SORTED SEQUENTIAL FILES

A **sorted file** is one in which records are stored:

- in order of the values of one field (e.g., ID number)
- or in order of the concatenation of several fields. (e.g., first & last names)

The **sort field** is sometimes called as a **key** of the file.

Our Assumptions:

- The sort field is a single field.
- Access to sorted file is mostly sequential.
- File contains only fixed-length records.

Handling of Additions to a Sorted File

- All records are shifted forward to keep the order in place.
  - very expensive
- An unsorted **overflow area** is reserved to keep all the records which were added after the file was first loaded.
  - To find a record first the main area then the overflow area is searched.
  - Periodic reorganizations will be necessary.

main area (sorted)	overflow area (unsorted)

Fetching One Record

Given the value of the field used for sorting we can do:

- a binary search in sorted area
- followed by a sequential search in overflow area.

Binary Search or Logarithmic Search Bisection Method

A binary search compares the key of the sought record with the middle record of the file:

- Then either the sought record has been located or
- half of the file is eliminated from further consideration.
  (Finally we are left with one block)

Note that block-based binary search of (Salzberg, 1988) is different from single-value binary search of (Tharp, 1988.) We follow Salzberg unless noted otherwise.

Total # of blocks in the file $\boxed{b = x + y}$

where

y: Numbers of blocks in sorted area.

x: Numbers of blocks in overflow area.

Average # random accesses if the record is in sorted area:

$= (1/y)*1 + (2/y)*2 + (4/y)*3 + \ldots + (1/2)*(\log y) \cong (\log y)-1$

Average # random accesses for an unsuccessful search: (worst case)

$\cong (\log y)$

Time for binary search for one record **in sorted are**:

$\cong [(\log y) - 1] * (s + r + btt)$

We don't consider the following fact that the first seek is near average seek time but later on seek time gets smaller.

Time for sequential search for one record in unsorted overflow area:

$= r + s + (x/2)*(ebt)$

Total time for searching a record which is placed in overflow area:

$$= \underbrace{(\log y)*(s + r + btt)}_{\substack{\text{Time for searching sorted area} \\ ①}} + \underbrace{r + s + (x/2)*(ebt)}_{\substack{\text{Time for sorting overflow area} \\ ②}}$$

- If x is very large, 2 is the most significant.
- If x is very small, 1 is the most significant.

y/b : The probability that the record is in sorted area.

x/b : The probability that the record is in overflow area.

Time to fetch a present record, given the value on the sort field :

$T_F \cong (y/b) * [(\log y)-1] * (s + r + btt)$
$\quad + (x/b) * [(\log y) * (s + r + btt) + r + s + (x/2)*ebt]$

When x is large:

$\boxed{T_F \text{ (significant overflow)} = (x/b) * (x/2) * ebt}$

When y is large:

$\boxed{T_F \text{ (significant sorted area)} = [(\log b)-1] * (s + r + btt)}$

Example: The Hospital File

A hospital file of 40 MB in 16677 blocks (b).

T_F = ...

Binary search is faster than sequential search

	Seq. Search	Binary search
T_F (Pile File)	7 sec	326 msec
Time for looking up 10,000 recs by name	19 hours	54 min
Number of disk accesses	8333 probes	13 probes

After we add 16667 more records, we have a 50 MB file
b = 33334 and x=y=16667

T_F = (1/2)*(326) + (1/2)*(14*(16 + 8.3 + 0.8) +24.3 +8333*0.84) = 3850 msec.

If we had used T_F =(x/b)*(x/2)*ebt then T_F =3500 msec

The performance for a fetch degrades as the proportion of the file in the overflow area becomes larger

Fetching the Next Record

Assume that
- small overflow area
- disk arm is already in correct track
- contents of last block read is still in memory
- next block is in the same cylinder.

Probability of next record being in the same block (i.e. still in memory) as the most recently read block = 1 - (1/Bfr)

Probability that it is not in the same block = 1/Bfr

Estimated time for reading next record :

T_N = [1-(1/Bfr)] * 0 + (1/Bfr)*(r+btt) * 1 = (1/Bfr)*(r + btt)

(r is needed since a separate command is given for the next record.)

Deleting and Modifying Records

The record is marked deleted in case of a deletion, so deletion operation is the same as modification.

- If the field is not the same as the field on which the records are sorted:

$$\boxed{T_D = T_U = T_F + 2r}$$

2r is the time it takes after reading the correct block to revolve back to the beginning of the correct block and write the new block in.

- If the modification involves changing the value of the sort field, the update is a deletion plus an insertion:

$$\boxed{T_U = T_D + T_I}$$

Inserting a Record

We assume that duplicate values are allowed:
- makes overflow area larger
- increases file space
- increases search time

+ decreases insertion time. *How ?*

- The new record is merely placed at the end of the overflow area:

$$\boxed{T_I = s + r + btt + 2r}$$

The time for insertion is the same as for an insertion in the pile file.

Exhaustive Reading of the File

Assuming nothing has been added to the overflow area or the order is not important:

$$\boxed{T_x = b * ebt}$$

Keeping the file sorted do not have any importance if the whole file is to be read.

The time to read in the whole file in order (if there are records in the overflow area) is the time:
- to read in all overflow area into memory (assuming it fits,)
- sort the overflow data,
- merge the overflow with the sorted area as the sorted area blocks are read in.

Creating a File of Female Patients

Generate a female patients file from the hospital file.
- **Assume the records are sorted by sex and the size of the overflow area is insignificant.**

Algorithm:
- Make a search for the first female patient (if males were listed first.)
- Then read through the rest of the file:

Analysis of Algorithm:
- Binary search = 326 msec
- Reading half of the file sequentially = 7 sec

(With a pile file we would need 14 seconds.)

Reorganization

$$T_x = \underbrace{x*ebt}_{\text{reading in overflow area}} + \underbrace{(z \log z)*0.01}_{\text{sorting it}} + \underbrace{y*ebt + (n/Bfr)*ebt}_{\text{writing it out}}$$

Assume overflow fits in memory, z is the number of active records in overflow, and sort takes 10 msecs for each iteration of the sort routine.

See Figure 3.8

Merging sorted segments of the file:

- read into memory a piece of each segment (one segment, the sorted overflow is already in memory)
- look at first record from each sorted segments, choose the record with the smaller key and move it to output buffer.
- continue the same operation until two segments are merged.

Merging Two Sorted Segments with One Disk Drive

- Read into memory large pieces of both sorted segments, A and B. See Figure 3.9
- While unmerged remain in both A and B, repeat the following:
  - Compare the smallest key from unmerged records A in memory with the smallest key from unmerged records of B in memory.
  - Move the record with smaller key to the output buffer. Write the output buffers to disk as they fill, using double buffering.
  - If the piece of A in memory of piece of B in memory no longer contains unmerged records, stop writing to disk and read in the next piece of A or B.
- One of A or B is exhausted. Write the remaining records of the nonexhausted segment to the merged file.

Merging Two Sorted Segments with Two Disk Drives

We can read, merge and write at the same time.

- See Figure 3.10
- read in and sort the overflow first (**Assuming it fits into memory**)
- read in the sorted part of the file using double buffering
- merge with the overflow
- write out the result at once

Incoming records

- are kept in sorted order in a linked list in memory (saves room)
- take the place of outgoing records from both segments

Extended Example: The Intersection File

We have two hospital files (MA & BCBS) of 100,000 records of 400 bytes each. 70,000 of these records are in both files. We wish to make an intersection file.

Assume both files are sorted acc. to patient's SSN.

<u>Algorithm:</u>

1- Read in 1/4 of (10 MB) MA file

2- For each MA record check if it matches a record in BCBS file

- if there are some BCBS records that come before first MA record, they are not in intersection.
- if in going through the BCBS file we find a number bigger than the one we are trying to match, before we find a match, there is a match.

We need only read through BCBS file once for the entire operation (using double buffering.)

3- As soon as we find a BCBS record whose SSN field is larger than the largest value in memory we write out the matched records so far and read in the next MA file segment.

Natural Join

- If two files are both sorted on the same criteria, we need only read through each file once to find matches.

<u>Analysis:</u>

Reading MA file	14 seconds
Reading BCBS file	14 seconds
Writing intersection file	10 seconds (70% of records match)

TOTAL of 38 seconds.

(Compare this with 10.5 days when unsorted pile files are used for this operation.)

Interpolation Search (IS)

IS chooses next position for a comparison based upon the estimated position of the sought key relative to the remainder of the file to be searched.

Ex: Humans search names in a telephone list using IS.

$$\text{Next Position} = \text{ceiling}\left(\text{lower} + \frac{\text{key[sought]} - \text{key[lower]}}{\text{key[upper]} - \text{key[lower]}} (\text{upper} - \text{lower})\right)$$

Worst case complexity : O(n)

Average case complexity : $O(\log_2 \log_2 n)$

- Its performance improves as the distribution of the keys becomes more uniform.
- A binary search is preferable to IS when data is stored in primary memory (because IS involves additional calculations.)
- Both binary search and IS is of limited use since as the file grows average number of probes gets higher (for sequential file organizations)