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

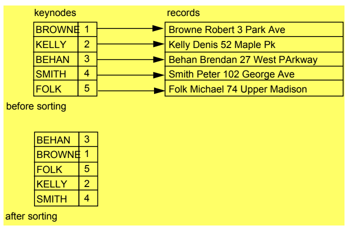
University of Dublin
Trinity College

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



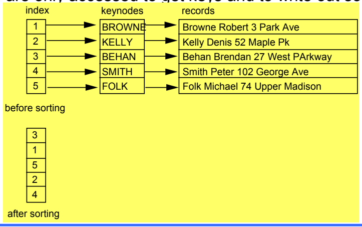
RAMSORT#2 Approach

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

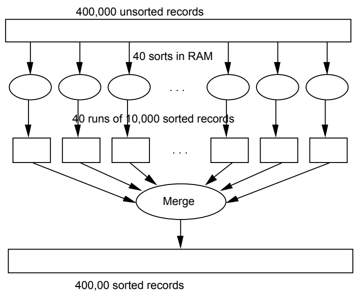
Suitable for sorting large files stored on disk that do not fit in RAM, such as most database files

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

4765	Dowling Anne
3401	Kelly John
7421	Flood George
4531	Howe Mary
3193	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)