CHAPTER 6

Organizing Files for Performance

CHAPTER OBJECTIVES

- Look at several approaches to *data compression.*
- Look at *storage compaction* as a simple way of reusing space in a file.
- Develop a procedure for deleting fixed-length records that allows vacated file space to be reused dynamically.
- Illustrate the use of *linked lists* and *stacks* to manage an *avail list.*
- Consider several approaches to the problem of deleting variable-length records.
- Introduce the concepts associated with the terms *internal fragmentation* and *external fragmentation.*
- Outline some *placement strategies* associated with the reuse of space in a variable-length record file.
- Provide an introduction to the idea of a *binary search.*
- Examine the limitations of binary searching.
- Develop a *keysort* procedure for sorting larger files; investigate the costs associated with keysort.
- Introduce the concept of a *pinned record.*

CHAPTER OUTLINE

We have already seen how important it is for the file system designer to consider how a file is to be accessed when deciding on how to create fields, records, and other file structures. In this chapter we continue to focus on file organization, but the motivation is different. We look at ways to organize or reorganize files in order to improve performance.

In the first section we look at how we organize files to make them smaller. Compression techniques let us make files smaller by encoding the basic information in the file.

Next we look at ways to reclaim unused space in files to improve performance. Compaction is a batch process that we can use to purge holes of unused space from a file that has undergone many deletions and updates. Then we investigate dynamic ways to maintain performance by reclaiming space made available by deletions and updates of records during the life of a file.

In the third section we examine the problem of reorganizing files by sorting them to support simple binary searching. Then, in an effort to find a better sorting method, we begin a conceptual line of thought that will continue throughout the rest of this text: we find a way to improve file performance by creating an external structure through which we can access the file.

6.1 Data Compression

In this section we look at some ways to make files smaller. There are many reasons for making files smaller. Smaller files

- Use less storage, resulting in cost savings;
- Can be transmitted faster, decreasing access time or, alternatively, allowing the same access time with a lower and cheaper bandwidth; and
- 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 different techniques are available for compressing data. Some are very general, and some are designed for specific kinds of data, such as speech, pictures, text, or instrument data. The variety of data compression techniques is so large that we can only touch on the topic here, with a few examples.

6.1.1 Using a Different Notation

Remember our `Person` file from Chapter 4? It had several fixed-length fields, including `LastName`, `State`, and `ZipCode`. Fixed-length fields such as these are good candidates for compression. For instance, the `State` field in the `Person` file required 2 ASCII bytes, 16 bits. How many bits are *really* needed for this field? Since there are only fifty states, we could represent all possible states with only 6 bits. Thus, we could encode all state names in a single 1-byte field, resulting in a space savings of 1 byte, or 50 percent, per occurrence of the state field.

This type of compression technique, in which we decrease the number of bits by finding a more *compact notation*,[1] is one of many compression

1. Note that the original two-letter notation we used for "state" is itself a more compact notation for the full state name.

techniques classified as *redundancy reduction*. The 10 bits that we were able to throw away were redundant in the sense that having 16 bits instead of 6 provided no extra information.

What are the costs of this compression scheme? In this case, there are many:

- By using a pure binary encoding, we have made the file unreadable by humans.
- We incur some cost in encoding time whenever we add a new state-name field to our file and a similar cost for decoding when we need to get a readable version of state name from the file.
- We must also now incorporate the encoding and/or decoding modules in all software that will process our address file, increasing the complexity of the software.

With so many costs, is this kind of compression worth it? We can answer this only in the context of a particular application. If the file is already fairly small, if the file is often accessed by many different pieces of software, and if some of the software that will access the file cannot deal with binary data (for example, an editor), then this form of compression is a bad idea. On the other hand, if the file contains several million records and is generally processed by one program, compression is probably a very good idea. Because the encoding and decoding algorithms for this kind of compression are extremely simple, the savings in access time is likely to exceed any processing time required for encoding or decoding.

6.1.2 Suppressing Repeating Sequences

Imagine an 8-bit image of the sky that has been processed so only objects above a certain brightness are identified and all other regions of the image are set to some background color represented by the pixel value 0. (See Fig. 6.1.)

Sparse arrays of this sort are very good candidates for a kind of compression called *run-length encoding*, which in this example works as follows. First, we choose one special, unused byte value to indicate that a run-length code follows. Then, the run-length encoding algorithm goes like this:

- Read through the pixels that make up the image, copying the pixel values to the file in sequence, except where the same pixel value occurs more than once in succession.

Figure 6.1 The empty space in this astronomical image is represented by repeated sequences of the same value and is thus a good candidate for compression. (This FITS image shows a radio continuum structure around the spiral galaxy NGC 891 as observed with the Westerbork Synthesis radio telescope in The Netherlands.)

- Where the same value occurs more than once in succession, substitute the following 3 bytes, in order:
  - The special run-length code indicator;
  - The pixel value that is repeated; and
  - The number of times that the value is repeated (up to 256 times).

For example, suppose we wish to compress an image using run-length encoding, and we find that we can omit the byte 0xff from the representation of the image. We choose the byte 0xff as our run-length code indicator. How would we encode the following sequence of hexadecimal byte values?

```
22 23 24 24 24 24 24 24 24 25 26 26 26 26 26 26 25 24
```

The first three pixels are to be copied in sequence. The runs of 24 and 26 are both run-length encoded. The remaining pixels are copied in sequence. The resulting sequence is

```
22 23 ff 24 07 25 ff 26 06 25 24
```

Run-length encoding is another example of redundancy reduction. It can be applied to many kinds of data, including text, instrument data, and sparse matrices. Like the compact notation approach, the run-length encoding algorithm is a simple one whose associated costs rarely affect performance appreciably.

Unlike compact notation, run-length encoding does not guarantee any particular amount of space savings. A "busy" image with a lot of variation will not benefit appreciably from run-length encoding. Indeed, under some circumstances, the aforementioned algorithm could result in a "compressed" image that is larger than the original image.

6.1.3 Assigning Variable-Length Codes

Suppose you have two different symbols to use in an encoding scheme: a dot (·) and a dash (-). You have to assign combinations of dots and dashes to letters of the alphabet. If you are very clever, you might determine the most frequently occurring letters of the alphabet (*e* and *t*) and use a single dot for one and a single dash for the other. Other letters of the alphabet will be assigned two or more symbols, with the more frequently occurring letters getting fewer symbols.

Sound familiar? You may recognize this scheme as the oldest and most common of the *variable-length codes*, the Morse code. Variable-length codes, in general, are based on the principle that some values occur more frequently than others, so the codes for those values should take the least amount of space. Variable-length codes are another form of redundancy reduction.

A variation on the compact notation technique, the Morse code can be implemented using a table lookup, where the table never changes. In contrast, since many sets of data values do not exhibit a predictable frequency distribution, more modern variable-length coding techniques dynamically build the tables that describe the encoding scheme. One of the most successful of these is the *Huffman code*, which determines the probabilities of each value occurring in the data set and then builds a binary tree in which the search path for each value represents the code for that value. More frequently occurring values are given shorter search paths in the tree. This tree is then turned into a table, much like a Morse code table, that can be used to encode and decode the data.

For example, suppose we have a data set containing only the seven letters shown in Fig. 6.2, and each letter occurs with the probability indicated. The third row in the figure shows the Huffman codes that would be assigned to the letters. Based on Fig. 6.2, the string "abde" would be encoded as "101000000001."

In the example, the letter *a* occurs much more often than any of the others, so it is assigned the 1-bit code 1. Notice that the minimum number of bits needed to represent these seven letters is 3, yet in this case as many as 4 bits are required. This is a necessary trade-off to ensure that the

Letter:	a	b	c	d	e	f	g
Probability:	0.4	0.1	0.1	0.1	0.1	0.1	0.1
Code	1	010	011	0000	0001	0010	0011

Figure 6.2 Example showing the Huffman encoding for a set of seven letters, assuming certain probabilities (from Lynch, 1985).

distinct codes can be stored together, without delimiters between them, and still be recognized.

6.1.4 Irreversible Compression Techniques

The techniques we have discussed so far preserve all information in the original data. In effect, they take advantage of the fact that the data, in its original form, contains redundant information that can be removed and then reinserted at a later time. Another type of compression, *irreversible compression,* is based on the assumption that some information can be sacrificed.[2]

An example of irreversible compression would be shrinking a raster image from, say, 400-by-400 pixels to 100-by-100 pixels. The new image contains 1 pixel for every 16 pixels in the original image, and there is no way, in general, to determine what the original pixels were from the one new pixel.

Irreversible compression is less common in data files than reversible compression, but there are times when the information that is lost is of little or no value. For example, speech compression is often done by *voice coding,* a technique that transmits a paramaterized description of speech, which can be synthesized at the receiving end with varying amounts of distortion.

6.1.5 Compression in Unix

Both Berkeley and System V Unix provide compression routines that are heavily used and quite effective. System V has routines called `pack` and `unpack`, which use Huffman codes on a byte-by-byte basis. Typically, `pack` achieves 25 to 40 percent reduction on text files, but appreciably less on binary files that have a more uniform distribution of byte values. When

2. Irreversible compression is sometimes called "entropy reduction" to emphasize that the average information (entropy) is reduced.

`pack` compresses a file, it automatically appends a .z to the end of the packed file, signaling to any future user that the file has been compressed using the standard compression algorithm.

Berkeley Unix has routines called `compress` and `uncompress`, which use an effective dynamic method called Lempel-Ziv (Welch, 1984). Except for using different compression schemes, `compress` and `uncompress` behave almost the same as `pack` and `unpack`.[3] Compress appends a .Z to the end of files it has compressed.

Because these routines are readily available on Unix systems and are very effective general-purpose routines, it is wise to use them whenever there are no compelling reasons to use other techniques.

6.2 Reclaiming Space in Files

Suppose a record in a variable-length record file is modified in such a way that the new record is longer than the original record. What do you do with the extra data? 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. Or you could rewrite the whole record at the end of the file (unless the file needs to be sorted), leaving a hole at the original location of the record. Each solution has a drawback: in the former case, the job of processing the record is more awkward and slower than it was originally; in the latter case, the file contains wasted space.

In this section we take a close look at the way file organization deteriorates as a file is modified. In general, modifications can take any one of three forms:

- Record addition,
- Record updating, and
- Record deletion.

If the only kind of change to a file is record addition, there is no deterioration of the kind we cover in this chapter. It is only when variable-length records are updated, or when either fixed- or variable-length records are deleted, that maintenance issues become complicated and interesting. Since record updating can always be treated as a record dele-

3. Many implementations of System V Unix also support `compress` and `uncompress` as Berkeley extensions.

tion followed by a record addition, our focus is on the effects of record deletion. When a record has been 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. Since empty spaces occur in files when we delete records, we begin our discussion of compaction with a look at record deletion.

Any record-deletion strategy must provide some way for us to recognize records as deleted. A simple and usually workable approach is to place a special mark in each deleted record. For example, in the file of `Person` objects with delimited fields developed in Chapter 4, we might place an asterisk as the first field in a deleted record. Figures 6.3(a) and 6.3(b) show a name and address file similar to the one in Chapter 4 before and after the second record is marked as deleted. (The dots at the ends of records 0 and 2 represent padding between the last field and the end of each record.)

Once we are able to recognize a record as deleted, the next question is how to reuse the space from the record. Approaches to this problem that rely on storage compaction do not reuse the space for a while. The records are simply marked as deleted and left in the file for a period of time. Programs using the file must include logic that causes them to ignore records that are marked as deleted. One benefit to this approach is that it is usually possible to allow the user to undelete a record with very little

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

(a)

```
Ames|Mary|123 Maple|Stillwater|OK|74075|.......................
*|rrison|Sebastian|9035 South Hillcrest|Forest Village|OK|74820|
Brown|Martha|625 Kimbark|Des Moines|IA|50311|..................
```

(b)

```
Ames|Mary|123 Maple|Stillwater|OK|74075|.......................
Brown|Martha|625 Kimbark|Des Moines|IA|50311|..................
```

(c)

Figure 6.3 Storage requirements of sample file using 64-byte fixed-length records. (a) Before deleting the second record. (b) After deleting the second record.(c) After compaction—the second record is gone.

effort. This is particularly easy if you keep the deleted mark in a special field rather than destroy some of the original data, as in our example.

The reclamation of space from 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 shown in Fig. 6.3(c). If there is enough space, the simplest way to do this compaction is through a file copy program that skips over the deleted records. It is also possible, though more complicated and time-consuming, to do the compaction in place. Either of these approaches can be used with both fixed- and variable-length records.

The decision about how often to run the storage compaction program can be based on either the number of deleted records or the calendar. In accounting programs, for example, it often makes sense to run a compaction procedure on certain files at the end of the fiscal year or at some other point associated with closing the books.

6.2.2 Deleting Fixed-Length Records for Reclaiming Space Dynamically

Storage compaction is the simplest and most widely used of the storage reclamation methods we discuss. There are some applications, however, that are too volatile and interactive for storage compaction to be useful. In these situations we want to reuse the space from deleted records as soon as possible. We begin our discussion of such dynamic storage reclamation with a second look at fixed-length record deletion, since fixed-length records make the reclamation problem much simpler.

In general, to provide a mechanism for record deletion with subsequent reutilization of the freed space, we need to be able to guarantee two things:

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

We have already identified a method of meeting the first requirement: we mark records as deleted by putting a field containing an asterisk at the beginning of deleted records.

If you are working with fixed-length records and are willing to search sequentially through a file before adding a record, you can always provide the second guarantee if you have provided the first. Space reutilization can take the form of looking through the file, record by record, until a deleted

record is found. If the program reaches the end of the file without finding a deleted record, the new record can be appended at the end.

Unfortunately, this approach makes adding records an intolerably slow process, if the program is an interactive one and the user has to sit at the terminal and wait as the record addition takes place. To make record reuse happen 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 those slots if they exist.

Linked Lists

The use of a *linked list* for stringing together all of the available records can meet both of these needs. A linked list is a data structure in which each element or *node* contains some kind of reference to its successor in the list. (See Fig. 6.4.)

If you have a head reference to the first node in the list, you can move through the list by looking at each node and then at the node's pointer field, so you know where the next node is located. When you finally encounter a pointer field with some special, predetermined end-of-list value, you stop the traversal of the list. In Fig. 6.4 we use a -1 in the pointer field to mark the end of the list.

When a list is made up of deleted records that have become *available space* within the file, the 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. It follows that there is no reason to order the avail list in any particular way. (As we see later, this situation changes for variable-length records.)

Stacks

The simplest way to handle a list is as a stack. A stack is a list in which all insertions and removals of nodes take place at one end of the list. So, if we

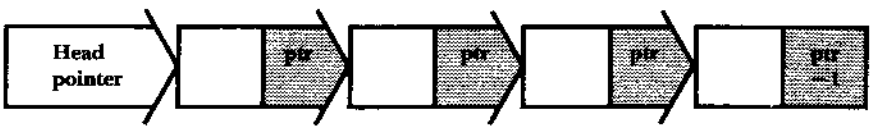
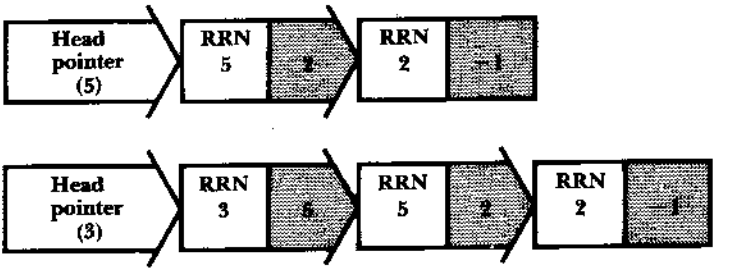


Figure 6.4 A linked list.

have an avail list managed as a stack that contains relative record numbers (RRN) 5 and 2, and then add RRN 3, it looks like this before and after the addition of the new node:



When a new node is added to the top or front of a stack, we say that it is pushed onto the stack. If the next thing that happens is a request for some available space, the request is filled by taking RRN 3 from the avail list. This is called popping the stack. The list returns to a state in which it contains only records 5 and 2.

Linking and Stacking Deleted Records

Now we can meet the two criteria for rapid access to reusable space from deleted records. We need

- A way to know immediately if there are empty slots in the file, and
- A way to jump directly to one of those slots if it exists.

Placing the deleted records on a stack meets both criteria. If the pointer to the top of the stack contains the end-of-list value, then we know that there are no empty slots and that we have to add new records by appending them to the end of the file. If the pointer to the stack top contains a valid node reference, then we know not only that a reusable slot is available, but also exactly where to find it.

Where do we keep the stack? Is it a separate list, perhaps maintained in a separate file, or is it somehow embedded within the data file? Once again, we need to be careful to distinguish between *physical* and *conceptual* structures. The deleted, available records are not moved anywhere when they are pushed onto the stack. They stay right where we need them, located in the file. 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 in a disk file rather than with memory addresses, the pointing is not done with *pointer* variables in the formal sense but through relative record numbers (RRNs).

Suppose we are working with a fixed-length record file that once contained seven records (RRNs 0–6). Furthermore, suppose that records 3 and 5 have been deleted, *in that order,* and that deleted records are marked by replacing the first field with an asterisk. We can then use the second field of a deleted record to hold the link to the next record on the avail list. Leaving out the details of the valid, in-use records, Fig. 6.5(a) shows how the file might look.

Record 5 is the first record on the avail list (top of the stack) as it is the record that is most recently deleted. Following the linked list, we see that record 5 points to record 3. Since the *link field* for record 3 contains -1, which is our end-of-list marker, we know that record 3 is the last slot available for reuse.

Figure 6.5(b) shows the same file after record 1 is also deleted. Note that the contents of all the other records on the avail list remain unchanged. Treating the list as a stack results in a minimal amount of list reorganization when we push and pop records to and from the list.

If we now add a new name to the file, it is placed in record 1, since RRN 1 is the first available record. The avail list would return to the

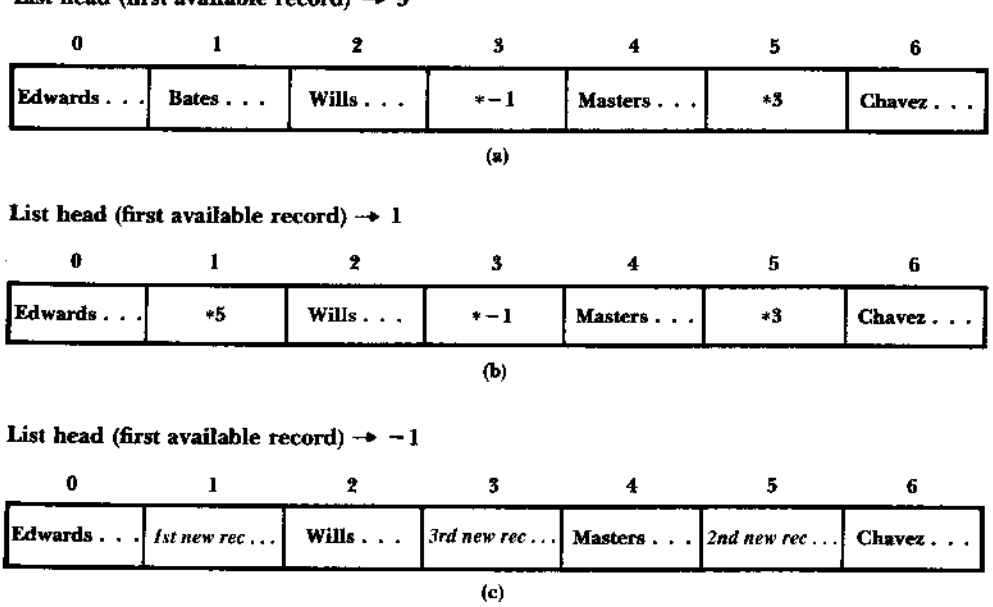


Figure 6.5 Sample file showing linked lists of deleted records. (a) After deletion of records 3 and 5, in that order. (b) After deletion of records 3, 5, and 1, in that order. (c) After insertion of three new records.

configuration shown in Fig. 6.5(a). Since there are still two record slots on the avail list, we could add two more names to the file without increasing the size of the file. After that, however, the avail list would be empty as shown in Fig. 6.5(c). 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.

Implementing Fixed-Length Record Deletion

Implementing mechanisms that place deleted records on a linked avail list and that treat the avail list as a stack is relatively straightforward. We need a suitable place to keep the RRN of the first available record on the avail list. Since this is information that is specific to the data file, it can be carried in a header record at the start of the file.

When we delete a record, we must be able to mark the record as deleted and then place it on the avail list. A simple way to do this is to place an * (or some other special mark) at the beginning of the record as a deletion mark, followed by the RRN of the next record on the avail list.

Once we have a list of available records within a file, we can reuse the space previously occupied by deleted records. For this we would write a single function that returns either (1) the RRN of a reusable record slot or (2) the RRN of the next record to be appended if no reusable slots are available.

6.2.3 Deleting Variable-Length Records

Now that we have a mechanism for handling an avail list of available space once records are deleted, let's apply this mechanism to the more complex problem of reusing space from deleted variable-length records. We have seen that to support record reuse through an avail list, we need

- A way to link the deleted records together into a list (that is, a place to put a link field);
- An algorithm for adding newly deleted records to the avail list; and
- 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

What kind of file structure do we need to support an avail list of variable-length records? Since we will want to delete whole records and then place

records on an avail list, we need a structure in which the record is a clearly defined entity. The file structure of `VariableLengthBuffer`, in which we define the length of each record by placing a byte count at the beginning of each record, will serve us well in this regard.

We can handle the contents of a deleted variable-length record just as we did with fixed-length records. That is, we can place a single asterisk in the first field, followed by a binary link field pointing to the next deleted record on the avail list. The avail list can be organized just as it was with fixed-length records, but with one difference: we cannot use relative record numbers for *links*. Since we cannot compute the byte offset of variable-length records from their RRNs, the links must contain the byte offsets themselves.

To illustrate, suppose we begin with a variable-length record file containing the three records for Ames, Morrison, and Brown introduced earlier. Figure 6.6(a) shows what the file looks like (minus the header) before any deletions, and Fig. 6.6(b) shows what it looks like after the deletion of the second record. The periods in the deleted record signify discarded characters.

Adding and Removing Records

Let's address the questions of adding and removing records to and from the list together, since they are clearly related. With fixed-length records we

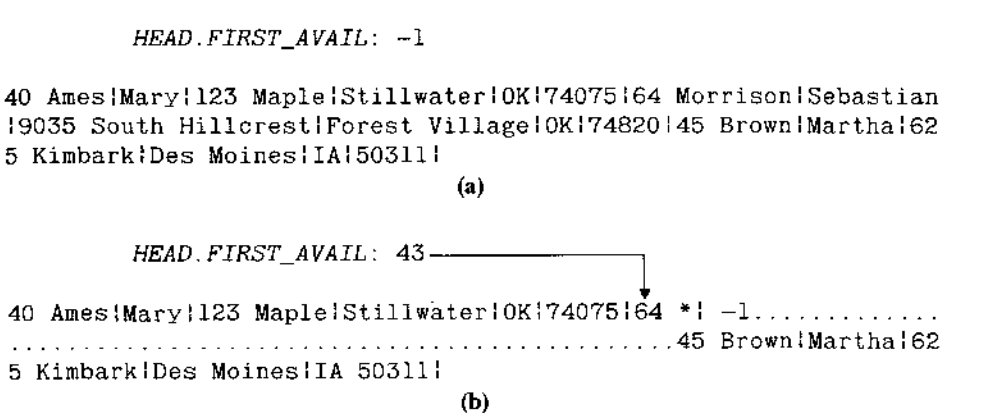


Figure 6.6 A sample file for illustrating variable-length record deletion. (a) Original sample file stored in variable-length format with byte count (header record not included). (b) Sample file after deletion of the second record (periods show discarded characters).

could access the avail list as a stack because one member of the avail list is just as usable as any other. That is not true when the record slots on the avail list differ in size, as they do in a variable-length record file. We now have an extra condition that must be met before we can reuse a record: the record must be the right size. For the moment we define *right size* as "big enough." Later we find that it is sometimes useful to be more particular about the meaning of *right size.*

It is possible, even likely, that we need to *search through* the avail list for a record slot that is the right size. We can't just pop the stack and expect the first available record to be big enough. Finding a proper slot on the avail list now means traversing the list until a record slot that is big enough to hold the new record is found.

For example, suppose the avail list contains the deleted record slots shown in Fig. 6.7(a), and a record that requires 55 bytes is to be added. Since the avail list is not empty, we traverse the records whose sizes are 47 (too small), 38 (too small), and 72 (big enough). Having found a slot big enough to hold our record, we remove it from the avail list by creating a new link that jumps over the record as shown in Fig. 6.7(b). If we had reached the end of the avail list before finding a record that was large enough, we would have appended the new record at the end of the file.

Because this procedure for finding a reusable record looks through the entire avail list if necessary, we do not need a sophisticated method for putting newly deleted records onto the list. If a record of the right size is

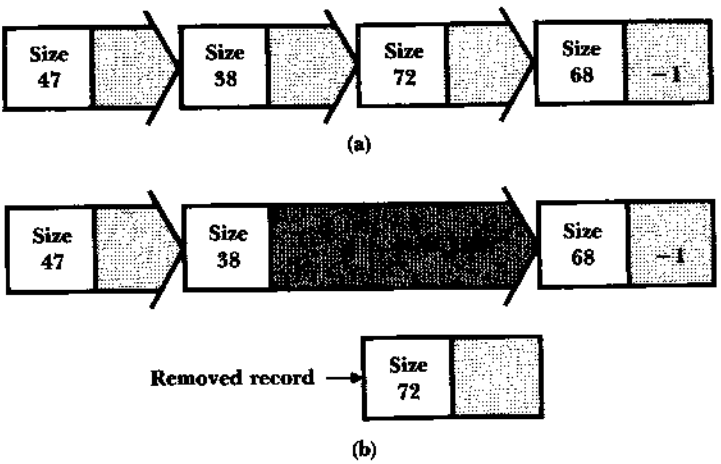


Figure 6.7 Removal of a record from an avail list with variable-length records. (a) Before removal. (b) After removal.

somewhere on this list, our get-available-record procedure eventually finds it. It follows that we can continue to push new members onto the front of the list, just as we do with fixed-length records.

Development of algorithms for adding and removing avail list records is left to you as part of the exercises found at the end of this chapter.

6.2.4 Storage Fragmentation

Let's look again at the fixed-length record version of our three-record file (Fig. 6.8). The dots at the ends of the records represent characters we use as padding between the last field and the end of the records. The padding is wasted space; it is part of the cost of using fixed-length records. Wasted space *within* a record is called *internal fragmentation.*

Clearly, we want to minimize internal fragmentation. If we are working with fixed-length records, we attempt this by choosing a record length that is as close as possible to what we need for each record. But unless the actual data is fixed in length, we have to put up with a certain amount of internal fragmentation in a fixed-length record file.

One of the attractions of variable-length records is that they minimize wasted space by doing away with internal fragmentation. The space set aside for each record is exactly as long as it needs to be. Compare the fixed-length example with the one in Fig. 6.9, which uses the variable-length record structure—a byte count followed by delimited data fields. The only space (other than the delimiters) that is not used for holding data in each record is the count field. If we assume that this field uses 2 bytes, this amounts to only 6 bytes for the three-record file. The fixed-length record file wastes 24 bytes in the very first record.

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

Figure 6.8 Storage requirements of sample file using 64-byte fixed-length records.

```
40 Ames|Mary|123 Maple|Stillwater|OK|74075|64 Morrison|Sebastian
|9035 South Hillcrest|Forest Village|OK|74820|45 Brown|Martha|62
5 Kimbark|Des Moines|IA|50311|
```

Figure 6.9 Storage requirements of sample file using variable-length records with a count field.

But before we start congratulating ourselves for solving the problem of wasted space due to internal fragmentation, we should consider what happens in a variable-length record file after a record is deleted and replaced with a shorter record. If the shorter record takes less space than the original record, internal fragmentation results. Figure 6.10 shows how the problem could occur with our sample file when the second record in the file is deleted and the following record is added:

```
Ham|Al|28 Elm|Ada|OK|70332|
```

It appears that escaping internal fragmentation is not so easy. The slot vacated by the deleted record is 37 bytes larger than is needed for the new record. Since we treat the extra 37 bytes as part of the new record, they are not on the avail list and are therefore unusable. But instead of keeping the 64-byte record slot intact, suppose we break it into two parts: one part to hold the new Ham record, and the other to be placed back on the avail list. Since we would take only as much space as necessary for the Ham record, there would be no internal fragmentation.

Figure 6.11 shows what our file looks like if we use this approach to insert the record for Al Ham. We steal the space for the Ham record *from the end* of the 64-byte slot and leave the first 35 bytes of the slot on the avail list. (The available space is 35 rather than 37 bytes because we need 2 bytes to form a new size field for the Ham record.) The 35 bytes still on the avail list can be used to hold yet another record. Figure 6.12 shows the effect of inserting the following 25-byte record:

```
Lee|Ed|Rt 2|Ada|OK|74820|
```

```
        HEAD.FIRST_AVAIL: 43 ──────────┐
                                       ▼
40 Ames |Mary|123 Maple|Stillwater|OK|74075|64 *| -1.............
.....................................................45 Brown|Martha|62
5 Kimbark|Des Moines|IA|50311|
```

(a)

```
        HEAD.FIRST_AVAIL: -1

40 Ames |Mary|123 Maple|Stillwater|OK|74075|64 Ham|Al|28 Elm|Ada|
OK|70332|.....................................45 Brown|Martha|62
5 Kimbark|Des Moines|IA|50311|
```

(b)

Figure 6.10 Illustration of fragmentation with variable-length records. (a) After deletion of the second record (unused characters in the deleted record are replaced by periods). (b) After the subsequent addition of the record for Al Ham.

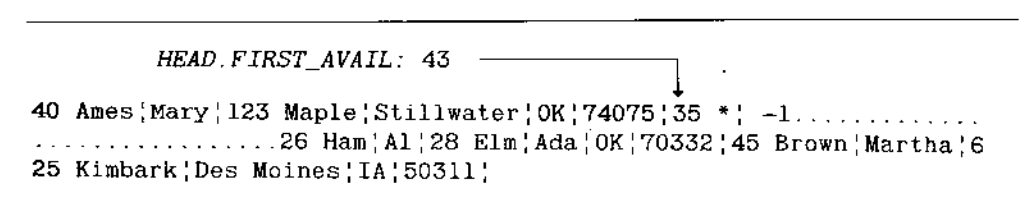


Figure 6.11 Combating internal fragmentation by putting the unused part of the deleted slot back on the avail list.

As we would expect, the new record is carved out of the 35-byte record that is on the avail list. The data portion of the new record requires 25 bytes, and we need 2 more bytes for another size field. This leaves 8 bytes in the record still on the avail list.

What are the chances of finding a record that can make use of these 8 bytes? Our guess would be that the probability is close to zero. These 8 bytes are not usable, even though they are not trapped inside any other record. This is an example of *external fragmentation.* The space is actually on the avail list rather than being locked inside some other record but is too fragmented to be reused.

There are some interesting ways to combat external fragmentation. One way, which we discussed at the beginning of this chapter, is *storage compaction.* We could simply regenerate the file when external fragmentation becomes intolerable. Two other approaches are as follows:

- If two record slots on the avail list are physically adjacent, combine them to make a single, larger record slot. This is called *coalescing the holes* in the storage space.
- Try to minimize fragmentation before it happens by adopting a placement strategy that the program can use as it selects a record slot from the avail list.

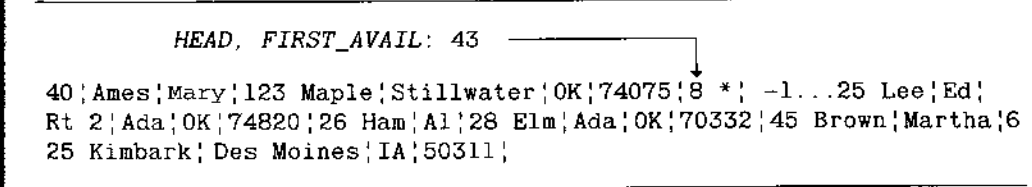


Figure 6.12 Addition of the second record into the slot originally occupied by a single deleted record.

Coalescing holes presents some interesting problems. The avail list is not kept in *physical* record order; if there are two deleted records that are physically adjacent, there is no reason to presume that they are linked adjacent to each other on the avail list. Exercise 15 at the end of this chapter provides a discussion of this problem along with a framework for developing a solution.

The development of better *placement strategies*, however, is a different matter. It is a topic that warrants a separate discussion, since the choice among alternative strategies is not as obvious as it might seem at first glance.

6.2.5 Placement Strategies

Earlier we discussed ways to add and remove variable-length records from an avail list. We add records by treating the avail list as a stack and putting deleted records at the front. When we need to remove a record slot from the avail list (to add a record to the file), we look through the list, starting at the beginning, until we either find a record slot that is big enough or reach the end of the list.

This is called a *first-fit* placement strategy. The least possible amount of work is expended when we place newly available space on the list, and we are not very particular about the closeness of fit as we look for a record slot to hold a new record. We accept the first available record slot that will do the job, regardless of whether the slot is ten times bigger than what is needed or whether it is a perfect fit.

We could, of course, develop a more orderly approach for placing records on the avail list by keeping them in either ascending or descending sequence by size. Rather than always putting the newly deleted records at the front of the list, these approaches involve moving through the list, looking for the place to insert the record to maintain the desired sequence.

If we order the avail list in *ascending* order by size, what is the effect on the closeness of fit of the records that are retrieved from the list? Since the retrieval procedure searches sequentially through the avail list until it encounters a record that is big enough to hold the new record, the first record encountered is the *smallest* record that will do the job. The fit between the available slot and the new record's needs would be as close as we can make it. This is called a *best-fit* placement strategy.

A best-fit strategy is intuitively appealing. There is, of course, a price to be paid for obtaining this fit. We end up having to search through at

least a part of the list—not only when we get records from the list, but also when we put newly deleted records on the list. In a real-time environment, the extra processing time could be significant.

A less obvious disadvantage of the best-fit strategy is related to the idea of finding the best possible fit and ensuring that the free area left over after inserting a new record into a slot is as small as possible. Often this remaining space is too small to be useful, resulting in external fragmentation. Furthermore, the slots that are least likely to be useful are the ones that will be placed toward the beginning of the list, making first-fit searches longer as time goes on.

These problems suggest an alternative strategy. What if we arrange the avail list so it is in *descending* order by size? Then the largest record slot on the avail list would always be at the head of the list. Since the procedure that retrieves records starts its search at the beginning of the avail list, it always returns the largest available record slot if it returns any slot at all. This is known as a *worst-fit* placement strategy. The amount of space in the record slot, beyond what is actually needed, is as large as possible.

A *worst-fit* strategy does not, at least initially, sound very appealing. But consider the following:

- The procedure for removing records can be simplified so it looks only at the first element of the avail list. If the first record slot is not large enough to do the job, none of the others will be.
- By extracting the space we need from the *largest* available slot, we are assured that the unused portion of the slot is as large as possible, decreasing the likelihood of external fragmentation.

What can you conclude from all of this? It should be clear that no one placement strategy is superior under all circumstances. The best you can do is formulate a series of general observations and then, given a particular design situation, try to select the strategy that seems most appropriate. Here are some suggestions. The judgment will have to be yours.

- Placement strategies make sense only with regard to volatile, variable-length record files. With fixed-length records, placement is simply not an issue.
- If space is lost due to *internal fragmentation*, the choice is between 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.

6.3 Finding Things Quickly: An Introduction to Internal Sorting and Binary Searching

This text begins with a discussion of the cost of accessing secondary storage. You may remember that the magnitude of the difference between accessing memory and seeking information on a fixed disk is such that, if we magnify the time for a memory access to twenty seconds, a similarly magnified disk access would take fifty-eight days.

So far we have not had to pay much attention to this cost. This section, then, marks a kind of turning point. Once we move from fundamental organizational issues to the matter of searching a file for a particular piece of information, the cost of a seek becomes a major factor in determining our approach. And what is true for searching is all the more true for sorting. If you have studied sorting algorithms, you know that even a good sort involves making many comparisons. If each of these comparisons involves a seek, the sort is agonizingly slow.

Our discussion of sorting and searching, then, goes beyond simply getting the job done. We develop approaches that minimize the number of disk accesses and therefore minimize the amount of time expended. This concern with minimizing the number of seeks continues to be a major focus throughout the rest of this text. This is just the beginning of a quest for ways to order and find things quickly.

6.3.1 Finding Things in Simple Field and Record Files

All of the programs we have written up to this point, despite any other strengths they offer, share a major failing: the only way to retrieve or find a record with any degree of rapidity is to look for it by relative record number. If the file has fixed-length records, knowing the RRN lets us compute the record's byte offset and jump to it using direct access.

But what if we do not know the byte offset or RRN of the record we want? How likely is it that a question about this file would take the form, "What is the record stored in RRN 23?" Not very likely, of course. We are much more likely to know the identity of a record by its key, and the question is more likely to take the form, "What is the record for Jane Kelly?"

Given the methods of organization developed so far, access by key implies a sequential search. What if there is no record containing the requested key? Then we would have to look through the entire file. What if we suspect that there might be more than one record that contains the key,

and we want to find them all? Once again, we would be doomed to looking at every record in the file. Clearly, we need to find a better way to handle keyed access. Fortunately, there are many better ways.

6.3.2 Search by Guessing: Binary Search

Suppose we are looking for a record for Jane Kelly in a file of one thousand fixed-length records, and suppose the file is sorted so the records appear in ascending order by key. We start by comparing KELLY JANE (the canonical form of the search key) with the middle key in the file, which is the key whose RRN is 500. The result of the comparison tells us which half of the file contains Jane Kelly's record. Next, we compare KELLY JANE with the middle key among records in the selected half of the file to find out which quarter of the file Jane Kelly's record is in. This process is repeated until either Jane Kelly's record is found or we have narrowed the number of potential records to zero.

This kind of searching is called binary searching. An algorithm for binary searching on a file of fixed-sized records is shown in Fig. 6.13. Binary searching takes at most ten comparisons—to find Jane Kelly's record if it is in the file, or to determine that it is not in the file. Compare this with a sequential search for the record. If there are one thousand records, then it takes at most one thousand comparisons to find a given record (or establish that it is not present); on the average, five hundred comparisons are needed.

We refer to the code in Fig. 6.13 as an algorithm, not a function, even though it is given in the form of a C++ function. This is because this is not a full implementation of binary search. Details of the implementation of the method are not given. From the code, we can infer that there must be a class `FixedRecordFile` that has methods `NumRecs` and `ReadByRRN` and that those methods have certain specific meaning. In particular, `NumRecs` must return the number of records in the `FixedRecordFile`, and `ReadByRRN` must read the record at a specific RRN and unpack it into a `RecType` object.

It is reasonable to suppose that a full implementation of binary search would be a template function with parameters for the type of the data record and the type of the key. It might also be a method of a fixed-record file class. Changing these details will not affect the algorithm and might not even require changes in the code. We do know, however, that in order to perform binary search, we must be able to read the file by relative record number, we must have assignment and key extraction methods on the data record type, and we must have relational operations on the key type.

```
int BinarySearch
   (FixedRecordFile & file, RecType & obj, KeyType & key)
// binary search for key
// if key found, obj contains corresponding record, 1 returned
// if key not found, 0 returned
{
   int low = 0; int high = file.NumRecs()-1;
   while (low <= high)
   {
      int guess = (high - low) / 2;
      file.ReadByRRN (obj, guess);
      if (obj.Key() == key) return 1; // record found
      if (obj.Key() < key) high = guess - 1;// search before guess
      else low = guess + 1;// search after guess
   }
   return 0; // loop ended without finding key
}
```

Figure 6.13 A binary search algorithm.

Figure 6.14 gives the minimum definitions that must be present to allow a successful compilation of `BinarySearch`. This includes a class `RecType` with a `Key` method that returns the key value of an object and class `KeyType` with equality and less-than operators. No further details of any of these classes need be given.

```
class KeyType
{public:
   int operator == (KeyType &); // equality operator
   int operator < (KeyType &); // less than operator
};

class RecType {public: KeyType Key();};

class FixedRecordFile
{public:
   int NumRecs();
   int ReadByRRN (RecType & record, int RRN);
};
```

Figure 6.14 Classes and methods that must be implemented to support the binary search algorithm.

This style of algorithm presentation is the object-oriented replacement for the pseudocode approach, which has been widely used to describe algorithms. Pseudocode is typically used to describe an algorithm without including all of the details of implementation. In Fig. 6.13, we have been able to present the algorithm without all of the details but in a form that can be passed through a compiler to verify that it is syntactically correct and conforms in its use of its related objects. The contrast between object-oriented design and pseudocode is that the object-oriented approach uses a specific syntax and a specific interface. The object-oriented approach is no harder to write but has significantly more detail.

6.3.3 Binary Search versus Sequential Search

In general, a binary search of a file with n records takes at most

$$\lfloor \log_2 n \rfloor + 1 \text{ comparisons}$$

and on average approximately

$$\lfloor \log_2 n \rfloor + 1/2 \text{ comparisons.}$$

A binary search is therefore said to be $O(\log_2 n)$. In contrast, you may recall that a sequential search of the same file requires at most n comparisons, and on average ?(tk?) n, which is to say that a sequential search is $O(n)$.

The difference between a binary search and a sequential search becomes even more dramatic as we increase the size of the file to be searched. If we double the number of records in the file, we double the number of comparisons required for sequential search; when binary search is used, doubling the file size adds only one more guess to our worst case. This makes sense, since we know that each guess eliminates half of the possible choices. So, if we tried to find Jane Kelly's record in a file of two thousand records, it would take at most

$$1 + \lfloor \log_2 2000 \rfloor = 11 \text{ comparisons}$$

whereas a sequential search would average

$$1/2\ n = 1000 \text{ comparisons}$$

and could take up to two thousand comparisons.

Binary searching is clearly a more attractive way to find things than sequential searching. But, as you might expect, there is a price to be paid before we can use binary searching: it works only when the list of records is ordered in terms of the key we are using in the search. So, to make use of binary searching, we have to be able to sort a list on the basis of a key.

Sorting is a very important part of file processing. Next, we will look at some simple approaches to sorting files in memory, at the same time introducing some important new concepts in file structure design. We take a second look at sorting in Chapter 8, when we deal with some tough problems that occur when files are too large to sort in memory.

6.3.4 Sorting a Disk File in Memory

Consider the operation of any internal sorting algorithm with which you are familiar. The algorithm requires multiple passes over the list that is to be sorted, comparing and reorganizing the elements. Some of the items in the list are moved a long distance from their original positions in the list. If such an algorithm were applied directly to data stored on a disk, it is clear that there would be a lot of jumping around, seeking, and rereading of data. This would be a very slow operation—unthinkably slow.

If the entire contents of the file can be held in memory, a very attractive alternative is to read the entire file from the disk into memory and then do the sorting there, using an *internal sort*. We still have to access the data on the disk, but this way we can access it sequentially, sector after sector, without having to incur the costs of a lot of seeking and of multiple passes over the disk.

This is one instance of a general class of solutions to the problem of minimizing disk usage: force your disk access into a sequential mode, performing the more complex, direct accesses in memory.

Unfortunately, it is often not possible to use this simple kind of solution, but when you can, you should take advantage of it. In the case of sorting, internal sorts are increasingly viable as the amount of memory space grows. A good illustration of an internal sort is the Unix `sort` utility, which sorts files in memory if it can find enough space. This utility is described in Chapter 8.

6.3.5 The Limitations of Binary Searching and Internal Sorting

Let's look at three problems associated with our "sort, then binary search" approach to finding things.

Problem 1: Binary Searching Requires More Than One or Two Accesses

In the average case, a binary search requires approximately $\lfloor \log_2 n \rfloor + 1/2$ comparisons. If each comparison requires a disk access, a series of binary

searches on a list of one thousand items requires, on the average, 9.5 accesses per request. If the list is expanded to one hundred thousand items, the average search length extends to 16.5 accesses. Although this is a tremendous improvement over the cost of a sequential search for the key, it is also true that 16 accesses, or even 9 or 10 accesses, is not a negligible cost. The cost of this searching is particularly noticeable and objectionable, if we are doing a large enough number of repeated accesses by key.

When we access records by relative record number rather than by key, we are able to retrieve a record with a single access. That is an order of magnitude of improvement over the ten or more accesses that binary searching requires with even a moderately large file. Ideally, we would like to approach RRN retrieval performance while still maintaining the advantages of access by key. In the following chapter, on the use of index structures, we begin to look at ways to move toward this ideal.

Problem 2: Keeping a File Sorted Is Very Expensive

Our ability to use a binary search has a price attached to it: we must keep the file in sorted order by key. Suppose we are working with a file to which we add records as often as we search for existing records. If we leave the file in unsorted order, conducting sequential searches for records, then on average each search requires reading through half the file. Each record addition, however, is very fast, since it involves nothing more than jumping to the end of the file and writing a record.

If, as an alternative, we keep the file in sorted order, we can cut down substantially on the cost of searching, reducing it to a handful of accesses. But we encounter difficulty when we add a record, since we want to keep all the records in sorted order. Inserting a new record into the file requires, on average, that we not only read through half the records, but that we also shift the records to open up the space required for the insertion. We are actually doing more work than if we simply do sequential searches on an unsorted file.

The costs of maintaining a file that can be accessed through binary searching are not always as large as in this example involving frequent record addition. For example, it is often the case that searching is required much more frequently than record addition. In such a circumstance, the benefits of faster retrieval can more than offset the costs of keeping the file sorted. As another example, there are many applications in which record additions can be accumulated in a transaction file and made in a batch mode. By sorting the list of new records before adding them to the main file, it is possible to merge them with the existing records. As we see in

Chapter 8, such merging is a sequential process, passing only once over each record in the file. This can be an efficient, attractive approach to maintaining the file.

So, despite its problems, there are situations in which binary searching appears to be a useful strategy. However, knowing the costs of binary searching also lets us see better solutions to the problem of finding things by key. Better solutions will have to meet at least one of the following conditions:

- They will not involve reordering of the records in the file when a new record is added, and
- They will be associated with data structures that allow for substantially more rapid, efficient reordering of the file.

In the chapters that follow we develop approaches that fall into each of these categories. Solutions of the first type can involve the use of simple indexes. They can also involve hashing. Solutions of the second type can involve the use of tree structures, such as a B-tree, to keep the file in order.

Problem 3: An Internal Sort Works Only on Small Files

Our ability to use binary searching is limited by our ability to sort the file. An internal sort works only if we can read the entire contents of a file into the computer's electronic memory. If the file is so large that we cannot do that, we need a different kind of sort.

In the following section we develop a variation on internal sorting called a *keysort*. Like internal sorting, keysort is limited in terms of how large a file it can sort, but its limit is larger. More important, our work on keysort begins to illuminate a new approach to the problem of finding things that will allow us to avoid the sorting of records in a file.

6.4 Keysorting

Keysort, sometimes referred to as *tag sort*, is based on the idea that when we sort a file in memory the only things that we really need to sort are the record keys; therefore, we do not need to read the whole file into memory during the sorting process. Instead, we read the keys from the file into memory, sort them, and then rearrange the records in the file according to the new ordering of the keys.

Since keysort never reads the complete set of records into memory, it can sort larger files than a regular internal sort, given the same amount of memory.

6.4.1 Description of the Method

To keep things simple, we assume that we are dealing with a fixed-length record file of the kind developed in Chapter 4, with a count of the number of records stored in a header record.

We present the algorithm in an object-oriented pseudocode. As in Section 6.3.3, we need to identify the supporting object classes. The file class (`FixedRecordFile`) must support methods `NumRecs` and `ReadByRRN`. In order to store the key RRN pairs from the file, we need a class `KeyRRN` that has two data members, `KEY` and `RRN`. Figure 6.15 gives the minimal functionality required by these classes.

The algorithm begins by reading the key RRN pairs into an array of `KeyRRN` objects. We call this array `KEYNODES[]`. Figure 6.16 illustrates the relationship between the array `KEYNODES[]` and the actual file at the

```
class FixedRecordFile
{public:
   int NumRecs();
   int ReadByRRN (RecType & record, int RRN);
   // additional methods required for keysort
   int Create (char * fileName);
   int Append (RecType & record);
};

class KeyRRN
// contains a pair (KEY, RRN)
{public:
   KeyType KEY; int RRN;
   KeyRRN();
   KeyRRN (KeyType key, int rrn);
};

int Sort (KeyRRN [], int numKeys); // sort array by key
```

Figure 6.15 Minimal functionality required for classes used by the keysort algorithm.

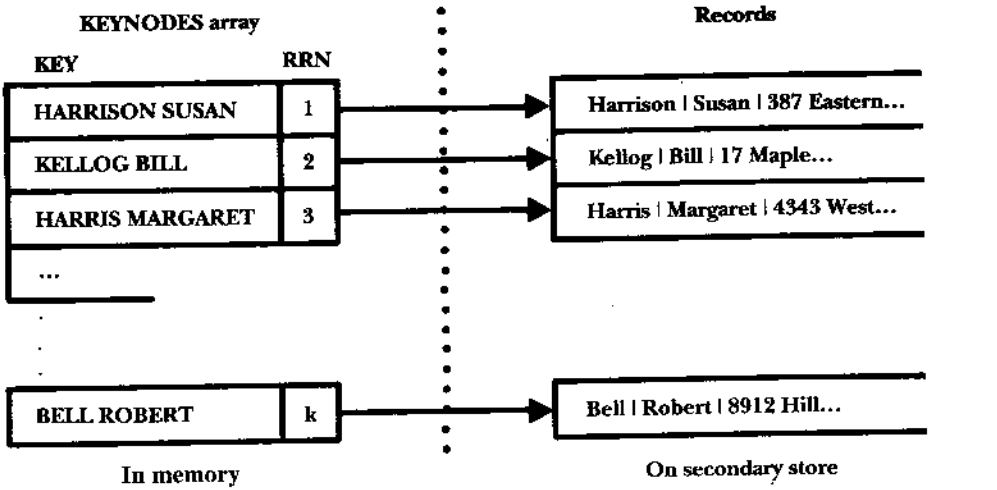


Figure 6.16 Conceptual view of KEYNODES array to be used in memory by internal sort routine and record array on secondary store.

time the keysort procedure begins. The RRN field of each array element contains the RRN of the record associated with the corresponding key.

The actual sorting process simply sorts the `KEYNODES[]` array according to the `KEY` field. This produces an arrangement like that shown in Fig. 6.17. The elements of `KEYNODES[]` are now sequenced in such a way that the first element has the RRN of the record that should be moved to the first position in the file, the second element identifies the record that should be second, and so forth.

Once `KEYNODES[]` is sorted, we are ready to reorganize the file according to this new ordering by reading the records from the input file and writing to a new file in the order of the `KEYNODES[]` array.

Figure 6.18 gives an algorithm for keysort. This algorithm works much the same way that a normal internal sort would work, but with two important differences:

- Rather than read an entire record into a memory array, we simply read each record into a temporary buffer, extract the key, then discard it; and
- When we are writing the records out in sorted order, we have to read them in a second time, since they are not all stored in memory.

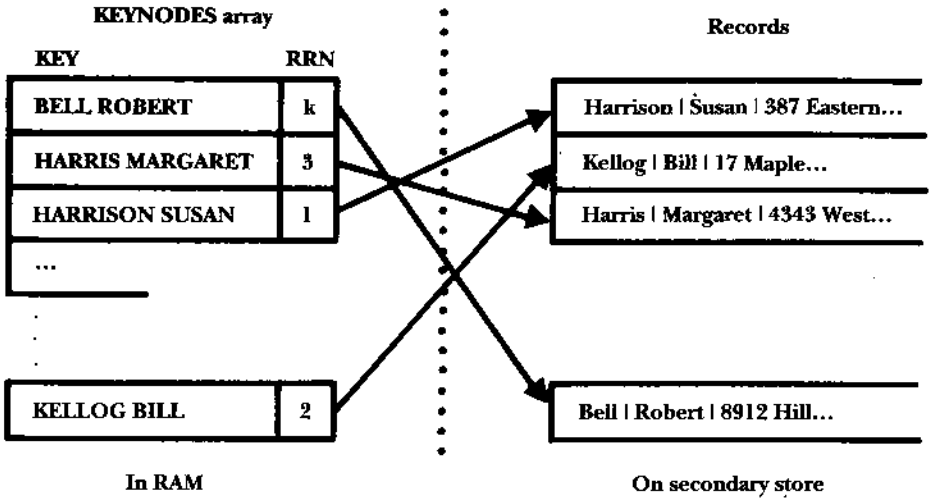


Figure 6.17 Conceptual view of KEYNODES array and file after sorting keys in memory.

```
int KeySort (FixedRecordFile & inFile, char * outFileName)
{
  RecType obj;
  KeyRRN * KEYNODES = new KeyRRN [inFile . NumRecs()];
  // read file and load Keys
  for (int i = 0; i < inFile . NumRecs(); i++)
  {
    inFile . ReadByRRN (obj, i);// read record i
    KEYNODES[i] = KeyRRN(obj.Key(),i);//put key and RRN into Keys
  }
  Sort (KEYNODES, inFile . NumRecs());// sort Keys
  FixedRecordFile outFile;// file to hold records in key order
  outFile . Create (outFileName);// create a new file
  // write new file in key order
  for (int j = 0; j < inFile . NumRecs(); j++)
  {
    inFile . ReadByRRN (obj, KEYNODES[j].RRN);//read in key order
    outFile . Append (obj);// write in key order
  }
  return 1;
}
```

Figure 6.18 Algorithm for keysort

6.4.2 Limitations of the Keysort Method

At first glance, keysorting appears to be an obvious improvement over sorting performed entirely in memory; it might even appear to be a case of getting something for nothing. We know that sorting is an expensive operation and that we want to do it in memory. Keysorting allows us to achieve this objective without having to hold the entire file in memory at once.

But, while reading about the operation of writing the records out in sorted order, even a casual reader probably senses a cloud on this apparently bright horizon. In keysort we need to read in the records a second time before we can write out the new sorted file. Doing something twice is never desirable. But the problem is worse than that.

Look carefully at the `for` loop that reads in the records before writing them out to the new file. You can see that we are not reading through the input file sequentially. Instead, we are working in sorted order, moving from the sorted KEYNODES[] to the RRNs of the records. Since we have to seek to each record and read it in before writing it back out, creating the sorted file requires as many random seeks into the input file as there are records. As we have noted a number of times, there is an enormous difference between the time required to read all the records in a file sequentially and the time required to read those same records if we must seek to each record separately. What is worse, we are performing all of these accesses in alternation with write statements to the output file. So, even the writing of the output file, which would otherwise appear to be sequential, involves seeking in most cases. The disk drive must move the head back and forth between the two files as it reads and writes.

The getting-something-for-nothing aspect of keysort has suddenly evaporated. Even though keysort does the hard work of sorting in memory, it turns out that creating a sorted version of the file from the map supplied by the KEYNODES[] array is not at all a trivial matter when the only copies of the records are kept on secondary store.

6.4.3 Another Solution: Why Bother to Write the File Back?

The idea behind keysort is an attractive one: why work with an entire record when the only parts of interest, as far as sorting and searching are concerned, are the fields used to form the key? There is a compelling parsimony behind this idea, and it makes keysorting look promising. The promise fades only when we run into the problem of rearranging all the records in the file so they reflect the new, sorted order.

It is interesting to ask whether we can avoid this problem by simply not bothering with the task that is giving us trouble. What if we just skip the time-consuming business of writing out a sorted version of the file? What if, instead, we simply write out a copy of the array of canonical key nodes? If we do without writing the records back in sorted order, writing out the contents of our KEYNODES[] array instead, we will have written a program that outputs an *index* to the original file. The relationship between the two files is illustrated in Fig. 6.19.

This is an instance of one of our favorite categories of solutions to computer science problems: if some part of a process begins to look like a bottleneck, consider skipping it altogether. Ask if you can do without it. Instead of creating a new, sorted copy of the file to use for searching, we have created a second kind of file, an index file, that is to be used in conjunction with the original file. If we are looking for a particular record, we do our binary search on the index file and then use the RRN stored in the index file record to find the corresponding record in the original file.

There is much to say about the use of index files, enough to fill several chapters. The next chapter is about the various ways we can use a simple index, which is the kind of index we illustrate here. In later chapters we talk about different ways of organizing the index to provide more flexible access and easier maintenance.

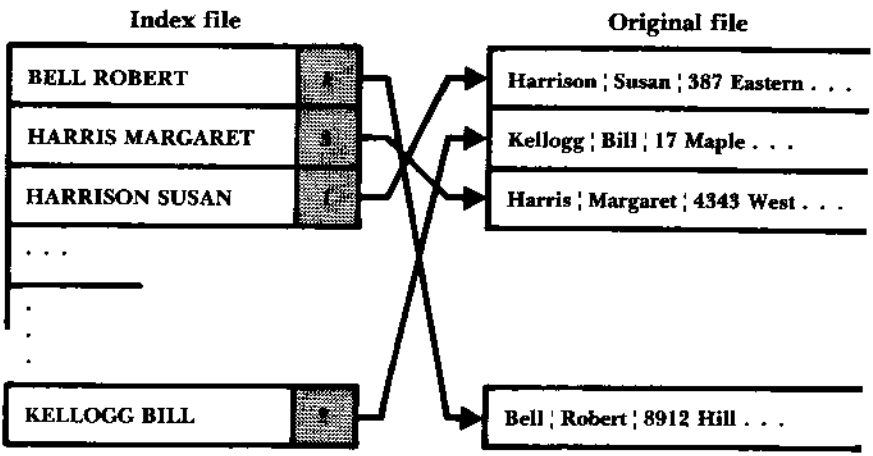


Figure 6.19 Relationship between the index file and the data file.

6.4.4 Pinned Records

In section 6.2 we discussed the problem of updating and maintaining files. Much of that discussion revolved around the problems of deleting records and keeping track of the space vacated by deleted records so it can be reused. An avail list of deleted record slots is created by linking all of the available slots together. This linking is done by writing a link field into each deleted record that points to the next deleted record. This link field gives very specific information about the exact physical location of the next available record.

When a file contains such references to the physical locations of records, we say that these records are *pinned*. You can gain an appreciation for this particular choice of terminology if you consider the effects of sorting one of these files containing an avail list of deleted records. A pinned record is one that cannot be moved. Other records in the same file or in some other file (such as an index file) contain references to the physical location of the record. If the record is moved, these references no longer lead to the record; they become *dangling pointers*, pointers leading to incorrect, meaningless locations in the file.

Clearly, the use of pinned records in a file can make sorting more difficult and sometimes impossible. But what if we want to support rapid access by key while still reusing the space made available by record deletion? One solution is to use an index file to keep the sorted order of the records while keeping the data file in its original order. Once again, the problem of finding things leads to the suggestion that we need to take a close look at the use of indexes, which, in turn, leads us to the next chapter.

SUMMARY

In this chapter we look at ways to organize or reorganize files to improve performance in some way.

Data compression methods are used to make files smaller by re-encoding data that goes into a file. Smaller files use less storage, take less time to transmit, and can often be processed faster sequentially.

The notation used for representing information can often be made more compact. For instance, if a 2-byte field in a record can take on only fifty values, the field can be encoded using 6 bits instead of 16. Another

form of compression called *run-length encoding* encodes sequences of repeating values rather than writing all of the values in the file.

A third form of compression assigns variable-length codes to values depending on how frequently the values occur. Values that occur often are given shorter codes, so they take up less space. *Huffman codes* are an example of variable-length codes.

Some compression techniques are *irreversible* in that they lose information in the encoding process. The Unix utilities `compress`, `uncompress`, `pack`, and `unpack` provide good compression in Unix.

A second way to save space in a file is to recover space in the file after it has undergone changes. A volatile file, one that undergoes many changes, can deteriorate very rapidly unless measures are taken to adjust the file organization to the changes. One result of making changes to files is storage fragmentation.

Internal fragmentation occurs when there is wasted space within a record. In a fixed-length record file, internal fragmentation can result when variable-length records are stored in fixed slots. It can also occur in a variable-length record file when one record is replaced by another record of a smaller size. *External fragmentation* occurs when holes of unused space between records are created, normally because of record deletions.

There are a number of ways to combat fragmentation. The simplest is *storage compaction,* which squeezes out the unused space caused from external fragmentation by sliding all of the undeleted records together. Compaction is generally done in a batch mode.

Fragmentation can be dealt with *dynamically* by reclaiming deleted space when records are added. The need to keep track of the space to be reused makes this approach more complex than compaction.

We begin with the problem of deleting fixed-length records. Since finding the first field of a fixed-length record is very easy, deleting a record can be accomplished by placing a special mark in the first field.

Since all records in a fixed-length record file are the same size, the reuse of deleted records need not be complicated. The solution we adopt consists of collecting all the available record slots into an *avail list.* The avail list is created by stringing together all the deleted records to form a *linked list* of deleted record spaces.

In a fixed-length record file, any one record slot is just as usable as any other slot; they are interchangeable. Consequently, the simplest way to maintain the linked avail list is to treat it as a *stack.* Newly available records are added to the avail list by *pushing* them onto the front of the

list; record slots are removed from the avail list by *popping* them from the front of the list.

Next, we consider the matter of deleting variable-length records. We still form a linked list of available record slots; but with variable-length records we need to be sure that a record slot is the right size to hold the new record. Our initial definition of *right size* is simply in terms of being big enough. Consequently, we need a procedure that can search through the avail list until it finds a record slot that is big enough to hold the new record. Given such a function and a complementary function that places newly deleted records on the avail list, we can implement a system that deletes and reuses variable-length records.

We then consider the amount and nature of fragmentation that develops inside a file due to record deletion and reuse. Fragmentation can happen *internally* if the space is lost because it is locked up inside a record. We develop a procedure that breaks a single, large, variable-length record slot into two or more smaller ones, using exactly as much space as is needed for a new record and leaving the remainder on the avail list. We see that, although this could decrease the amount of wasted space, eventually the remaining fragments are too small to be useful. When this happens, space is lost to *external fragmentation*.

There are a number of things that one can do to minimize external fragmentation. These include (1) *compacting* the file in a batch mode when the level of fragmentation becomes excessive; (2) *coalescing* adjacent record slots on the avail list to make larger, more generally useful slots; and (3) adopting a *placement strategy* to select slots for reuse in a way that minimizes fragmentation. Development of algorithms for coalescing holes is left as part of the exercises at the end of this chapter. Placement strategies need more careful discussion.

The placement strategy used up to this point by the variable-length record deletion and reuse procedures is a *first-fit* strategy. This strategy is simple: If the record slot is big enough, use it. By keeping the avail list in sorted order, it is easy to implement either of two other placement strategies:

- *Best fit*, in which a new record is placed in the smallest slot that is still big enough to hold it. This is an attractive strategy for variable-length record files in which the fragmentation is *internal*. It involves more overhead than other placement strategies.
- *Worst fit*, in which a new record is placed in the largest record slot available. The idea is to have the leftover portion of the slot be as large as possible.

There is no firm rule for selecting a placement strategy; the best one can do is use informed judgment based on a number of guidelines.

In the third major section of this chapter, we look at ways to find things quickly in a file through the use of a key. In preceding chapters it was not possible to access a record rapidly without knowing its physical location or relative record number. Now we explore some of the problems and opportunities associated with keyed direct access.

This chapter develops only one method of finding records by key—binary searching. Binary searching requires $O(\log_2 n)$ comparisons to find a record in a file with n records and hence is far superior to sequential searching. Since binary searching works only on a sorted file, a sorting procedure is an absolute necessity. The problem of sorting is complicated by the fact that we are sorting files on secondary storage rather than vectors in memory. We need to develop a sorting procedure that does not require seeking back and forth over the file.

Three disadvantages are associated with sorting and binary searching as developed up to this point:

- Binary searching is an enormous improvement over sequential searching, but it still usually requires more than one or two accesses per record. The need for fewer disk accesses becomes especially acute in applications where a large number of records are to be accessed by key.
- The requirement that the file be kept in sorted order can be expensive. For active files to which records are added frequently, the cost of keeping the file in sorted order can outweigh the benefits of binary searching.
- A memory sort can be used only on relatively small files. This limits the size of the files that we could organize for binary searching, given our sorting tools.

The third problem can be solved partially by developing more powerful sorting procedures, such as a keysort. This approach to sorting resembles a memory sort in most respects, but does not use memory to hold the entire file. Instead, it reads in only the keys from the records, sorts the keys, and then uses the sorted list of keys to rearrange the records on secondary storage so they are in sorted order.

The disadvantage to a keysort is that rearranging a file of n records requires n random seeks out to the original file, which can take much more time than a sequential reading of the same number of records. The inquiry into keysorting is not wasted, however. Keysorting naturally leads to the suggestion that we merely write the sorted list of keys off to

secondary storage, setting aside the expensive matter of rearranging the file. This list of keys, coupled with RRN tags pointing back to the original records, is an example of an index. We look at indexing more closely in Chapter 7.

This chapter closes with a discussion of another, potentially hidden, cost of sorting and searching. Pinned records are records that are referenced elsewhere (in the same file or in some other file) according to their physical position in the file. Sorting and binary searching cannot be applied to a file containing pinned records, since the sorting, by definition, is likely to change the physical position of the record. Such a change causes other references to this record to become inaccurate, creating the problem of dangling pointers.

KEY TERMS

Avail list. A list of the space, freed through record deletion, that is available for holding new records. In the examples considered in this chapter, this list of space took the form of a linked list of deleted records.

Best fit. A placement strategy for selecting the space on the avail list used to hold a new record. Best-fit placement finds the available record slot that is closest in size to what is needed to hold the new record.

Binary search. A binary search algorithm locates a key in a sorted list by repeatedly selecting the middle element of the list, dividing the list in half, and forming a new, smaller list from the half that contains the key. This process is continued until the selected element is the key that is sought.

Coalescence. If two deleted, available records are physically adjacent, they can be combined to form a single, larger available record space. This process of combining smaller available spaces into a larger one is known as *coalescing holes*. Coalescence is a way to counteract the problem of external fragmentation.

Compaction. A way of getting rid of all *external fragmentation* by sliding all the records together so there is no space lost between them.

Data compression. Encoding information in a file in such a way as to take up less space.

External fragmentation. A form of fragmentation that occurs in a file when there is unused space outside or between individual records.

First fit. A placement strategy for selecting a space from the avail list. First-fit placement selects the first available record slot large enough to hold the new record.

Fragmentation. The unused space within a file. The space can be locked within individual records *(internal fragmentation)* or between individual records *(external fragmentation)*.

Huffman code. A variable-length code in which the lengths of the codes are based on their probability of occurrence.

Internal fragmentation. A form of fragmentation that occurs when space is wasted in a file because it is locked up, unused, inside of records. Fixed-length record structures often result in internal fragmentation.

Irreversible compression. Compression in which information is lost.

Keysort. A method of sorting a file that does not require holding the entire file in memory. Only the keys are held in memory, along with pointers that tie these keys to the records in the file from which they are extracted. The keys are sorted, and the sorted list of keys is used to construct a new version of the file that has the records in sorted order. The primary advantage of a keysort is that it requires less memory than a memory sort. The disadvantage is that the process of constructing a new file requires a lot of seeking for records.

Linked list. A collection of nodes that have been organized into a specific sequence by means of references placed in each node that point to a single successor node. The *logical* order of a linked list is often different from the physical order of the nodes in the computer's memory.

Pinned record. A record is pinned when there are other records or file structures that refer to it by its physical location. It is pinned in the sense that we are not free to alter the physical location of the record: doing so destroys the validity of the physical references to the record. These references become useless dangling pointers.

Placement strategy. As used in this chapter, a placement strategy is a mechanism for selecting the space on the avail list that is to be used to hold a new record added to the file.

Redundancy reduction. Any form of compression that does not lose information.

Run-length encoding. A compression method in which runs of repeated codes are replaced by a count of the number of repetitions of the code, followed by the code that is repeated.

Stack. A kind of list in which all additions and deletions take place at the same end.

Variable-length encoding. Any encoding scheme in which the codes are of different lengths. More frequently occurring codes are given shorter lengths than frequently occurring codes. Huffman encoding is an example of variable-length encoding.

Worst fit. A placement strategy for selecting a space from the avail list. Worst-fit placement selects the largest record slot, regardless of how small the new record is. Insofar as this leaves the largest possible record slot for reuse, worst fit can sometimes help minimize *external fragmentation.*

FURTHER READINGS

A thorough treatment of data compression techniques can be found in Lynch (1985). The Lempel-Ziv method is described in Welch (1984). Huffman encoding is covered in many data structures texts and also in Knuth (1997).

Somewhat surprising, the literature concerning storage fragmentation and reuse often does not consider these issues from the standpoint of secondary storage. Typically, storage fragmentation, placement strategies, coalescing of holes, and garbage collection are considered in the context of reusing space within electronic random access memory. As you read this literature with the idea of applying the concepts to secondary storage, it is necessary to evaluate each strategy in light of the cost of accessing secondary storage. Some strategies that are attractive when used in electronic memory are too expensive on secondary storage.

Discussions about space management in memory are usually found under the heading "Dynamic Storage Allocation." Knuth (1997) provides a good, though technical, overview of the fundamental concerns associated with dynamic storage allocation, including placement strategies. Standish (1989) provides a more complete overview of the entire subject, reviewing much of the important literature on the subject.

This chapter only touches the surface of issues relating to searching and sorting files. A large part of the remainder of this text is devoted to exploring the issues in more detail, so one source for further reading is the present text. But there is much more that has been written about even the relatively simple issues raised in this chapter. The classic reference on sort-

ing and searching is Knuth (1998). Knuth provides an excellent discussion of the limitations of keysort methods. He also develops a very complete discussion of binary searching, clearly bringing out the analogy between binary searching and the use of binary trees.

EXERCISES

1. In our discussion of compression, we show how we can compress the "state name" field from 16 bits to 6 bits, yet we say that this gives us a space savings of 50 percent, rather than 62.5 percent, as we would expect. Why is this so? What other measures might we take to achieve the full 62.5 percent savings?
2. What is redundancy reduction? Why is run-length encoding an example of redundancy reduction?
3. What is the maximum run length that can be handled in the run-length encoding described in the text? If much longer runs were common, how might you handle them?
4. Encode each of the following using run-length encoding. Discuss the results, and indicate how you might improve the algorithm.

   a. 01 01 01 01 01 01 01 01 01 04 04 02 02 02 03 03 03 03 04 05 06 06 07

   b. 07 07 02 02 03 03 05 05 06 06 05 05 04 04
5. From Fig. 6.2, determine the Huffman code for the sequence "cdffe."
6. What is the difference between internal and external fragmentation? How can compaction affect the amount of internal fragmentation in a file? What about external fragmentation?
7. In-place compaction purges deleted records from a file without creating a separate new file. What are the advantages and disadvantages of in-place compaction compared withto compaction in which a separate compacted file is created?
8. Why is a best-fit placement strategy a bad choice if there is significant loss of space due to external fragmentation?
9. Conceive of an inexpensive way to keep a continuous record of the amount of fragmentation in a file. This fragmentation measure could be used to trigger the batch processes used to reduce fragmentation.
10. Suppose a file must remain sorted. How does this affect the range of placement strategies available?

11. Develop an algorithm in the style of Fig. 6.13 for performing in-place compaction in a variable-length record file that contains size fields at the start of each record. What operations must be added to class `RecordFile` to support this compaction algorithm?
12. Consider the process of updating rather than deleting a variable-length record. Outline a procedure for handling such updating, accounting for the update possibly resulting in either a longer or shorter record.
13. In Section 6.3, we raised the question of where to keep the stack containing the list of available records. Should it be a separate list, perhaps maintained in a separate file, or should it be embedded within the data file? We chose the latter organization for our implementation. What advantages and disadvantages are there to the second approach? What other kinds of file structures can you think of to facilitate various kinds of record deletion?
14. In some files, each record has a delete bit that is set to 1 to indicate that the record is deleted. This bit can also be used to indicate that a record is inactive rather than deleted. What is required to reactivate an inactive record? Could reactivation be done with the deletion procedures we have used?
15. In this chapter we outlined three general approaches to the problem of minimizing storage fragmentation: (a) implementation of a placement strategy, (b) coalescing of holes, and (c) compaction. Assuming an interactive programming environment, which of these strategies would be used on the fly, as records are added and deleted? Which strategies would be used as batch processes that could be run periodically?
16. Why do placement strategies make sense only with variable-length record files?
17. Compare the average case performance of binary search with sequential search for records, assuming
    a. That the records being sought are guaranteed to be in the file,
    b. That half of the time the records being sought are not in the file, and
    c. That half of the time the records being sought are not in the file and that missing records must be inserted.

    Make a table showing your performance comparisons for files of 5000, 10 000, 20 000, 50 000, and 100 000 records.

18. If the records in Exercise 17 are blocked with 30 records per block, how does this affect the performance of the binary and sequential searches?

19. An internal sort works only with files small enough to fit in memory. Some computing systems provide users who have an almost unlimited amount of memory with a memory management technique called *virtual memory*. Discuss the use of internal sorting to sort large files on systems that use virtual memory. Be sure to consider the disk activity that is required to support virtual memory.

20. Our discussion of keysorting covers the considerable expense associated with the process of actually creating the sorted output file, given the sorted vector of pointers to the canonical key nodes. The expense revolves around two primary areas of difficulty:

    a. Having to jump around in the input file, performing many seeks to retrieve the records in their new, sorted order; and
    b. Writing the output file at the same time we are reading the input file—jumping back and forth between the files can involve seeking.

    Design an approach to this problem usingthat uses buffers to hold a number of records and, therefore mitigating these difficulties. If your solution is to be viable, obviously the buffers must use less memory than a sort taking place entirely within electronic memory.

PROGRAMMING EXERCISES

Exercises 21–22 and 23–26 investigate the problem of implementing record deletion and update. It is very appropriate to combine them into one or two design and implementation projects.

21. Add method `Delete` to class `BufferFile` to support deletion of fixed-length records. Add a field to the beginning of each record to mark whether the record is active or deleted. Modify the Read and Append methods to react to this field. In particular, Read should either fail to read, if the current record is deleted, or read the next active record. You may need to modify classes `IOBuffer` and `FixedLengthRecord`.

22. Extend the implementation of Exercise 21 to keep a list of deleted records so that deleted records can be reused by the `Append` method. Modify the `Append` method to place a new record into a deleted

record, if one is available. You may consider adding a field to the file header to store the address of the head of the deleted list and using space in each deleted record to store the address of the next deleted record.

23. Repeat Exercise 21 for variable-length records.

24. Repeat Exercise 22 for variable-length records.

25. Add an `Update` method (or modify `Write`) to class `BufferFile` to support the correct replacement of the record in the current file position with a new record. Your implementation of these methods must properly handle the case in which where the size of the new record is different from that of the record it replaces. In the case where the new size is smaller, you may choose to make the necessary changes to allow the new record to occupy the space of the old record, even though not all bytes are used. Note that in this case, the record size in the file, and the buffer size may be different.

26. Improve the variable-length record deletion procedure from Exercise 24 so that it checks to see if the newly deleted record is contiguous with any other deleted records. If there is contiguity, coalesce the records to make a single, larger available record slot. Some things to consider as you address this problem are as follows:

    a. The avail list does not keep records arranged in physical order; the next record on the avail list is not necessarily the next deleted record in the physical file. Is it possible to merge these two views of the avail list, the physical order and the logical order, into a single list? If you do this, what placement strategy will you use?

    b. Physical adjacency can include records that precede as well as follow the newly deleted record. How will you look for a deleted record that precedes the newly deleted record?

    c. Maintaining two views of the list of deleted records implies that as you discover physically adjacent records you have to rearrange links to update the nonphysical avail list. What additional complications would we encounter if we were combining the coalescing of holes with a best-fit or worst-fit strategy?

27. Implement the `BinarySearch` function of Fig. 6.13 for class `Person` using the canonical form of the combination of last name and first name as the key. Write a driver program to test the function. Assume that the files are created with using class `RecordFile<Person>` using a fixed-length buffer.

28. Modify the `BinarySearch` function so that if the key is not in the file, it returns the relative record number that the key would occupy were it in the file. The function should also continue to indicate whether the key was found or not.

29. Write a driver that uses the new `BinarySearch` function developed in Exercise 28. If the sought-after key is in the file, the program should display the record contents. If the key is not found, the program should display a list of the keys that surround the position that the key would have occupied. You should be able to move backward or forward through this list at will. Given this modification, you do not have to remember an entire key to retrieve it. If, for example, you know that you are looking for someone named Smith, but cannot remember the person's first name, this new program lets you jump to the area where all the Smith records are stored. You can then scroll back and forth through the keys until you recognize the right first name.

30. Write an internal sort that can sort a variable-length record file created with class `BufferFile`.

PROGRAMMING PROJECT

This is the fourth part of the programming project. We add methods to delete records from files and update objects in files. This depends on the solution to Exercises 21–25. This part of the programming project is optional. Further projects do not depend on this part.

31. Use the `Delete` and `Update` operations described in Exercises 21–25 to produce files of student records that support delete and update.

32. Use the `Delete` and `Update` operations described in Exercises 21–25 to produce files of student records that support delete and update.

The next part of the programming project is in Chapter 7.