Chapter 6 : Organizing Files For Performance

- Goal

1. Look at several approaches to data compression
2. Storage compaction as a simple way of reusing space in file
3. Illustrate the use of linked list and stacks to manage an avail list
4. Internal and external fragmentation
5. Introduction to binary search
6. Key sorting

- Introduction

- look at ways to organize / reorganize files in order to improve performance
- Compression techniques make files smaller by encoding the basic information in the file
- reorganize files by binary searching

*6.1 Data Compression

- use less storage, resulting in cost savings
- can be transmitted faster, decreasing access time
- can be processed faster sequentially

↳ data compression involves encoding the information in a file in such a way that it takes up less space. Many techniques are available for compressing data

• 6.1.1 Using Different Notation

↳ fixed-lenght fields are good candidates for compression

EX:

State field in Person required 16 bits.
There are 50 states, we could represent all of them with 6 bits

↳ compact notation: type of compression technique in which we decrease the number of bits

- kind of redundancy reduction
(10 bits were redundant in this sense)

↳ What are the costs of this compression scheme?

- by using a binary encoding (for example) we make the file unreadable by humans
- we incur some cost in encoding time when saving info or retrieving it from file
- increase the complexity of the software

↳ Good for:

- files with several million records and is generally processed by one program
- because the encoding/decoding algorithms for this kind of compression are extremely simple, the saving time in access time is likely to exceed any processing time required for encoding/decoding

6.1.2 Suppressing Repeating Sequences

↳ Sparse arrays (images with a lot of 0s)

↳ run-length encoding

EX: 8-bit image

- read all the pixels and write in a file, except where the same pixel value occurs more than once in succession

- where the same value occurs more than once in succession, substitute the following 3 bytes in order:
  - * the special run-lenght code indicator
  - * the pixel value that is repeated
  - * the number of times the value is repeated (up to 256)

- EX:

run-lenght code indicator ⇒ byte 0xff

Hexodecimal bytes:

[22 23 24 24 24 24 24 24 24 25 26 26 26 26 26 26
25 24]

Encoding:

[22 23 ff 24 07 25 ff 26 06 25 24]

ff → Special run-length code indicator
24 → the value that is repeated
07 → number of times the value is repeated

↳ run-lenght encoding is another example of redundancy reduction. It can be applied to:

- text, instrument data, space matrices

* 6.2 Reclaiming Space in Files

↳ Situation: a record in a variable-length record file is modified in such a way that the new record is no longer than the original record.

↳ What do you do with the extra data?

(1) You could append it to the end of the file and put a pointer from the original record space to the extension of the record

(2) Rewrite the whole record at the end of the file (unless the data needs to be sorted), leaving a hole at the original location of the record

↳ Each solution has a drawback

①: the job of processing the record is more awkward and slower

②: the file contains wasted space

↳ We will see at the way file organization deteriorates as a file is modified. Modifications can be:

- record addition
- record deletion
- record updating,

Obs:

- if the only kind of change to a file is record addition, there is no deterioration
- it deteriorates when variable-length records are updated
- or when either fixed- or variable-length records are deleted
- since record updating can be treated as a record deletion followed by a record addition, we will focus on the effects of deletion
- when a record is deleted, we want to reuse the space

6.2.1 Record Deletion and Storage Compaction

↳ Storage Compaction makes files smaller by looking for places in a file where there is no data at all and recovering this space

↳ Any record-deletion strategy must provide some way for us to recognize records as deleted

↳ Example: We may place an asterisk as the first field in a deleted record

[before]

Ames	Mary	123 Maple	Stillwater	Ok	74075	.....
Morrison	Sebastian	9035 South Hill	Forest Village	Ok	74820	
Brown	Martha	625 Kimbark	Des Moines	IA	50311	...

[after]

Ames	Mary	123 Maple	Stillwater	Ok	74075	.....
*	rrison	Sebastian	9035 South Hill	Forest Village	Ok	74820
Brown	Martha	625 Kimbark	Des Moines	IA	50311	....

↳ the next question is: how to reuse the space from the record?

- Approaches based on storage compaction do not reuse the space for a while
- the records are marked as deleted and left in the file for a period of time
- programs must include logic to ignore records that are marked as deleted
- One benefit is that is possible to undo a deletion with very little effort

↳ the reclamation of space from all the deleted records happens all at once

- after deleted records have accumulated for some time, a special program is used to reconstruct the file with all the deleted records squeezed out as:

[Ames | Mary | 123 Maple | Stillwater | Ok | 74075 | .....
Brown | Martha | 625 Kimbark | Des Moines | IA | 50311 | ....]

- a simple solution is through a file copy program that skips over the deleted records
- it is also possible to do the compaction in place

↳ When to run the storage compaction program can be based on either the number of deleted records or the calendar.

↳ Storage Compaction is the simplest and most widely used of the storage reclamation methods we discuss.

6.2.2 Deleting Fixed-Length Records For Reclaiming Space Dinamically

↳ there are some applications that are too volatile and interactive for storage compaction to be useful

- We want to reuse the space from deleted space as soon as possible
- dynamic storage reclamation

↳ So, to provide dynamic storage reclamation, we need two things:

- that deleted records are marked in some special way
- that we can find the space the deleted records once occupied so we can reuse the space when we add records

↳ the first requirement is easily solved:

- mark record as deleted by putting a field containing an asterisk at the beginning of deleted records

↳ Space reutilization can take the form of:

- looking throught the file
- check record by record, until a deleted record is found
- if the file reaches the end of the file without finding a deleted record, the new record can be appended at the end

*but it makes the process slow if the program is interactive. to make it more quickly, we need:

- a way to know immediately if there are empty slots in the file, and
- a way to jump directly to one of these slots, if they exist

Linked-List

↳ the use of a linked list can achieve both criteria

- you can move through the list by looking at each node, and knowing the node's pointer field

- when a list is made up of deleted records that have become available space within the file, this list is usually called an avail list

- When inserting a new record into a fixed-length record file, any one available record is just as good as any other. there is no reason to prefer one open slot over another since all the slots are the same size.

Stacks

↳ it is the simplest way to handle a list

↳ stack that contains relative record numbers (RRN)
  - the most recent space is reused

Linking and Stacking Deleted Records

↳ So, placing deleted records in a stack meets both of the previous criteria

  - empty → no deleted records, so append to the end of the file

  - not empty → reuse space, and using RRN we know where to find

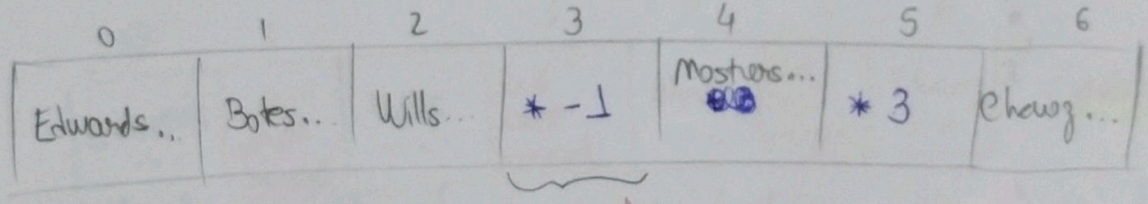
↳ the stacking and linking are done by arranging and rearranging the links used to make one available record slot point to the next

- since we are working with fixed-length records, "the pointer" is done through relative record number (RRN)

Example:

(a) List head → 5

How file might look



end of list marker (end of list) last space

- 5 is the first record in the avail list (most recent deleted)
- record 5 points to record 3
- record 3 contains -1, so it is the last position of the list / stack

List Head → 1

(b)

0	1	2	3	4	5	6
Edwards ...	*5	Wills ...	*-1	Masters ...	*3	Chavez ...

- same file after record 1 is deleted
- treating the list as a stack results in a minimal amount of list reorganizations

List Head → -1

(c)

0	1	2	3	4	5	6
Edwards ...	1st new record	Wills ...	3rd new record	Masters ...	2nd new record	Chavez ...

- after the insertion of three records
- if yet another name is added to the file, the program knows that the avail list is empty and that the name requires the addition of a new record at the end of the file

* We need a suitable place to keep the RRN of the first available record on the avail list
  - it can be carried in a header record at the start of the file

- When deleting a record, place * and place it on the avail list

  ↳ * + <RRN> at the beginning of the record

- Simple function that returns:
  - the RRN of a reusable record slot, or
  - the RRN of the next record to be appended if no reusable slots are available

6.2.3 Deleting Variable-Length Records

↳ When handling variable-length records, we need:
- a way to link deleted records together into a list
- an algorithm for adding newly deleted records to the avail list
- an algorithm for finding and removing records from the avail list when we are ready to use them

An Avail List of Variable-Length Records

↳ We need a structure in which the record is a clearly defined entity

4 We can handle the contents of a deleted variable-length record

- place an asterisk in the first field, followed by a binary linked field pointing to the next deleted record on the avail list
- but, we cannot use relative record number
- the links must contain the byte offsets themselves

EX:

(a) Head.First_Avail: -1

40 Ames|Mary|123 Maple|Stillwater|OK|74075|64 Morrison|
Sebastian|9035 South Hillcrest|Forest Village|OK|74820|45
Brown|Martha|625 Kimbark|Des Moines|IA|50311|

- original file

Head.First_Avail: 43

(b) 40 Ames|Mary|123 Maple|Stillwater|OK|74075|64 *|-1...
.....|45
Brown|Martha|625 Kimbark|Des Moines|IA|50311|

.... } means discarded characters

Adding and Removing Records

↳ adding and removing records to and from the list together

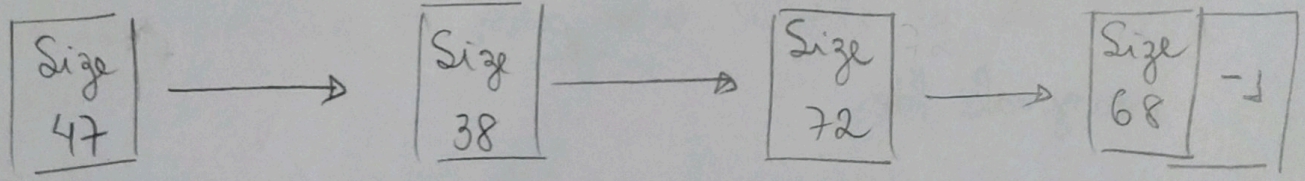
- we cannot use stacks because records have different lengths
- so, we have an extra condition that must be met before we can use a record:

"the record must be the right size"

- right size = big enough
- we need to traverse the list until a record slot that is big enough to hold the new record is found

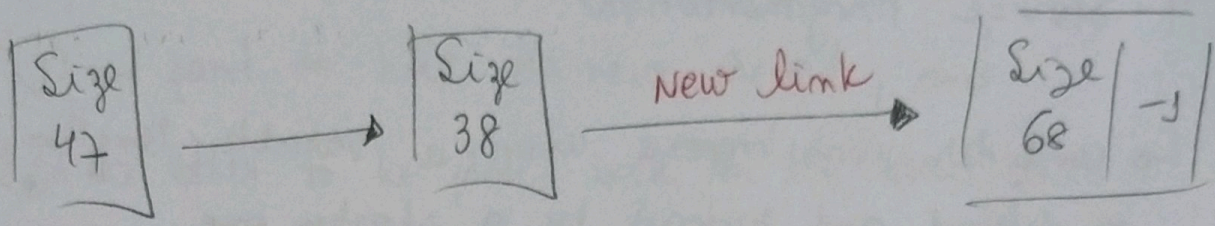
EX:

a)



- avail list containing the deleted records slots

b)



- new record with 55 bytes is to be added
- need to traverse the list and find a slot
  - 47 (too small)
  - 38 (too small)
  - 72 (big enough)!
- remove it from the avail list, creating a new link
- if we reached the end of the list before finding a record that was large enough
  - append the record to the end of the file
- can continue to push new records to the list

- Developing algorithms for adding and removing avail list records is left to you as part of the exercises

6.2.4 Storage Fragmentation

↳ consider what happens when a variable-length record is deleted and replaced by a shorter one

Example:

New Record = Ham|A1|28 Elm|Ada|Ok|70332|

a) Head.First_Avail: 43

40 Ames|Mary|123 Maple|Stillwater|Ok|74075|64 *1|-1
.....|45 Brown|
Martha|625 Kimbark|Des Moines|IA|50311|

b) Head.First_Avail: -1

40 Ames|Mary|123 Maple|Stillwater|Ok|74075|64 Ham|A1|
28 Elm|Ada|Ok|70332|.....|45 Brown|
Martha|625 Kimbark|Des Moines|IA|50311|

- the slot vacated is 37 bytes larger than is needed for the new record
- the 37 extra bytes won't be on the avail list and therefore unusable
- We can:
  - → break the space into two parts

① one part to hold the new record, and

② the other to be placed back on the avail list

- thus, this way there would be no internal fragmentation

EX:

Head.First_Avail: 43

40 Ames | Mary | 123 Maple | Stillwater | OK | 74075 | 35 * | -1

..... | 26 Ham | Al | 28 Elm | Ada | OK | 70332 | 45

Brown | Martha | 625 Kimbark | Des Moines | IA | 50311 |

- We use the space from the end of the record

*External Fragmentation

→ the space is on the avail list rather than being locked inside some other record

→ it is too fragmented to be reused

→ Some ways to handle:

- storage compaction: regenerate the file
- if two slots on the avail list are physically adjacent, combine them. Coalescing the holes

- Improve the program that selects records from the avail list

6.2.5 Placement Strategies

* First-fit : from the beginning, until we find a record slot that is big enough or reach the end of the file

* We can keep a more orderly approach for placing records on the avail list by keeping them in either ascending or descending sequence by size

  - ascending : the first record encountered is the smallest record that will do the job

    → best-fit placement

    → longer as time goes on

  - descending : worst-fit

    → if the first option is not enough, none will be

    → extracting space from the largest available records reduces the external fragmentation

* No one placement strategy is superior under all circumstances

* Some observations:

→ placement strategies make sense only with regard to volatile, variable-length record files. With fixed-length records, placement is not an issue

→ if space is lost due internal fragmentation, the choice is between first-fit and best-fit

→ if space is lost due external fragmentation, should consider the worst-fit