University of Dublin
Trinity College

# File Organisation

Owen.Conlan@cs.tcd.ie

# Unordered Records: Heap/Pile/Serial Files

New records are inserted at the end of the file

To search for a record

- *linear search* through the file records is necessary
- Typically requires reading and searching half the file blocks on average, and is hence quite expensive

Record insertion is quite efficient

- Last block read into a buffer, new block added, buffer rewritten back to disk

# Unordered Records: Heap/Pile/Serial Files

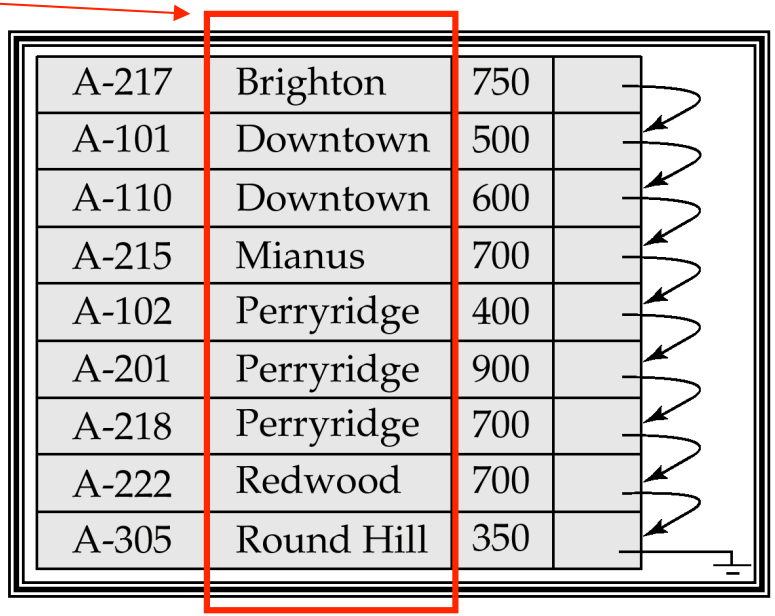
Record deletion requires a search operation first

- Approach #1: Appropriate block read into buffer, record deleted, block rewritten
- Approach#2: Appropriate block read into buffer, record flagged as deleted but not physically removed as yet. In future only non-deleted flagged records are searched
- Can lead to unused space

Both cases require periodic file reorganisation to reclaim unused space

# Ordered Records: Sequential File

Store records in sequential order based on the value of a **single** ordering field in each record



| | | | |
|---|---|---|---|
| A-217 | Brighton | 750 | |
| A-101 | Downtown | 500 | |
| A-110 | Downtown | 600 | |
| A-215 | Mianus | 700 | |
| A-102 | Perryridge | 400 | |
| A-201 | Perryridge | 900 | |
| A-218 | Perryridge | 700 | |
| A-222 | Redwood | 700 | |
| A-305 | Round Hill | 350 | |

If the ordering field corresponds to key field (that is guaranteed unique) of the file then it is called the ordering key

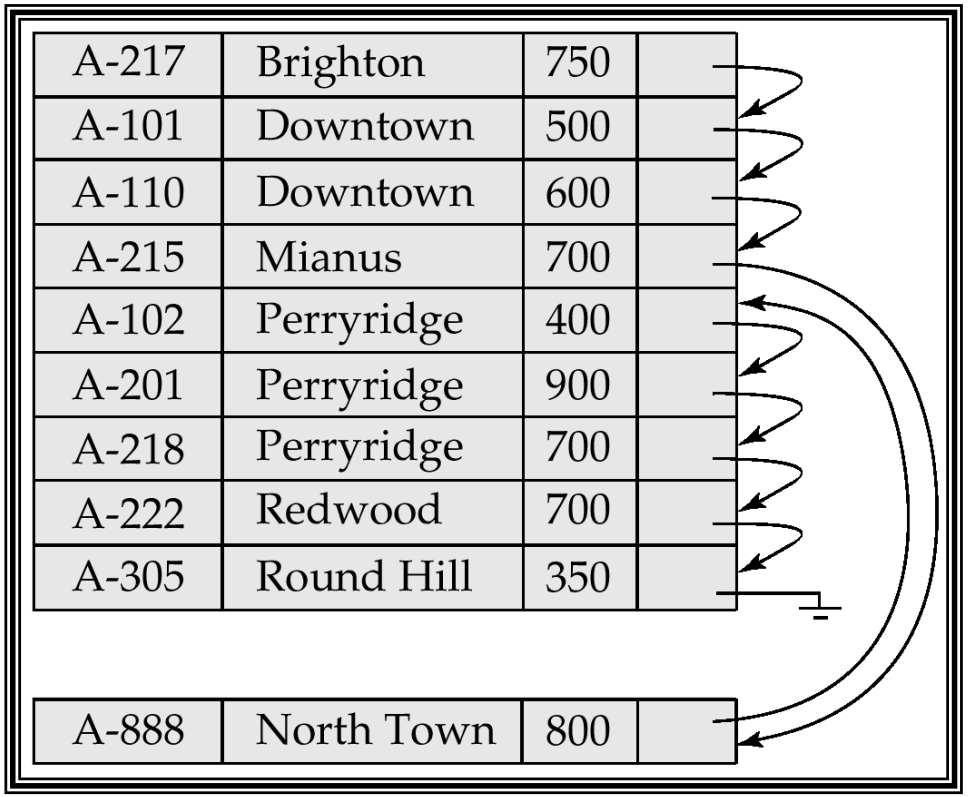
# Operations on Sequential Files

## Record Insertion

- Locate position for insertion
- If free block space then insert
  - *Records stored in sequence of blocks means less disk seeks*
- If not, then insert in overflow block
- Periodically reorganise to restore contiguous blocks

## Record Deletion

- Mark record invalid
- Periodically reorganise to remove deleted records



# Sequential File Processing

Update (insertion, deletion, modification) is expensive

- records must be inserted in the *correct order*

It is common to keep a separate unordered overflow/ transaction file for new records to improve update efficiency; this is periodically merged with the master file

Normally processing done in Batch Mode

- Transaction file first ordered on same ordering field as master
- Transaction file and Master file can then be processed sequentially
- How often depends on rate of change of data; size of master file and need for current data in master file

# Master File Update

### Master File

| Key | Name | Subj | Year |
|---|---|---|---|
| 21023 | Jones | Maths | JS |
| 24019 | Webster | CS | JF |
| 30987 | Carter | Biol | SF |
| 31290 | Reed | Maths | SF |
| 37400 | Dyson | CS | JF |

### Transaction File

| Key | Type | Parameters | | |
|---|---|---|---|---|
| 21023 | I | White | Maths | JS |
| 24019 | M | 3 | Maths | |
| 24750 | I | Kelly | Biol | SS |
| 29974 | D | | | |
| 31290 | M | 4 | JS | |
| 37400 | D | | | |
| 37512 | I | Green | CS | JF |

### Updated Master File

| Key | Name | Subj | Year |
|---|---|---|---|
| 21023 | Jones | Maths | JS |
| 24019 | Webster | Maths | JF |
| 24750 | Kelly | Biol | SS |
| 30987 | Carter | Biol | SF |
| 31290 | Reed | Maths | JS |
| 37512 | Green | CS | JF |

### Log File

| Key | Error message |
|---|---|
| 21023 | Insert error - record already exists |
| 29974 | Delete error - nonexistent record |

# Techniques for inserting into Sequential Files stored on Direct Access Devices

## Deferred insertion

- Ala Master File Update

## Distributed Free Space

- Anticipate growth of file - leave space in each block initially for new records…. Limited growth and periodic reorg required

## Overflow

- When space in a block runs out put records in an overflow area. Link them to the block they follow.

## Cellular Splitting

- When a block becomes full split it into two blocks. This creates more distributed free space.
- Cellular splitting is better than the other methods (but more complex) because: more flexible and dynamic;  keeps records at smallest cell level in physical sequence (unlike chained overflow); clustered addition won't cause imbalance in search time (cf. overflow method)

# Usefulness of Sequential Files

## Advantages

- Sequential files are appropriate if pattern of access matches ordering of records on the file

## Disadvantages

- Rarely used to implement database systems unless an additional access path called a primary index is used, resulting in an indexed-sequential file
- We will cover indexes later in the course

# Direct File

We would like to be able to jump directly to the exact record we want, given its key

A direct file is one where any record can be accessed without reference to any other record

- Analogous to an array (whereas a serial file is analogous to a linked list).
- Predictable relationship between key and location of records: R(key) = address

# Addressing Technique#1: Direct Mapping

Design the file structure so that the key is also the address where the record is stored.

For example, if the key is a student number, we could place student 89252675 in record number 89252675 of the file

But there is a problem.....

- Few universities have 89,252,675 students
- If there are only 500,000 student records, and the file has 90,000,000 spaces, the file will mostly consist of empty records
- This system will work if artificial keys can be created that are small consecutive numbers − 1, 2, 3, 4, etc.

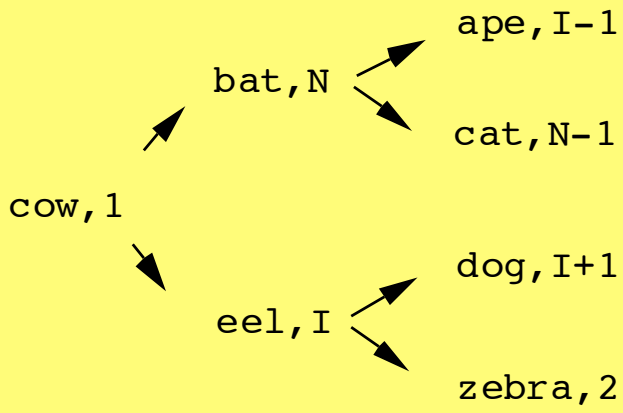
# Addressing Technique#2: Directory Lookup

directory

| key | address |
|---|---|
| ape | I-1 |
| bat | N |
| cat | n-1 |
| . . . | |
| cow | 1 |
| dog | I+1 |
| eel | I |
| . . . | |
| zebra | 2 |

directory as a table

| relative file | relative address |
|---|---|
| cow | 1 |
| zebra | 2 |
| . . . | |
| ape | I-1 |
| eel | I |
| dog | I+1 |
| . . . | |
| cat | N-1 |
| bat | N |




directory as a binary search tree

Difficulty in maintaining directory organisation, see indexes later

# Addressing Technique#3: Hashing

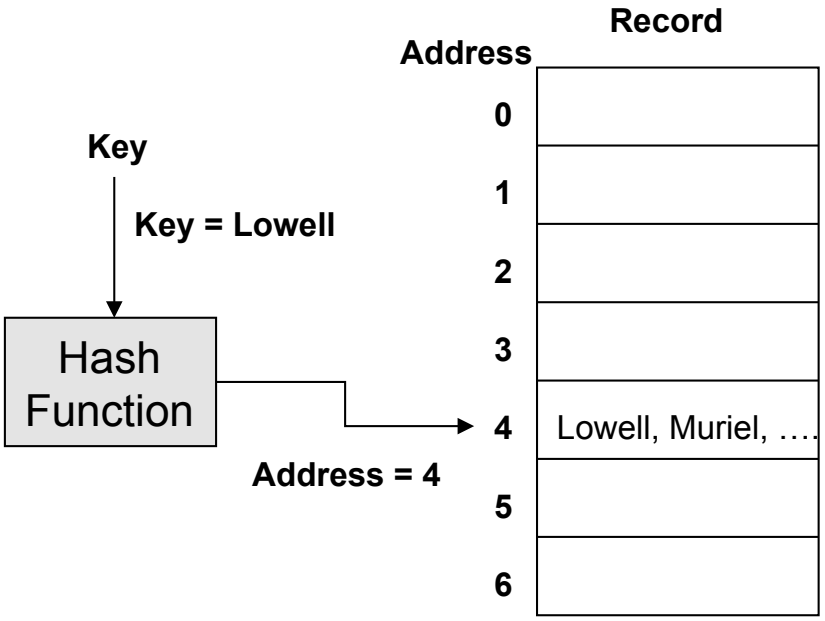
A hash function is like a black box into which one drops a key, and out comes an address

- Addresses appear to be "random" −− there is no obvious connection between the key and the corresponding record
- Two different keys may be transformed to the same address −− this is called a **collision**

# Hashing

Consider the following example:

- You want to store 75 records in a file, where the key to each record in the person's name. Assume there is space set aside for 1000 records. How can we provide hashed access to this file?



# Usefulness of Direct Files

## Advantages

Useful in interactive processing where :

- Only need to access target record
- Update can be done in place, insertion and deletion can be cheap
- No need to sort the file again when records added or removed
- Records are identified by one field (the key, assumed unique)

## Disadvantages

- Can’t have duplicate keys
- Does not easily handle variable length records
- Records are not accessible sequentially
- Only one key field, extra data structures needed for access by other keys

# File Organisation for Database Purposes

## Ordered files with fixed sized records not generally used

- Binary search is slow
- Files can be searched only by a single key value
- Insertion is horribly slow
- For example, take a movie record insertion whose movieid is less than all others…. Requires moving all of these records

## Unordered files with variable sized records more suited

- Use delimited-text representation or
- Use length-based binary representation

# Updating and Deleting

Updating involves, reading, changing then writing back

What if record being written back changes size?

- Move the record following the one being updated from its current position by deleting and appending to end of file

How do you delete a record?

- We represent length as a signed **positive** short integer
- Change sign of deleted record, but length still gives number of bytes to the next record
- Amend the direct read method so that it fails when trying to read a deleted record

# Review

Unordered Records: Heap/Pile/Serial Files

Ordered Records: Sequential File

Sequential File Processing

Direct Access

- Direct Mapping
- Directory Lookup (later in the course)
- Hashing (later in the course)

University of Dublin
Trinity College

# Finding Records

Owen.Conlan@cs.tcd.ie

# Finding Things in a File: Sequential Search

Sequential search is one of the simplest forms of file searching

The file is searched one record at a time, until a record is found with a particular key

Sequential search is slow:

- If there are n records in the file, you may have to look at all of them before you find the one you want
- If the key you are looking for is in the file, on average you will need to look through n/2 records before finding it

Sequential search is said to be O(n), because the time it takes is proportional to n

# Finding Things in a File: Sequential Search

Although sequential search is slow, it is not appalling

Sequential search always looks at the adjacent record in the file next

- Therefore, it makes good use of the fact that every read of a file does not result in a disk access
- A big chunk of the file is read into a buffer in main memory
- So most reads of the file will not actually result in disk accesses

# Motivation for Binary Search

Let's take an example:

- Suppose we're looking for a student with id number 76634 in a file of 10,000 fixed length records
- Assume further than the file has been sorted into ascending order of student numbers
- We start by comparing 76634 with the student number of the record in the middle of the file, that is record 5,000
- If record 5000's student number if greater than 76634, we know that 76634 will be found in the first half of the file

# Binary Search

- If it is less than 76634, then we know 76634 can be found in the second half of the file
- Assuming it we know that 76634 is in the first half, we now compare this with the student number of the record at position 2,500 to find out which quarter of the file 76634 is in
- The process is repeated until either 76634 is found or we have narrowed the number of potential records to zero
- This is called binary search

# Pseudocode for Binary Search

```
low = 0
high = number of records −1
while ( low <= high )
        guess = (low + high) / 2
        key_found = read_key_number(guess)
        if ( key_sought > key_found )
                low = guess + 1
        else if (key_sought < key_found ) high = guess − 1
                else
                        we've found it
endwhile
```

# Binary Search

The difference becomes dramatic if there are a lot of records in the file

- When we double the number of records, we double the number of comparisons for sequential search
- When we double the number of records, we add one to the number of comparisons for binary search
- BUT, even though it might take sequential search 5,000 comparisons, and binary search only 14 comparisons, does not mean that binary search is 5,000 / 14 = 357 times faster than sequential search
- Why?

# Binary Search Limitations

If we have a sorted file we can find a record quickly with binary search

But binary search is still not ideal:

Problem 1: binary search requires several disk accesses:

- Although binary search is a tremendous improvement over sequential search, those disk accesses are still expensive
- Ideally we would be able to find the data in just one or two accesses
- Ideally, we would be able to work out at which record number the data is stored from the key. We'll look at this in the coming lectures.

# Binary Search Limitations

Problem 2: Keeping a file sorted can be very expensive

- When we add records to the file, we need to resort the file
- This can be very, very expensive (see coming lectures)
- If we add records as often as we search for records, we will spend most of our time sorting the file
- Even if we can find the position to put the new record into cheaply, we need to move records to make space for the new record

Better solutions will have at least one of the following features:

- They will not involve re−ordering the file when a new record is added
- They will use data structures that allow rapid, efficient re−ordering of the file

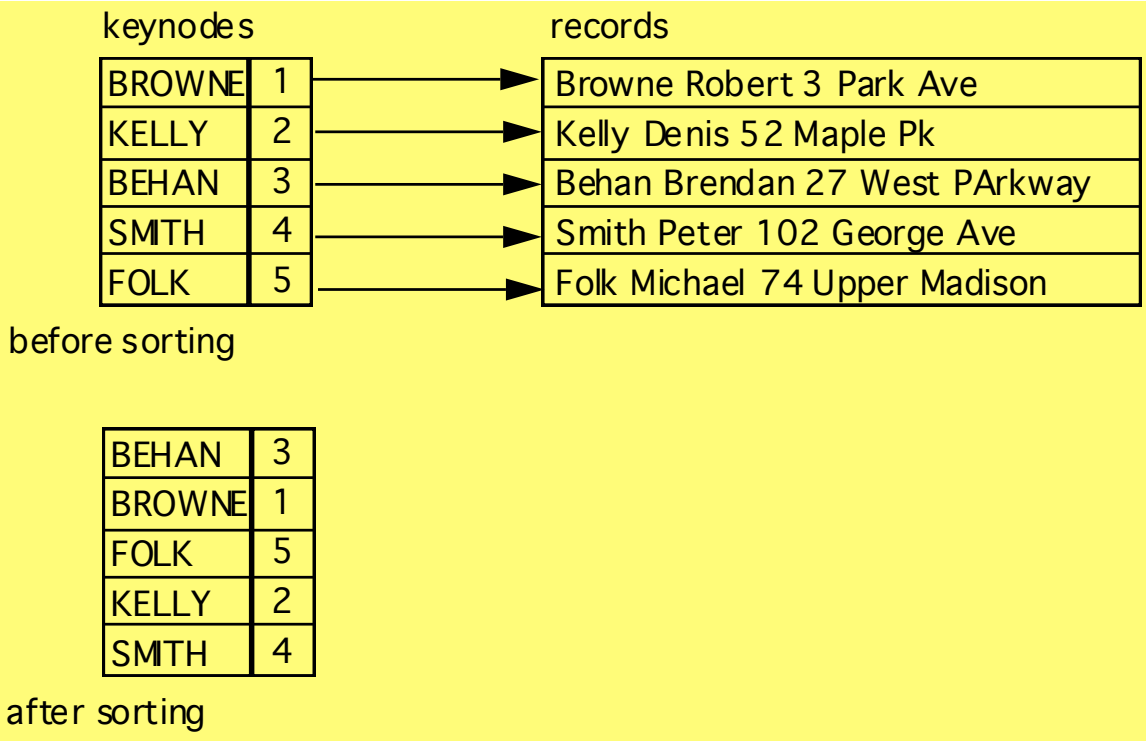
University of Dublin
Trinity College

# File Sorting

# RAMSORT#1 approach

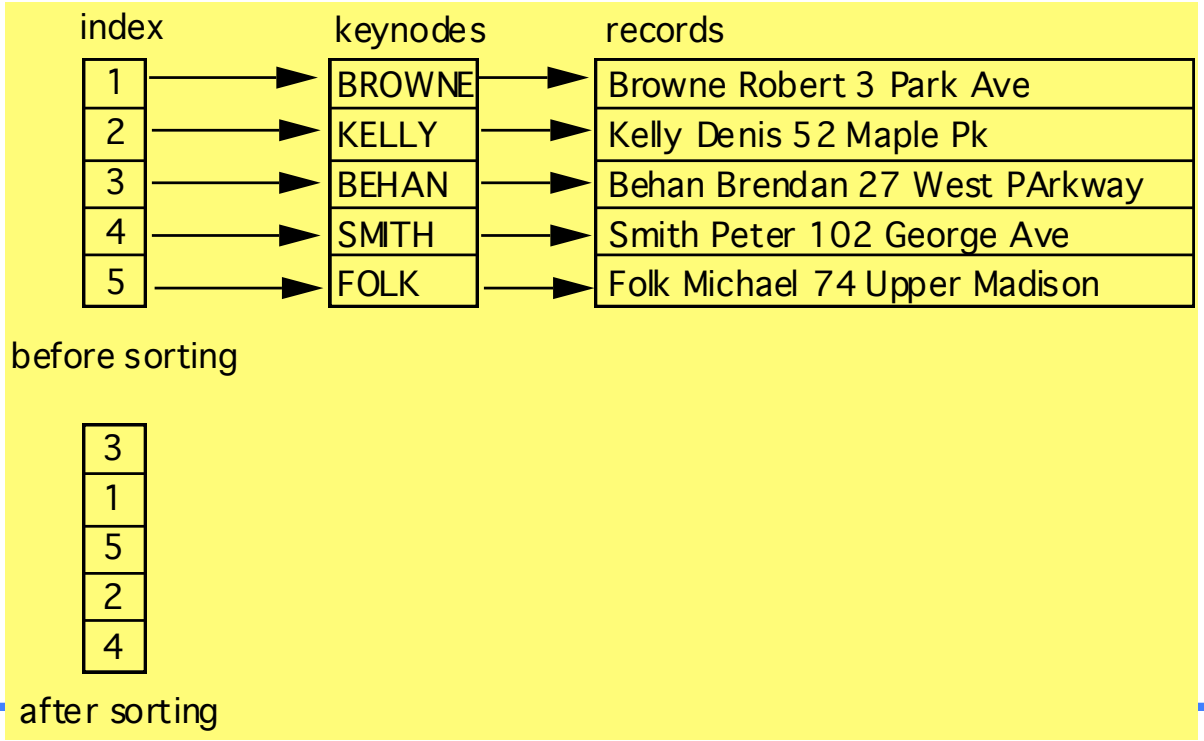
If the entire file fits in RAM

- Read in all the records sequentially
- Extract the key values (in canonical form)
- Keep a pointer (or record number) to the record with each key
- Sort the keys
- Write out the records in the order of the sorted keys



# RAMSORT#2 Approach

Separate the index from the keys
Sort the index based on the key values
Only the keys are needed in RAM while doing the sort
Records are only accessed to get keys and to write out sorted file



# KEYSORT Approach

If the entire file does **not** fit in RAM and stored on direct access device

- Only read the keys from disk
- Sort as for RAMSORT#2
- Write out the sorted file by
  - *reading the record corresponding to the next index value from disk*
  - *Writing it out to disk in a new position*

Limitations - Can be more expensive than first appears

- Applies only to files on disk
- Reads each record twice
- Second read involves reading records in sorted order which may require a random seek on disk
- Writes are sequential but interleaved with random seeks
- Size of file that can be sorted is limited by the number of key/ pointer pairs that can be contained in RAM

# KEYSORT

Sometimes you don't actually need to have the records in order - e.g. for binary search you can use the sorted index and keys to find the required key, and then follow the pointer to the record. In this case you wouldn't actually do the last stage of the KEYSORT.

But for sequential processing (MFU, merge) this would not be a good idea.

# EXTERNAL SORTING

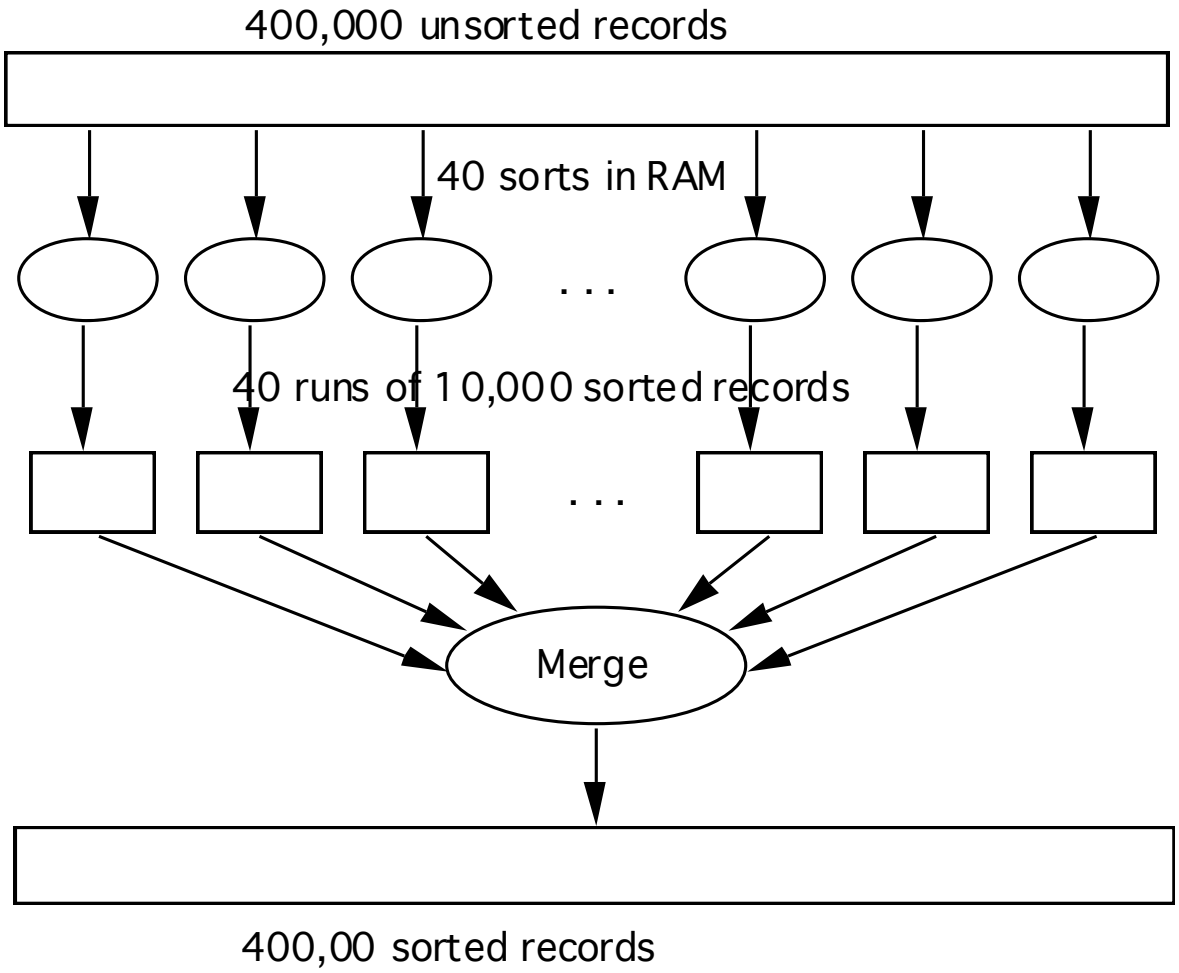
Suitable for sorting large files stored on disk that do not fit in RAM, such as most database files

Consists of Sorting and Merging Phases –

During the Sorting Phase **runs** (portions or pieces) of the file that can fit into the buffer space are read into main memory, sorted using internal sorting and written back to disk as temporary subfiles/runs

During the Merging Phases the sorted runs are merged during one or more **passes**

# EXTERNAL SORTING
# Example 40 Way Merge



# Example 2 Way Merge

| | |
|---|---|
| 14765 | Dowling Anne |
| 23401 | Kelly John |
| 27421 | Flood George |
| 34531 | Howe Mary |
| 43193 | Murphy David |

| | |
|---|---|
| 12453 | Green Michael |
| 31007 | Flynn Rita |
| 41002 | Browne Robert |

| | |
|---|---|
| 12453 | Green Michael |
| 14765 | Dowling Anne |
| 23401 | Kelly John |
| 27421 | Flood George |
| 31007 | Flynn Rita |
| 34531 | Howe Mary |
| 41002 | Browne Robert |
| 43193 | Murphy David |

# M-Way Merge

Store one record of each file in a buffer array

Write out smallest buffer element and read in a new record from the corresponding file

Invalid_Key is used to indicate which files we have finished. Normally larger than any possible value of the ordering field

# Pseudo code

```
open N input files and one output file
(* initialize buffers *)
loop from i = 1 to N
        if end_of_file (file i)
           then buffer[i] <- invalid_key else buffer[i] <- first record of
file i;
(* merge *)
stop <- false
repeat
        s <- index of smallest buffer element
        if buffer[s] = invalid_key
                then stop <- true else write buffer[s]
                        if end_of_file (file s)
                           then buffer[s] <- invalid_key
                           else buffer[s] <- next record from file s
until stop = true
close files
```

# Measuring Performance

The size of a run and **number of initial runs (nr)** is dictated by the number of file blocks (b) and the available buffer space (nb)

- Nr = b/nb
- E.g b = 1024 blocks and nb = 5 blocks then 205 initial runs will be needed

The **degree of merging (dm)** is the number of runs that can be merged together in each pass.

- One buffer block is needed to hold one block from each run being merged
- One buffer block is needed for containing one block of merged result
- Dm is the smaller of (nb – 1) and nr
- Number of passes = $\log_{dm}(nr)$

# Performance Issues

M-Way merges can require a significant amount of copying of data back and forth, causing significant I/O activity

The greater the number of initial runs and the lower the degree (the “M”) of the merge, the greater the I/O requirements

Other Merge techniques have been introduced to address this problem

- Balanced Merges
- Polyphase Merges
- Cascade Merges

# Review

## Finding Records

- Sequential Search (Performance)
- Binary Search (Pseudocode, Performance, Limitations)

## File Sorting

- RAMSORT
- KEYSORT
- External Sorting
- M-Way Merge (Pseudocode, Performance)