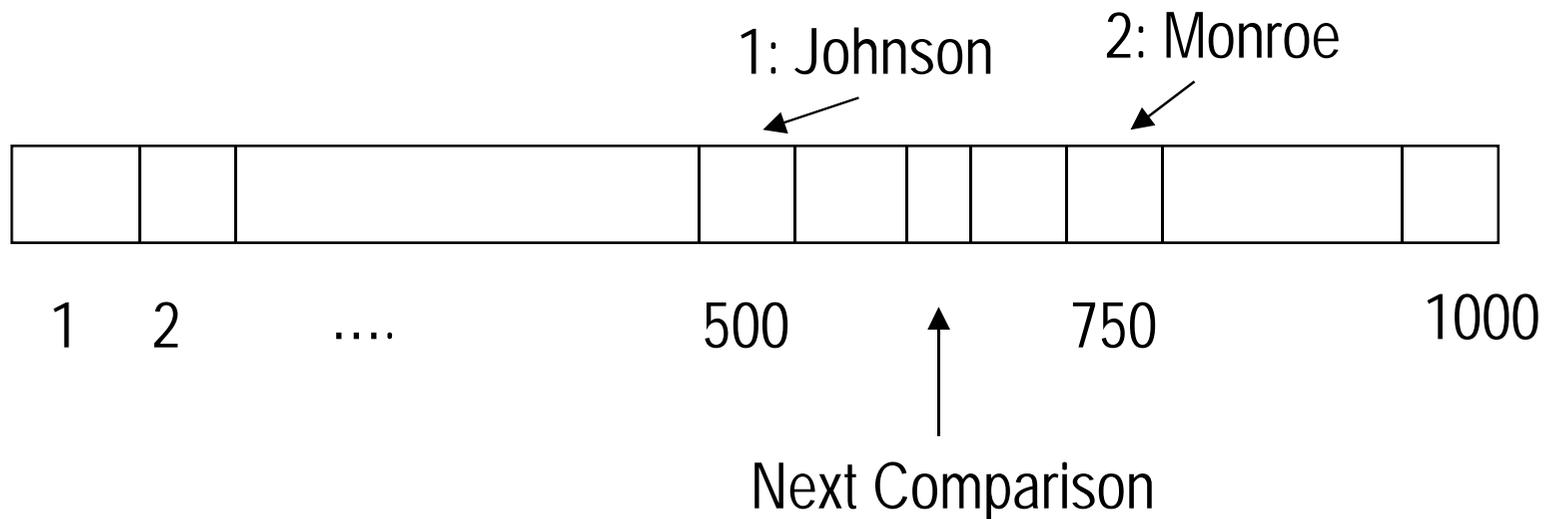
# Finding things Quickly II:

## Overview II

- So far, the only way we have to retrieve or find records *quickly* is by using their RRN (in case the record is of fixed-length).
- Without a RRN or in the case of variable-length records, the only way, so far, to look for a record is by doing a sequential search. This is a very *inefficient* method.
- We are interested in *more efficient* ways to retrieve records based on their *key-value*.

# Finding things Quickly III: Binary Search

- Let's assume that the file is sorted and that we are looking for record whose key is *Kelly* in a file of 1000 fixed-length records.



# Finding things Quickly IV: Binary Search versus Sequential Search

- Binary Search of a file with  $n$  records takes  $O(\log_2 n)$  comparisons.
- Sequential search takes  $O(n)$  comparisons.
- When *sequential search* is used, *doubling* the number of records in the file *doubles* the number of comparisons required for sequential search.
- When binary search is used, *doubling* the number of records in the file only *adds one* more guess to our worst case.
- In order to use binary search, though, the file first has to be sorted. This can be very expensive.

# Finding things Quickly V: Sorting a Disk File in Memory

- If the entire content of a file can be held in memory, then we can perform an internal sort. Sorting in memory is very efficient.
- However, if the file does not hold entirely in memory, any sorting algorithm will require a large number of seeks. Sorting would, thus, be extremely slow. Unfortunately, this is often the case, and solutions have to be found.

# Finding things Quickly VI: The limitations of Binary Search and Internal Sorting

- Binary Search requires more than one or two accesses. Accessing a record using the RRN can be done with a single access ==> We would like to achieve RRN retrieval performance while keeping the advantage of key access.
- Keeping a file sorted is very expensive: in addition to searching for the right location for the insert, once this location is found, we have to shift records to open up the space for insertion.
- Internal Sorting only works on small files. ==> Keysorting

# Finding things Quickly VII: KeySorting

- Overview: when sorting a file in memory, the only thing that really needs sorting are record keys.
- Keysort algorithms work like internal sort, but with 2 important differences:
  - Rather than read an entire record into a memory array, we simply read each record into a temporary buffer, extract the key and then discard.
  - If we want to write the records in sorted order, we have to read them a second time.

## Finding things Quickly VIII: Limitation of the KeySort Method

- Writing the records in sorted order requires as many random seeks as there are records.
- Since writing is interspersed with reading, writing also requires as many seeks as there are records.
- Solution: Why bother to write the file of records in key order: simply write back the sorted index.

# Finding things Quickly IX: Pinned Records

- Indexes are also useful with regard to deleted records.
- The avail list indicating the location of unused records consists of *pinned* records in the sense that these unused records cannot be moved since moving them would create *dangling pointers*.
- Pinned records make sorting very difficult. One solution is to use an ordered index and not to move the records.

# Data and File Structures

Indexing

# Overview

- An *index* is a table containing a list of keys associated with a reference field pointing to the record where the information referenced by the key can be found.
- *An index lets you impose order on a file without rearranging the file.*
- A simple index is simply an array of (key, reference) pairs.
- You can have different indexes for the same data: *multiple access paths.*
- Indexing give us *keyed access to variable-length record files.*

# A Simple Index for Entry- Sequenced Files I

- 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)

# A Simple Index for Entry- Sequenced Files II

- We choose to organize the file as a series of *variable-length record* with a size field preceding each record. The fields within each record are also of variable-length but are separated by delimiters.
- We form a *primary key* by concatenating the record company label code and the record's ID number. This should form a unique identifier.

# A Simple Index for Entry- Sequenced Files III

- In order to provide rapid keyed access, we build a *simple index* with a *key field* associated with a *reference field* which provides the address of the first byte of the corresponding data record.
- The index may be sorted while the file does not have to be. This means that the data file may be *entry sequenced*: the records occur in the order they are entered in the file.

# A Simple Index for Entry- Sequenced Files IV

A few comments about our Index Organization:

- The index is easier to use than the data file because 1) it uses fixed-length records and 2) it is likely to be much smaller than the data file.
- By requiring fixed-length records in the index file, we impose a limit on the size of the primary key field. This could cause problems.
- The index could carry more information than the key and reference fields. (e.g., we could keep the length of each data file record in the index as well).

# Basic Operations on an Indexed Entry-Sequenced File

- **Assumption:** the index is small enough to be held in memory. Later on, we will see what can be done when this is not the case.
  - Create the original empty index and data files
  - Load the index into memory before using it.
  - Rewrite the index file from memory after using it.
  - Add records to the data file and index.
  - Delete records from the data file.
  - Update records in the data file.

# Creating, Loading and Re-writing

- The index is represented as an array of records. The loading into memory can be done sequentially, reading a large number of index records (which are short) at once.
- What happens if the index changed but its re-writing does not take place or takes place incompletely?
  - Use a mechanism for indicating whether or not the index is out of date.
  - Have a procedure that reconstructs the index from the data file in case it is out of date.

# Record Addition

- When we add a record, both the data file and the index should be updated.
- In the data file, the record can be added anywhere. However, the *byte-offset* of the new record should be saved.
- Since the index is sorted, the location of the new record does matter: we have to shift all the records that belong after the one we are inserting to open up space for the new record. However, this operation is not too costly as it is performed in memory.

# Record Deletion

- Record deletion can be done using the methods discussed last week (and in Chapter 6).
- In addition, however, the index record corresponding to the data record being deleted must also be deleted. Once again, since this deletion takes place in memory, the record shifting is not too costly.

# Record Updating

- Record updating falls into two categories:
  - The update changes the value of the key field.
  - The update does not affect the key field.
- In the first case, both the index and data file may need to be reordered. The update is easiest to deal with if it is conceptualized as a delete followed by an insert (but the user needs not know about this).
- In the second case, the index does not need reordering, but the data file may. If the updated record is smaller than the original one, it can be re-written at the same location. If, however, it is larger, then a new spot has to be found for it. Again the delete/insert solution can be used.

# Indexes that are too large to hold in memory I

- Problems:
  - Binary searching requires several seeks rather than being performed at memory speed.
  - Index rearrangement requires shifting or sorting records on secondary storage ==> Extremely time consuming.
- Solutions:
  - Use a hashed organization
  - Use a tree-structured index (e.g., a B-Tree)

# Indexes that are too large to hold in memory II

- Nonetheless, simple indexes should not be completely discarded:
  - They allow the use of a binary search in a variable-length record file.
  - If the index entries are significantly smaller than the data file records, sorting and file maintenance is faster.
  - If there are pinned records in the data file, rearrangements of the keys are possible without moving the data records.
  - They can provide access by multiple keys.

# 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. ==> Not that useful!
- 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 (see why later). Instead, we relate the secondary key to a primary key which then will point to the actual byte offset.

# Record Addition in multiple key access settings

- When a secondary index is used, adding a record involves updating the data file, the primary index and the secondary index. The secondary index update is similar to the primary index update.
- Secondary keys are entered in canonical form (all capitals). The upper- and lower- case form must be obtained from the data file. As well, because of the length restriction on keys, secondary keys may sometimes be truncated.
- The secondary index may contain duplicate (the primary index couldn't).

# Record Deletion in multiple key access settings

- Removing a record from the data file means removing its corresponding entry in the primary index and may mean removing all of the entries in the secondary indexes that refer to this primary index entry.
- However, it is also possible not to worry about the secondary index (since, as we mentioned before, secondary keys were made to point at primary ones). ==> savings associated with the lack of rearrangement of the secondary index.
- Cost associated with not purging the secondary index.

# Record Updating in multiple key access settings

- Three possible situations:
  - Update changes the secondary key: may have to rearrange secondary index.
  - Update changes the primary key: changes to the primary index are required, but very few are needed for the secondary index.
  - Update confined to other fields: no changes necessary to primary nor secondary index.

# Retrieval using combinations of secondary keys

- With secondary keys, we can now search for things like all the recordings of “Beethoven’s work” or all the recordings titled “Violin Concerto”.
- More importantly, we can use combinations of secondary keys. (e.g., find all recordings of Beethoven’s Symphony no. 9).
- Without the use of secondary indexes, this request requires a very expensive sequential search through the entire file. Using secondary indexes, responding to this query is simple and quick.

# Improving the secondary index structure I: The problem

- Secondary indexes lead to two difficulties:
- The index file has to be rearranged *every time* a new record is added to the file.
- If there are duplicate secondary keys, the secondary key field is repeated for each entry ==> Space is wasted.

# Improving the secondary index structure II: Solution 1

- **Solution 1:** Change the secondary index structure so it associates an array of reference with each secondary key.
- **Advantage:** helps avoid the need to rearrange the secondary index file too often.
- **Disadvantages:**
  - It may restrict the number of references that can be associated with each secondary key.
  - It may cause internal fragmentation, i.e., waste of space.

# Improving the secondary index

## structure III: Solution 2

- **Method:** each secondary key points to a different list of primary key references. Each of these lists could grow to be as long as it needs to be and no space would be lost to internal fragmentation.

### **Advantages:**

- The secondary index file needs to be rearranged only upon record addition.
- The rearranging is faster.
- It is not that costly to keep the secondary index on disk.
- The primary index never needs to be sorted.
- Space from deleted primary index records can easily be reused.

### **Disadvantage:**

- Locality (in the secondary index) has been lost ==> More . seeking may be necessary.

# Selective Indexes

- Using secondary keys, you can divide the file into parts and provide a *selective* view.
- For example, you can build a *selective index* that contains only titles to classical recordings or recordings released prior to 1970, and since 1970.
- A possible query could then be: “List all the recordings of Beethoven’s Symphony no. 9 released since 1970.”

# Binding I

- **Question:** At what point is the key bound to the physical address of its associated record?
- **Answer so far:** the binding of our primary keys takes place at construction time. The binding of our secondary keys takes place at the time they are used.
- **Advantage of construction time binding:**
  - Faster access
- **Disadvantage of construction time binding:**
  - Reorganization of the data file must result in modifications to all bound index files.
- **Advantage of retrieval time binding:**
  - Safer

# Binding II

- Tradeoff in binding decisions:
  - Tight, construction time binding is preferable when:
    - The data file is static or nearly static, requiring little or no adding, deleting or updating.
    - Rapid performance during actual retrieval is a high priority.
  - Postponing binding as long as possible is simpler and safer when the data file requires a lot of adding, deleting and updating.

# Data and File Structures

Cosequential Processing and the  
Sorting of Large Files

# Definition

- *Cosequential operations* involve the coordinated processing of two or more sequential lists to produce a single output list.
- This is useful for *merging* (or taking the *union*) of the items on the two lists and for *matching* (or taking the *intersection*) of the two lists.
- These kinds of operations are extremely useful in file processing.

# Overview

- **Part 1:**
  - Development of a general model for doing co-sequential operations.
  - Illustration of this model's use for simple matching and merging operations.
  - Application of this model to a more complex general ledger program
- **Part 2:**
  - Multi-Way Merging
  - External Sort-Merge

# A Model for Implementing Cosequential Processes: Matching I

## Matching Names in Two Lists

- Adams
- Carter
- Chin
- Davis
- Foster
- Garwick
- James
- Johnson
- Karns
- Lambert
- Miller
- Adams
- Anderson
- Andrews
- Bech
- Burns
- Carter
- Davis
- Dempsey
- Gray
- James
- Johnson
- Katz
- Peters

# A Model for Implementing Consequential Processes: Matching II

## Matching names in two lists: Matters to Consider:

- Initializing: we need to arrange things so that the procedure gets going properly.
- Getting and accessing the next list item: we need simple methods to do so.
- Synchronizing: we have to make sure that the current item from one list is never so far ahead of the current item on the other that a match will be missed.
- Handling end-of-file conditions
- Recognizing Errors
- Matching the names efficiently --> Good synchronization

# A Model for Implementing Cosequential Processes: Matching III

## *Synchronization*

- Let Item(1) be the current item from list 1 and Item(2) be the current item from list 2.
- *Rules:*
  - If  $\text{Item}(1) < \text{Item}(2)$ , get the next item from list 1.
  - If  $\text{Item}(1) > \text{Item}(2)$ , get the next item from list 2.
  - If  $\text{Item}(1) = \text{Item}(2)$ , output the item and get the next items from the two lists.

# A Model for Implementing Cosequential Processes: Merging I

- The matching procedure can easily be modified to handle *merging* of two lists.
- An important difference between matching and merging is that with merging, we must read *completely* through each of the lists.
- We have to recognize, however, when one of the two lists has been completely read and avoid reading again from it.

# Application of the Cosequential Model to a General Ledger Program I

- **The problem**: To design a general ledger posting program as part of an accounting system.
- The system contains:
  - A **journal file**: with the monthly transactions that are ultimately to be posted to the ledger file.
  - A **ledger file** containing month-by-month summaries of the values associated with each of the bookkeeping accounts.
- **Posting** involves associating each transaction with its account in the ledger.

# Application of the Cosequential Model to a General Ledger Program II

- How is the posting process implemented?
- **Solution 1:** Build an index for the ledger organized by account number. ==> **2 problems:** 1) lots of seeking back and forth; 2) the journal entries relating to one account are not collected together.
- **Solution 2:** collect all the journal transactions that relate to a given account by sorting the journal transactions by account number and working through the ledger and the sorted journal **cosequentially.**

# Application of the Cosequential Model to a General Ledger Program III

- *Goal of our program*: To produce a printed version of the ledger that not only shows the beginning and current balance for each account but also lists all the journal transactions for the month.
- From the point of view of the ledger accounts, the posting process is a *merge* (even unmatched ledger accounts appear in the output). From the point of view of the journal accounts, the posting process is a *match*.
- Our program must implement a combined merge/match while simultaneously printing account title lines, individual transactions and summary balances.

# Application of the Cosequential Model to a General Ledger Program IV

- Summary of the steps involved in processing the ledger entries:
  - Immediately after reading a new ledger object, print the header line and initialize the balance for the next month from the previous month's balance.
  - For each transaction object that matches, update the account balance.
  - After the last transaction for the account, print the balance line.

# Application of the Cosequential Model to a General Ledger Program V

## **The posting process has three cases:**

- If the ledger account number is less than the journal transaction account number, then print the ledger account balance and then read in the next ledger account and print its title line if the account exists.
- If the account numbers match, then add the transaction amount to the account balance, print the description of the transaction, and read the next journal entry.
- If the journal account is less than the ledger account, then it is an unmatched journal account. Print an error message and continue with the next transaction.

# A K-Way Merge Algorithm

*Let there be two arrays:*

- An array of  $k$  lists and
- 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:*

- Find the index of the minimum current item,  $\text{minItem}$
- Process  $\text{minItem}$  (output it to the output list)
- For  $i=0$  until  $i=k-1$  (in increments of 1)
  - If the current item of list  $i$  is equal to  $\text{minItem}$  then advance list  $i$ .
- Go back to the first step.

# A Selection Tree for Merging Large Number of Lists

- The K-Way Merging Algorithm just described works well if  $k < 8$ . Otherwise, the number of comparisons needed to find the minimum value each step of the way is very large.
- Instead, it is easier to use a selection tree which allows us to determine a minimum key value more quickly.
- Merging  $k$  lists using this method is related to  $\log_2 k$  (the depth of the selection tree) rather than to  $k$ .
- Updating selection trees is not easy  $\implies$  Keep a *tree of losers* (Knuth, 73).

# Keeping Trees of Losers rather than Trees of Winners I

- *Advantages of the Tree of Losers:*
- When using a *tree of winners*, the records with which the winner has to be compared--so as to find the next winner--are located in different subtrees. Updating such a tree is not very convenient.
- When using a *tree of losers*,
  - The value of each leaf (apart from the smallest, the winner) occurs only once in an internal node)
  - All the records with which the winner has to be compared lie on a path from the winner leaf to the root.
  - As long as each node in the tree has a pointer to its parent, then it is very easy to find the next winner.

# Keeping Trees of Losers rather than Trees of Winners II

- *Algorithm for updating a selection tree of losers:*
- *T* is a pointer to an internal node in the tree of losers
- *topoftree* is a flag indicating if updating has reached the root

```
T <-- parent of Buffer[s]
topoftree <-- false
repeat   if key(Buffer(loser(T))) < key(Buffer[s])
         then interchange loser(T) and s
         if T = root
         then topoftree <-- true
         else T <-- parent of node pointed to by T
until topoftree
```

# An Efficient Approach to Sorting in Memory

- When we previously discussed sorting a file that is small enough to fit in memory, we assumed that:
  - We would read the entire file from disk into memory.
  - We would sort the records using a standard sorting procedure, such as shellsort.
  - We would write the file back to disk.
- If the file is read and written as efficiently as possible and if the best sorting algorithm is used, it seems that we cannot improve the efficiency of this procedure.
- Nonetheless, we can improve it by doing things *in parallel*: we can do the reading or writing at the same time as the sorting.

# Overlapping Processing and I/O:

## Heapsort

- Heapsort can be combined with reading from the disk and writing to the disk as follows:
  - The heap can be built while reading the file.
  - Sorting can be done while writing to the file.
- Heaps show certain similarities with selection trees, but they have a somewhat looser structure.
- Heaps have three important properties:
  - Each node has a single key and that key is **greater than or equal** to the key at its parent node.
  - A Heap is a **complete** binary tree.
  - Storage can be allocated sequentially as an array with left and right children of node  $i$  located at index  **$2i$**  and  **$2i+1$**  respectively. ==> Pointers are unnecessary.

# Building the Heap

Insert(NewKey) {

- if (NumElements=MaxElements) return false
- NumElement++
- HeapArray[NumElements]= NewKey
- int k=NumElements; int parent;
- while (k>1)
  - { parent=k/2
  - if (Compare(k, parent) >= 0) break;
  - else Exchange(k, parent);
  - k=parent}
- Return true}

# Building the Heap while Reading the File I

- Rather than seeking every time we want a new record, we read blocks of records at a time into a buffer and operate on that block before moving to a new block.
- The input buffer for each new block of keys becomes part of the memory area set up for the heap. Each time we read a new block, we just append it to the end of the heap.
- The first new record is then at the end of the heap array, as required by the insert function.
- Once a record is inserted, the next new record is at the end of the heap array ready to be inserted as well.

# Building the Heap while Reading the File II

- Reading block saves on seek time, but it does not allow to build the heap while reading input.
- In order to do so, we need to use *multiple buffers*: as we process the keys in one block from the file, we can simultaneously read later blocks from the file.
- *Question:* How many buffers should be used and where should we put them?
- *Answer:* the number of buffers is the number of blocks in the file, and they are located in sequence in the array.
- Note: since building the heap can be faster than reading blocks, there may be some delays in processing.

# Heap Sorting I

There are three repetitive steps involved in sorting the keys:

- Determine the value of the key in the first position of the heap (i.e., the smallest value).
- Move the largest value in the heap (last heap element) into the first position, and decrease the number of elements by one. At this point, the heap is out of order.
- Reorder the heap by exchanging the largest element with the smaller of its children and moving down the tree to the new position of the largest element until the heap is back in order.

# Heap Sorting II

Remove()

- `val=HeapArray[1];`
- `HeapArray[1]=HeapArray[NumElements];`
- `NumElements--;`
- `int k=1; int newK;`
- `while (2*k <= NumElements) {`
  - `if (Compare(2*k, 2*k+1)) < 0) newK=2*k; else newK=2*k+1;`
  - `if (Compare(k, newK) <0) break;`
  - `Exchange(k,newK);`
  - `k=newK;}`
- `return val;}`

# Heap Sorting while Writing to the File

- The smallest record in the heap is known during the first step of the sorting algorithm. Therefore, it can be buffered until a whole block is known.
- While that block is written onto the disk a new block can be processed and so on.
- Since every time a block can be written to disk, the heap size decreases by one block, that block can be used as a buffer. i.e., we can have as many output buffers as there are blocks in the file.
- Since all the I/O is sequential, this algorithm works as well with disks and tapes. As well, a minimum amount of seeking is necessary and thus the procedure is efficient.

# An Efficient way of Sorting

## Large Files on Disks: MergeSort

- A solution for this problem was previously presented in the form of the Keysort algorithm. However, Keysort has two shortcomings:
  - Once the key were sorted, it was expensive to seek each record in sorted order and then write them to the new, sorted file.
  - If the file contains many records, even the index is too large to fit in memory.
- **Solution:** (1) Break the file into several sorted subfiles (runs), using an internal sorting method; and (2) merge the runs. ==> MergeSort

# MergeSort: Advantages

- It can be applied to files of any size.
- Reading of the input during the run-creation step is sequential ==> Not much seeking.
- Reading through each run during merging and writing the sorted record is also sequential. The only seeking necessary is as we switch from run to run.
- If heapsort is used for the in-memory part of the merge, its operation can be overlapped with I/O
- Since I/O is largely sequential, tapes can be used.

# How much Time does a MergeSort take?

## *Simplifying assumptions:*

- Only one seek is required for any single sequential access.
- Only one rotational delay is required per access.

## *Expensive steps (i.e. involving I/O) occurring in MergeSort*

- During the *sort phase*:
  - Reading all records into memory for sorting and forming runs.
  - Writing sorted runs to disk
- During the *merge phase*:
  - Reading sorted runs into memory for merging.
  - Writing sorted file to disk.

# What kinds of I/O take place during the Sort and the Merge phases?

- Since, during the sort phase, the runs are created using heapsort, I/O is sequential. No performance improvement can ever be gained in this phase.
- During the reading step of the merge phase, there are a lot of random accesses (since the buffers containing the different runs get loaded and reloaded at unpredictable times). The number and size of the memory buffers holding the runs determine the number of random accesses. Performance improvements can be made in this step.
- The write step of the merge phase, is not influenced by the way in which we organize the runs.

# The Cost of Increasing the File Size

- In general, for a K-way merge of K runs where each run is as large as the memory space available, the buffer size for each of the runs is:  
 $(1/K) * \text{size of memory space} = (1/K) * \text{size of each run}$ .
- So K seeks are required to read all of the records in each individual run and since there are K runs altogether, the merge operation requires  $K^2$  seeks.
- Since K is directly proportional to N, the number of records, SortMerge is an  $O(N^2)$  operation, measures in terms of seeks.

# What can be done to Improve MergeSort Performance?

There are different ways in which MergeSort's efficiency can be improved:

- Allocate more Hardware such as disk drives, memory, and I/O channels.
- Perform the merge in more than one step, reducing the order of each merge and increasing the buffer size for each run.
- Algorithmically increase the lengths of the initial sorted runs.
- Find ways to overlap I/O Operations.

# Hardware-Based Improvements

- **Increasing the amount of memory**: helps make the buffers larger and thus reduce the numbers of seeks.
- **Increasing the Number of Dedicated Disk Drives**: If we had one separate read/write head for every run, then no time would be wasted seeking.
- **Increasing the Number of I/O Channels**: With a single I/O Channel, no two transmission can occur at the same time. But if there is a separate I/O Channel for each disk drive, then I/O can overlap completely.
- But what if hardware based improvements are not possible?

# Decreasing the Number of Seeks

## Using Multiple-Step Merges

- The expensive part of the MergeSort algorithm is related to all the seeking performed during the reading step of the merge phase. A lot of seeks are involved because of the large number of runs that get merged simultaneously.
- In multi-step merging, we do not try to merge all runs at one time. Instead, we break the original set of runs into small groups and merge the runs in these groups separately. More buffer space is available for each run, and, therefore, fewer seeks are required per run).
- When all the smaller merges are completed, a second pass merges the new set of merged runs.

# Increasing Run Lengths Using Replacement Selection

## Replacement Selection Procedure:

- Read a collection of records and sort them using heapsort. The resulting heap is called the primary heap.
- Instead of writing the entire primary heap in sorted order, write only the record whose key has the lowest value.
- Bring in a new record and compare the values of its key with that of the key that has just been output.
  - If the new key value is higher, insert the new record into its proper place in the primary heap along with the other records that are being selected for output.
  - If the new record's key value is lower, place the record in a secondary heap of records with key values smaller than those already written.
- Repeat Step 3 as long as there are records left in the primary heap and there are records to be read. When the primary heap is empty, make the secondary heap into the primary heap and repeat steps 2 and 3.

# Analysis of Run Length Selection

- **Question 1**: Given  $P$  locations in memory, how long a run can we expect replacement selection to produce on average?
- **Answer 1**: On average we can expect a run length of  $2P$ .
- **Question 2**: What are the costs of using replacement selection?
- **Answer 2**: Replacement Selection requires much more seeking in order to form the runs. However, the reduction in the number of seeks required to merge the runs usually more than offsets that extra cost.

# Replacement Selection + MultiStep Merging

- In practice, Replacement Selection is not used with a one-step merge procedure.
- Instead, it is usually used in a two-step merge process.
- The reduction in total seek and rotational delay time is most affected by the move from one-step to two-step merges, but the use of Replacement Selection is also somewhat useful.

# Using Two Disk Drives with Replacement Selection

- Replacement Selection offers an opportunity to save on both transmission and seek times in ways that memory sort methods do not.
- We could use one disk drive to do only input operations and the other one to do only output operations.
- This means that:
  - Input and Output can overlap  $\implies$  Transmission time can be decreased by up to 50%.
  - Seeking is virtually eliminated.

# More Drives? More Processor?

- We can make the I/O process even faster by using more than two disk drives.
- If I/O becomes faster than processing, then more processors can be used. Different network architectures can be used for that:
  - Mainframe computers
  - Vector and Array processors
  - Massively parallel machines
  - Very fast local area networks and communication software.

# Data and File Structures

## Multi-Level Indexing and B-Trees

# Statement of the Problem

- When indexes grow too large they have to be stored on secondary storage.
- However, there are two fundamental problems associated with keeping an index on secondary storage:
  - Searching the index must be faster than binary searching.
  - Insertion and deletion must be as fast as search.

# Indexing with Binary Search

## Trees: Negative Aspects

- A sorted list can be expressed in a *Binary Search Tree* representation.
- However, there are 2 problems with binary search trees:
  - They are not fast enough for disk resident indexing.
  - There is no effective strategy of balancing the tree.
- ▲ We will look at 2 solutions: *AVL Trees* and *Paged Binary Trees*.

# Indexing with Binary Search

## Trees: Positive Aspects

- Tree structures give us an important new capability: we no longer have to sort the file to perform a binary search.
  - To add a new key, we simply link it to the appropriate leaf node.
  - If the tree remains *balanced*, then the search performance on this tree is good.
  - Problems occur when the tree gets *unbalanced*.
- ☞ We will look for schemes that allow trees to remain balanced

# AVL Trees I

- AVL Trees allow us to re-organize the nodes of the tree as we receive new keys, maintaining a near optimal tree structures.
- An AVL Tree is a height-balanced tree, i.e., a tree that places a limit on the amount of difference allowed between the heights of any two sub-trees sharing a common root.
- In an AVL or HB-1 tree, the maximum allowable difference is one.

# AVL Trees II

- The two features that make AVL trees important are:
  - By setting a maximum allowable difference in the height of any two sub-trees, AVL trees guarantee a minimum level of performance in searching.
  - Maintaining a tree in AVL form as new nodes are inserted involves the use of one of a set of four possible rotations. Each of the rotations is confined to a single local area of the tree. The most complex of the rotations requires only five pointer reassignments.

# AVL Tree III

- AVL Trees are not, themselves, directly applicable to most file structures because like all strictly binary trees, they have too many levels--they are too deep.
- AVL Trees, however, are important because they suggest that it is possible to define procedures that maintain height-balance.
- AVL Trees' search performance approximates that of a *completely balanced tree*. For a completely balanced tree, the worst-case search to find a key is  $\log_2(N+1)$ . For an AVL Tree it is  $1.44 \text{ Log}_2(N+2)$ .

# Paged Binary Trees

- AVL trees tackle the problem of keeping an index in sorted order cheaply. They do not address the problem regarding the fact that Binary Searching requires too many seeks.
- Paged Binary trees addresses this problem by locating multiple binary nodes on the same disk page.
- In a paged system, you do not incur the cost of a disk seek just to get a few bytes. Instead, once you have taken the time to seek to an area of the disk, you read in an entire page from the file.
- When searching a Binary Tree, the number of seeks necessary is  $\log_2(N+1)$ . It is  $\log_{k+1}(N+1)$  in the paged version.

# Problems with Paged Trees I

- Inefficient Disk Usage
- How should we build a paged tree?
  - Easy if we know what the keys are and their order before starting to build the tree.
  - Much more difficult if we receive keys in random order and insert them as soon as we receive them. The problem is that the wrong keys may be placed at the root of the trees and cause an imbalance.

# Problems with Paged Trees II

- Three problems arise with paged trees:
  - How do we ensure that the keys in the root page turn out to be good separator keys, dividing up the set of other keys more or less evenly.
  - How do we avoid grouping keys that shouldn't share a page?
  - How can we guarantee that each of the pages contains at least some minimum number of keys?

# Multi-Level Indexing: A Better Approach to Tree Indexes

- Up to this point, in this lecture, we've looked at indexing a file based on building a search tree. However, there are problems with this approach (see previous slide).
- Instead, we get back to the notion of the simple indexes we saw earlier in the course, but we extend this notion to that of multi-record indexes and then, multi-level indexes.
- While multi-record multi-level indexes really help reduce the number of disk accesses and their overhead space costs are minimal, inserting a new key or deleting an old one is very costly.

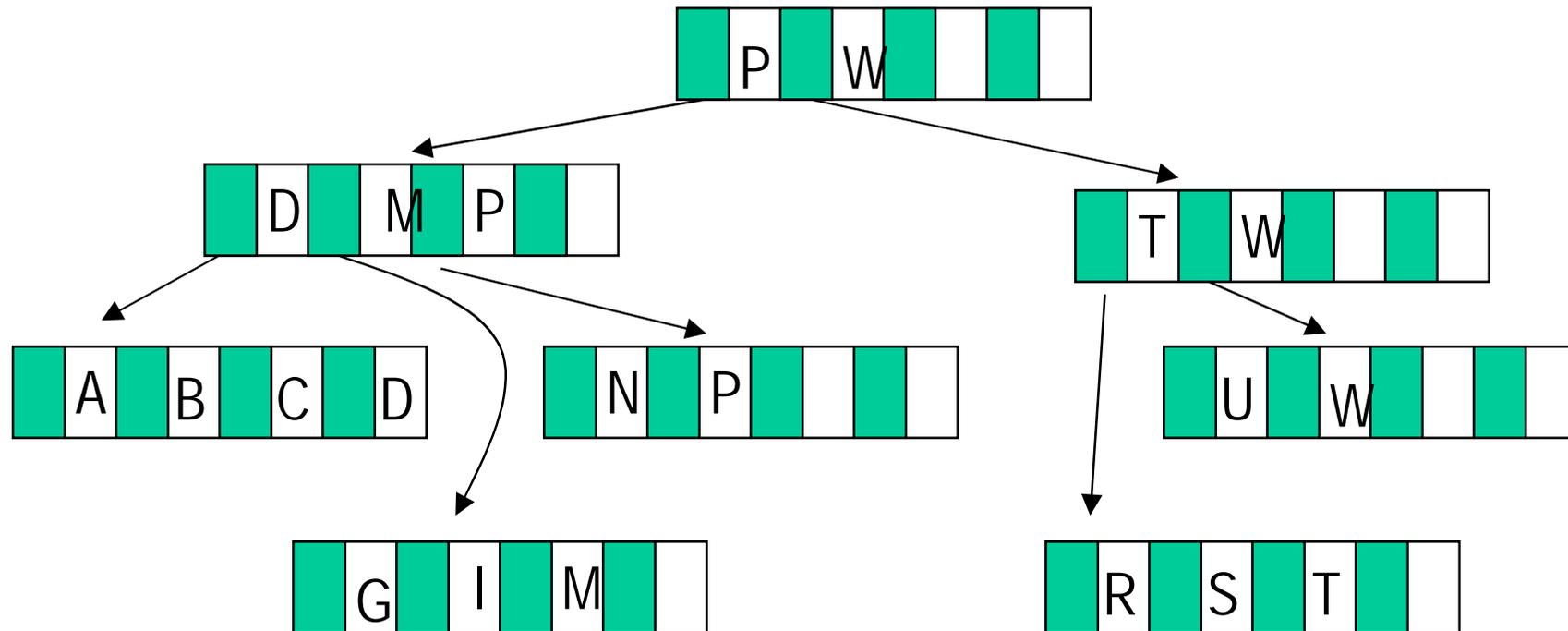
# B-Trees: Addressing the problems of Paged Trees and Multi-Level Indexing

- *Trees* appear to be a good general solution to indexing, but each particular solution we've looked at so far presents some problems.
- *Paged Trees* suffer from the fact that they are built downward from the top and that a "bad" root may unbalance the construct.
- *Multi-Level Indexing* takes a different approach that solves many problems but creates costly insertion and deletion.
- An ideal solution would be one that combines the advantages of the previous solutions and does not suffer from their disadvantages.
- *B-Trees* appear to do just that!

# B-Trees: An Overview

- B-Trees are built *upward from the bottom* rather than downward from the top, thus addressing the problems of Paged Trees: with B-Trees, we allow the root to emerge rather than set it up and then find ways to change it.
- B-Trees are multi-level indexes that solve the problem of linear cost of insertion and deletion.
- B-Trees are now the standard way to represent indexes.

# Example of a B-Tree



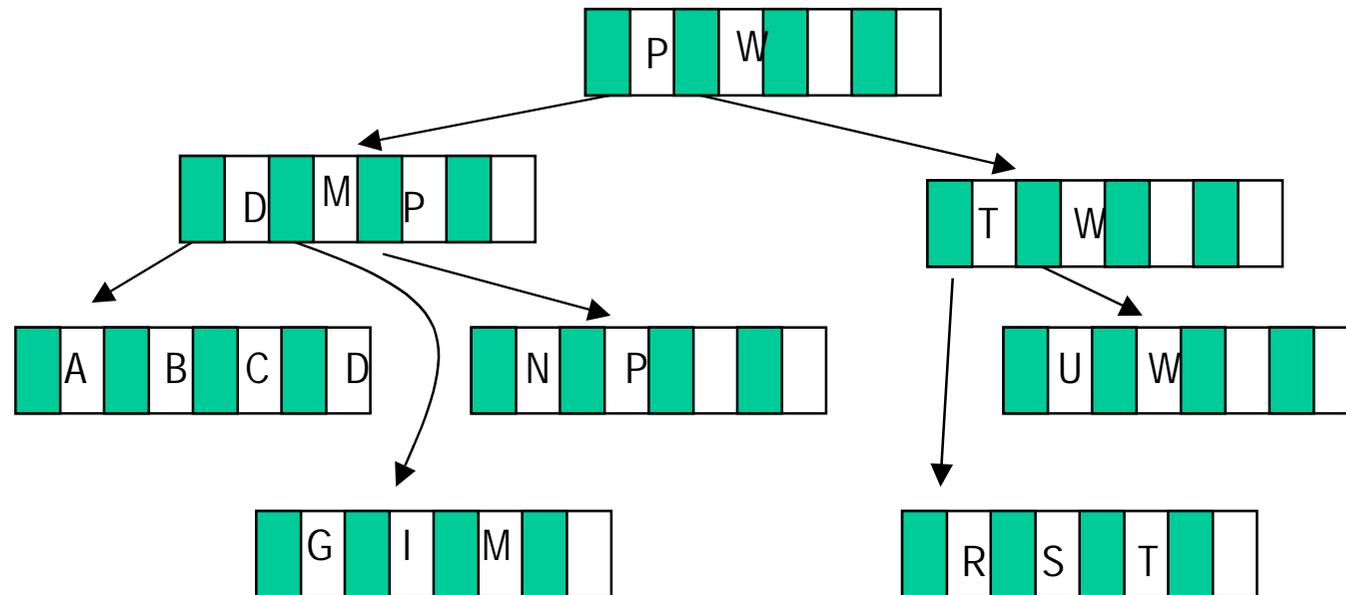
**Note:** references to actual record only occur in the leaf nodes.  
The interior nodes are only higher level indexes (this is why there are duplications in the tree)

# How do B-Trees work?

## Main Ideas

- Each node of a B-Tree is an *Index Record*. Each of these records has the same *maximum* number of key-reference pairs called the *order* of the B-Tree. The records also have a *minimum* number of key-reference pairs, typically, half the order.
- When inserting a new key into an index record that is not full, we simply need to update that record and possibly go up the tree recursively.
- When inserting a new key into an index record that is full, this record is *split* into two, each with half of the keys. The largest key of the split record is promoted which may cause a new recursive split.

# Searching a B-Tree



- *Problem 1*: Look for L
- *Problem 2*: Look for S

# Insertion into a B-Tree: General Strategy

- Search all the way down to the leaf level in order to find the insertion location.
- Insertion, overflow detection, and splitting on the upward path.
- Creation of a new root node if the current root was split.

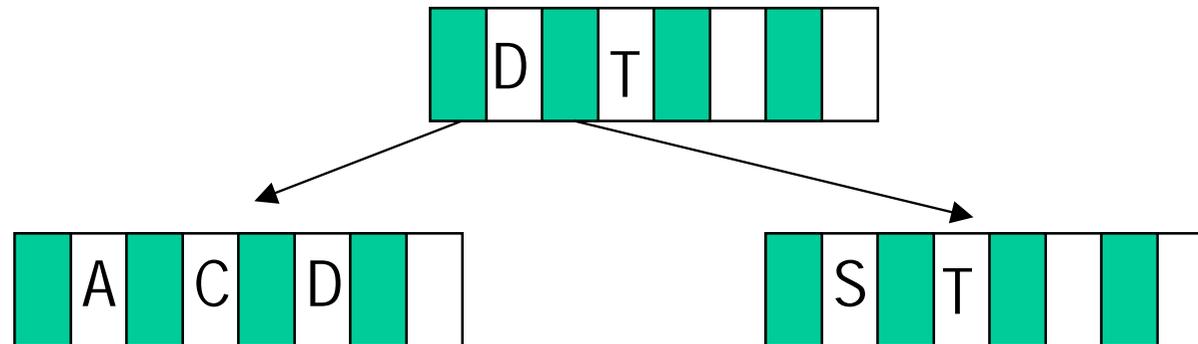
# Insertion into a B-Tree: No Split & Contained Splits

After inserting C, S, D, T: 

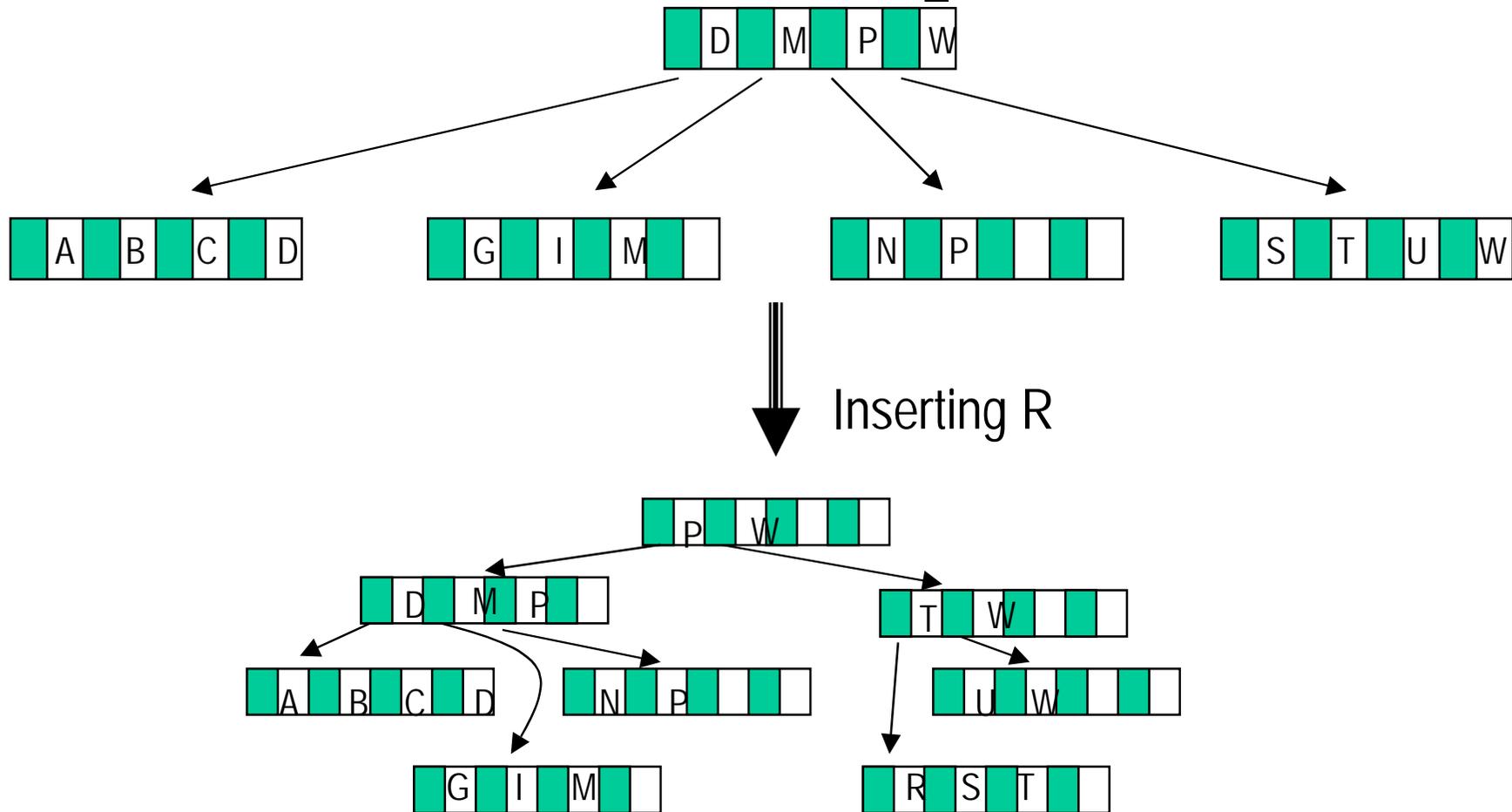
	C		D		S		T
--	---	--	---	--	---	--	---



Inserting A



# Insertion into a B-Tree: Recursive Split



# Formal Definition of B-Tree

## Properties

In a B-Tree of order  $m$ ,

- Every page has a maximum of  $m$  descendants
- Every page, except for the root and leaves, has at least  $m/2$  descendants.
- The root has at least two descendants (unless it is a leaf).
- All the leaves appear on the same level.
- The leaf level forms a complete, ordered index of the associated data file.

# Worst-Case Search Depth I

- Given 1,000,000 keys and a B-Tree of order 512, what is the maximum number of disk accesses necessary to locate a key in the tree? In other words, how deep will the tree be?
- Each key appears in the leaf  $\implies$  What is the maximum height of a tree with 1,000,000 leaves?
- The maximum height will be reached if all pages (or nodes) in the tree has the minimum allowed number of descendants
- For a B-Tree of order  $m$ , the minimum number of descendants from the root page is 2. It is  $\lceil m/2 \rceil$  for all the other pages.

# Worst-Case Search Depth II

- For any level  $d$  of a B-Tree, the minimum number of descendants extending from that level is

$$2 \lceil m/2 \rceil^{d-1}$$

- For a tree with  $N$  keys in its leaves, we have

$$N \leq 2 \lceil m/2 \rceil^{d-1}$$

$$\hat{U} \ d \leq 1 + \log_{\lceil m/2 \rceil} (N/2)$$

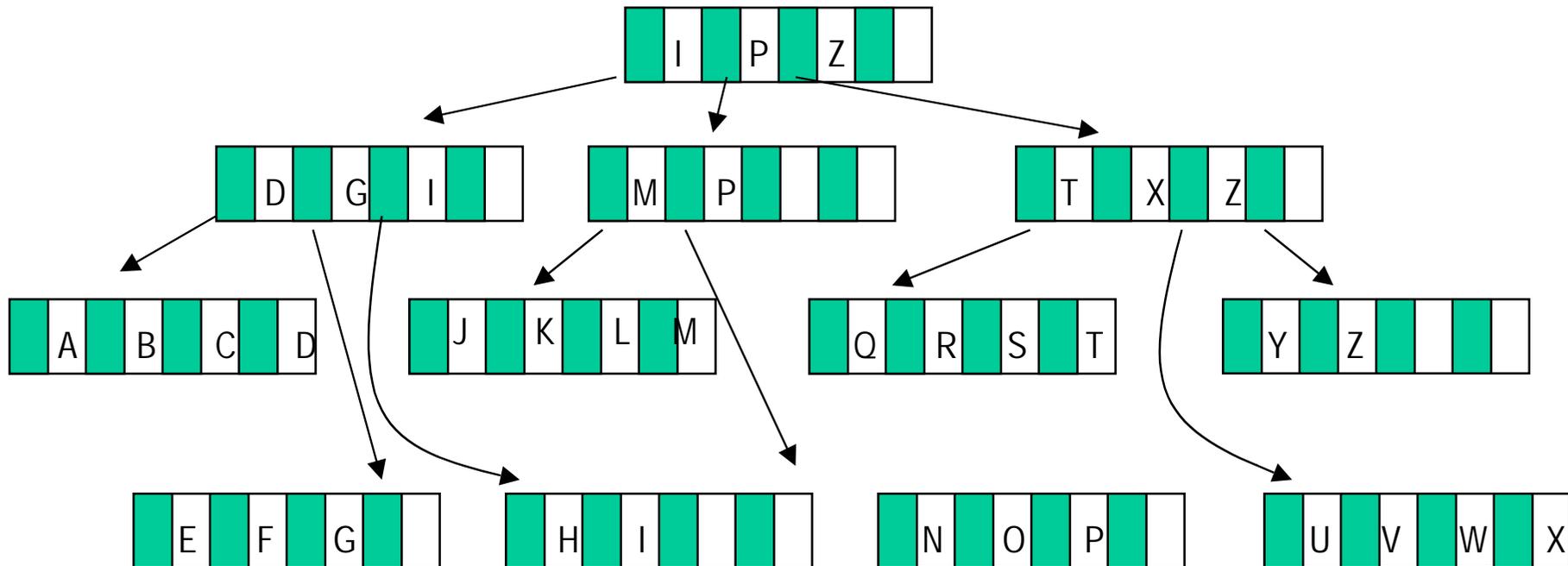
- For  $m=512$  and  $N=1,000,000$ , we thus get

$$d \leq 3.37$$

# Deletion from a B-Tree: Rules for Deleting a key $k$ from a node

- If  $n$  has more than the number  <sup>$n$</sup>  of keys and the  $k$  is not the largest in  $n$ , simply delete  $k$  from  $n$ .
- If  $n$  has more than the minimum number of keys and the  $k$  is the largest in  $n$ , delete  $k$  and modify the higher level indexes to reflect the new largest key in  $n$ .
- If  $n$  has exactly the minimum number of keys and one of the siblings of  $n$  has few enough keys, *merge*  $n$  with its sibling and delete a key from the parent node.
- If  $n$  has exactly the minimum number of keys and one of the siblings of  $n$  has extra keys, *redistribute* by moving some keys from a sibling to  $n$ , and modify the higher level indexes to reflect the new largest keys in the affected nodes.

# Deletion from a B-Tree: Example



- **Problem 1**: Delete C
- **Problem 2**: Delete P
- **Problem 3**: Delete H

# Redistribution during Insertion

- Redistribution during insertion is a way to avoid, or at least postpone, the creation of new pages.
- Redistribution allows us to place some of the overflowing keys into another page instead of splitting an overflowing page.
- B\* Trees formalize this idea

# Properties of a B\* Tree

- Every page has a maximum of  $m$  descendants.
- Every page except for the root has at least  $\lceil (2m-1)/3 \rceil$  descendants.
- The root has at least two descendants (unless it is a leaf)
- All the leaves appear on the same level.

The main difference between a B-Tree and a B\* Tree is in the second rule.

# Data and File Structures

Indexed Sequential File Access  
and Prefix B+ Trees

# Indexed Sequential Access

- Up to this point, we have had to choose between viewing a file from an indexed point of view or from a sequential point of view.
- Here, we are looking for a single organizational method that provides both of these views simultaneously.
- Why care about obtaining both views simultaneously? If an application requires both interactive random access and cosequential batch processing, both sets of actions have to be carried out efficiently. (E.g., a student record system at a University).

# Maintaining a Sequence Set: The Use of Blocks I

- A *sequence set* is a set of records in physical key order which is such that it stays ordered as records are added and deleted.
- Since sorting and resorting the entire sequence set as records are added and deleted is expensive, we look at other strategies. In particular, we look at a way to *localize* the changes.
- The idea is to use blocks that can be read into memory and rearranged there quickly. Like in B-Trees, blocks can be *split*, *merged* or their records *re-distributed* as necessary.

# Maintaining a Sequence Set: The Use of Blocks II

- Using blocks, we can thus keep a sequence set in order by key without ever having to sort the entire set of records.
- However, there are certain costs associated with this approach:
  - A Blocked file takes up more space than an unblocked file because of *internal fragmentation*.
- The order of the records is not necessarily *physically* sequential throughout the file. The maximum guaranteed extent of physical sequentiality is within a block.

# Maintaining a Sequence Set: The Use of Blocks III

- An important aspect of using blocks is the choice of a block size. There are 2 considerations to keep in mind when choosing a block size:
  - The block size should be such that we can hold several blocks in memory at once
  - The block size should be such that we can access a block without having to bear the cost of a disk seek within the block read or block write operation.

# Adding a Simple Index to the Sequence Set

- Each of the blocks we created for our Sequence Set contains a range of records that might contain the record we are seeking.
- We can construct a simple single-level index for these blocks.
- The combination of this kind of index with the sequence set of blocks provides complete indexed sequential access. This method works well as long as the entire index can be held in memory.
- If the entire index cannot be held in memory, then we can use a B+ Tree which is a B-Tree index plus a sequence set that holds the records.

# The Content of the Index: Separators Instead of Keys

- The index serves as a kind of road map for for the sequence set  $\implies$  We do not need to have keys in the index set.
- What we really need are separators capable of distinguishing between two blocks.
- We can save space by using variable-length separators and placing the shortest separator in the index structure.
- Rules are:  $\text{Key} < \text{separator} \implies$  Go left  
 $\text{Key} = \text{separator} \implies$  Go right  
 $\text{Key} > \text{separator} \implies$  Go right

# The Simple Prefix B+ Tree

- The separators we just identified can be formed into a B-Tree index of the sequence set blocks and the B-Tree index is called the *index set*.
- Taken together with the sequence set, the index set forms a file structure called a *simple prefix B+ Tree*.
- “simple prefix” indicates that the index set contains shortest separators, or prefixes of the keys rather than copies of the actual keys.

# Simple Prefix B+ Tree

## Maintenance

- Changes localized to single blocks in the sequence set:  
Make the changes to the sequence set and to the index set.
- Changes involving multiple blocks in the sequence set:
  - If blocks are split in the sequence set, a new separator must be inserted into the index set
  - If blocks are merged in the sequence set, a separator must be removed from the index set.
  - If records are re-distributed between blocks in the sequence set, the value of a separator in the index set must be changed.

# Index Set Block Size

- The physical size of a node for the index set is usually the same as the physical size of a block in the sequence set. We, then, speak of index set *blocks*, rather than nodes.
- There are a number of reasons for using a common block size for the index and sequence sets:
  - The block size for the sequence set is usually chosen because there is a good fit among this block size, the characteristics of the disk drive, and the amount of memory available.
  - A common block size makes it easier to implement a buffering scheme to create a virtual simple prefix B+Tree
  - The index set blocks and sequence set blocks are often mingled within the same file to avoid seeking between 2 separate files while accessing the simple prefix B+Tree.

# Internal Structure of Index Set Blocks: A Variable-Order B-Tree

- Given a large, fixed-size block for the index set, how do we store the separators within it?
- There are many ways to combine the list of separators, the index to separators, and the list of Relative Block Numbers (RBNs) into a single index set block.
- One possible approach includes a separator count and keeps a count of the total length of separators.

# Loading a Simple Prefix B+ Tree I

- Successive Insertions is not a good method because splitting and redistribution are relatively expensive and would be best to use only for tree maintenance.
- Starting from a sorted file, however, we can place the records into sequence set blocks one by one, starting a new block when the one we are working with fills up. As we make the transition between two sequence set blocks, we can determine the shortest separator for the blocks. We can collect these separators into an index set block that we build and hold in memory until it is full.

# Loading a Simple Prefix B+ Tree II: Advantages

- The advantages of loading a simple Prefix B+ Tree almost always outweigh the disadvantages associated with the possibility of creating blocks that contain too few records or too few separators.
- A particular advantage is that the loading process goes more quickly because:
  - The output can be written sequentially;
  - we make only one pass over the data;
  - No blocks need to be reorganized as we proceed.
- Advantages after the tree is loaded
  - The blocks are 100% full.
  - Sequential loading creates a degree of *spatial locality* within our file ==> Seeking can be minimized.

# B+ Trees

- The difference between a simple prefix B+ Tree and a plain B+ Tree is that the plain B+ Tree does not involve the use of prefixes as separators. Instead, the separators in the index set are simply copies of the actual keys.
- Simple Prefix B+ Tree are often more desirable than plain B+ Trees because the prefix separators take up less space than the full keys.
- B+ Trees, however, are sometimes more desirable since 1) they do not need variable length separator fields and 2) some key sets are not always easy to compress effectively.

# B-Trees, B+Trees and Simple Prefix B+ Trees in Perspective I

- B and B+ Trees are not the only tools useful for File Structure Design. Simple Indexes are useful when they can be held fully into memory and Hashing can provide much faster access than B and B+ Trees.
- Common Characteristics of B and B+ and Prefix B+ Trees:
  - Paged Index Structures ==> Broad and shallow trees
  - Height-Balanced Trees
  - The trees are grown Bottom Up and the operations used are: block splitting, merging and re-distribution
  - Two-to-Three Splitting and redistribution can be used to obtain greater storage efficiency.
  - Can be implemented as Virtual Tree Structures.
  - Can be adapted for use with variable-length records.

# B-Trees, B+Trees and Simple Prefix B+ Trees in Perspective II

## Differences between the various structures:

- **B-Trees:** multi-level indexes to data files that are entry-sequenced. **Strengths:** simplicity of implementation. **Weaknesses:** excessive seeking necessary for sequential access.
- **B-Trees with Associated Information:** These are B-Trees that contain record contents at every level of the B-Tree. Strengths: can save up space. Weaknesses: Works only when the record information is located within the B-Tree. Otherwise, too much seeking is involved in retrieving the record information.

# B-Trees, B+Trees and Simple Prefix B+ Trees in Perspective III

## *Differences between the various structures (Cont'd):*

- *B+ Trees:* In a B+ Tree all the key and record info is contained in a linked set of blocks known as the sequence set. Indexed access is provided through the Index Set. Advantages over B-Trees: 1) The sequence set can be processed in a truly linear, sequential way; 2) The index is built with a single key or separator per block of data records rather than with one key per data record. ==> index is smaller and hence shallower.
- *Simple Prefix B+ Trees:* The separators in the index set are smaller than the keys in the sequence set ==> Tree is even smaller.

# Data and File Structures

Hashing

# Motivation

- Sequential Searching can be done in  $O(N)$  access time, meaning that the number of seeks grows in proportion to the size of the file.
- B-Trees improve on this greatly, providing  $O(\log_k N)$  access where  $k$  is a measure of the leaf size (i.e., the number of records that can be stored in a leaf).
- What we would like to achieve, however, is an  $O(1)$  access, which means that no matter how big a file grows, access to a record always takes the same small number of seeks.
- *Static Hashing* techniques can achieve such performance provided that the file does not increase in time.

# What is Hashing?

- A *Hash function* is a function  $h(K)$  which transforms a key  $K$  into an address.
- Hashing is like indexing in that it involves associating a key with a relative record address.
- Hashing, however, is different from indexing in two important ways:
  - With hashing, there is no obvious connection between the key and the location.
  - With hashing two different keys may be transformed to the same address.

# Collisions

- When two different keys produce the same address, there is a *collision*. The keys involved are called *synonyms*.
- Coming up with a hashing function that avoids collision is extremely difficult. It is best to simply find ways to deal with them.
- Possible Solutions:
  - Spread out the records
  - Use extra memory
  - Put more than one record at a single address.

# A Simple Hashing Algorithm

- **Step 1:** Represent the key in numerical form
- **Step 2:** Fold and Add
- **Step 3:** Divide by a prime number and use the remainder as the address.

# Hashing Functions and Record Distributions

- Records can be distributed among addresses in different ways: there may be (a) no synonyms (uniform distribution); (b) only synonyms (worst case); (c) a few synonyms (happens with random distributions).
- Purely uniform distributions are difficult to obtain and may not be worth searching for.
- Random distributions can be easily derived, but they are not perfect since they may generate a fair number of synonyms.
- We want better hashing methods.

# Some Other Hashing Methods

- Though there is no hash function that guarantees better-than-random distributions in all cases, by taking into considerations the keys that are being hashed, certain improvements are possible.
- Here are some methods that are potentially better than random:
  - Examine keys for a pattern
  - Fold parts of the key
  - Divide the key by a number
  - Square the key and take the middle
  - Radix transformation

# Predicting the Distribution of Records

- When using a random distribution, we can use a number of mathematical tools to obtain conservative estimates of how our hashing function is likely to behave:
- Using the Poisson Function  $p(x) = (r/N)^x e^{-(r/N)} / x!$  applied to Hashing, we can conclude that:
- In general, if there are  $N$  addresses, then the expected number of addresses with  $x$  records assigned to them is  $Np(x)$

# Predicting Collisions for a Full File

- Suppose you have a hashing function that you believe will distribute records randomly and you want to store 10,000 records in 10,000 addresses.
- How many addresses do you expect to have no records assigned to them?
- How many addresses should have one, two, and three records assigned respectively?
- How can we reduce the number of overflow records?

# Increasing Memory Space I

- Reducing collisions can be done by choosing a good hashing function or using extra memory.
- The question asked here is how much extra memory should be use to obtain a given rate of collision reduction?
- **Definition: Packing density** refers to the ratio of the number of records to be stored ( $r$ ) to the number of available spaces ( $N$ ).
- The packing density gives a measure of the amount of space in a file that is used.

# Increasing Memory Space II

- The *Poisson Distribution* allows us to predict the number of collisions that are likely to occur given a certain packing density. We use the Poisson Distribution to answer the following questions:
- How many addresses should have no records assigned to them?
- How many addresses should have exactly one record assigned (no synonym)?
- How many addresses should have one record plus one or more synonyms?
- Assuming that only one record can be assigned to each home address, how many overflow records can be expected?
- What percentage of records should be overflow records?

# Collision Resolution by Progressive Overflow

- How do we deal with records that cannot fit into their home address? A simple approach: *Progressive Overflow* or *Linear Probing*.
- If a key,  $k_1$ , hashes into the same address,  $a_1$ , as another key,  $k_2$ , then look for the first available address,  $a_2$ , following  $a_1$  and place  $k_1$  in  $a_2$ . If the end of the address space is reached, then wrap around it.
- When searching for a key that is not in, if the address space is not full, then an empty address will be reached or the search will come back to where it began.

# Search Length when using Progressive Overflow

- Progressive Overflow causes extra searches and thus extra disk accesses.
- If there are many collisions, then many records will be far from “home”.
- **Definitions:** **Search length** refers to the number of accesses required to retrieve a record from secondary memory. The **average search length** is the average number of times you can expect to have to access the disk to retrieve a record.
- ***Average search length = (Total search length)/(Total number of records)***

# Storing More than One Record per Address: Buckets

- **Definition:** A **bucket** describes a block of records sharing the same address that is retrieved in one disk access.
- When a record is to be stored or retrieved, its home bucket address is determined by hashing. When a bucket is filled, we still have to worry about the record overflow problem, but this occurs much less often than when each address can hold only one record.

# Effect of Buckets on Performance

- To compute how densely packed a file is, we need to consider 1) the number of addresses,  $N$ , (buckets) 2) the number of records we can put at each address,  $b$ , (bucket size) and 3) the number of records,  $r$ . Then, ***Packing Density*** =  $r/bN$ .
- Though the packing density does not change when halving the number of addresses and doubling the size of the buckets, the expected number of overflows decreases dramatically.

# Making Deletions

- Deleting a record from a hashed file is more complicated than adding a record for two reasons:
  - The slot freed by the deletion must not be allowed to hinder later searches
  - It should be possible to reuse the freed slot for later additions.
- In order to deal with deletions we use *tombstones*, i.e., a marker indicating that a record once lived there but no longer does. Tombstones solve both the problems caused by deletion.
- Insertion of records is slightly different when using tombstones.

# Effects of Deletions and Additions on Performance

- After a large number of deletions and additions have taken place, one can expect to find many tombstones occupying places that could be occupied by records whose home address precedes them but that are stored after them.
- This deteriorates average search lengths.
- There are 3 types of solutions for dealing with this problem: a) local reorganization during deletions; b) global reorganization when the average search length is too large; c) use of a different collision resolution algorithm.

# Other Collision Resolution Techniques

- There are a few variations on random hashing that may improve performance:
  - **Double Hashing**: When an overflow occurs, use a second hashing function to map the record to its overflow location.
  - **Chained Progressive Overflow**: Like Progressive overflow except that synonyms are linked together with pointers.
  - **Chaining with a Separate Overflow Area**: Like chained progressive overflow except that overflow addresses do not occupy home addresses.
  - **Scatter Tables**: The Hash file contains no records, but only pointers to records. I.e., it is an index.

# Pattern of Record Access

- If we have some information about what records get accessed most often, we can optimize their location so that these records will have short search lengths.
- By doing this, we try to decrease the effective average search length even if the nominal average search length remains the same.
- This principle is related to the one used in Huffman encoding.

**THE END**

For more info visit

<http://engginfo2002.tripod.com>