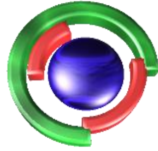
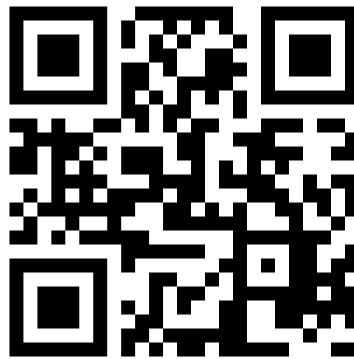
FUTURE VISION BIE

One Stop for All Study Materials & Lab Programs



By K B Hemanth Raj

Scan the QR Code to Visit the Web Page



Or

Visit : https://hemanthrajhemu.github.io

Gain Access to All Study Materials according to VTU,
CSE – Computer Science Engineering,
ISE – Information Science Engineering,
ECE - Electronics and Communication Engineering
& MORE...

Join Telegram to get Instant Updates: https://bit.ly/VTU_TELEGRAM

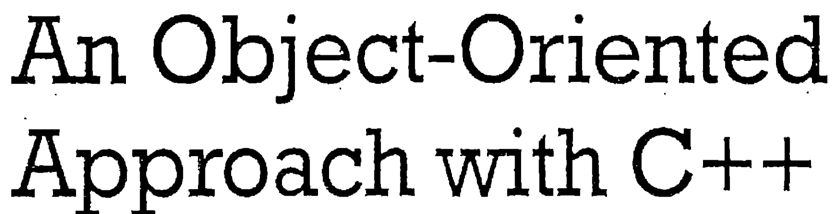
Contact: MAIL: futurevisionbie@gmail.com

INSTAGRAM: www.instagram.com/hemanthraj_hemu/

INSTAGRAM: www.instagram.com/futurevisionbie/

WHATSAPP SHARE: https://bit.ly/FVBIESHARE

File Structures



An Object-Oriented Approach with C++

Michael J. Folk
University of Illinois

Bill Zoellick
CAP Ventures

Greg Riccardi
Florida State University

ADDISON-WESLEY

Addison-Wesley is an imprint of Addison Wesley Longman, Inc.

Reading, Massachusetts • Harlow, England • Menlo Park, California
Berkeley, California • Don Mills, Ontario • Sydney
Bonn • Amsterdam • Tokyo • Mexico City

https://hemanthrajhemu.github.io

5.5 **Beyond Record Structures** 172
5.5.1 Abstract Data Models for File Access 172
5.5.2 Headers and Self-Describing Files 173
5.5.3 Metadata 174
5.5.4 Color Raster Images 176
5.5.5 Mixing Object Types in One File 179
5.5.6 Representation-Independent File Access 182
5.5.7 Extensibility 183
5.6 **Portability and Standardization** 184
5.6.1 Factors Affecting Portability 184
5.6.2 Achieving Portability 186
Summary 192 Key Terms 194 Further Readings 196
Exercises 196 Programming Exercises 199

Chapter 6 Organizing Files for Performance 201

6.1 **Data Compression** 203
6.1.1 Using a Different Notation 203
6.1.2 Suppressing Repeating Sequences 204
6.1.3 Assigning Variable-Length Codes 206
6.1.4 Irreversible Compression Techniques 207
6.1.5 Compression in Unix 207
6.2 **Reclaiming Space in Files** 208
6.2.1 Record Deletion and Storage Compaction 209
6.2.2 Deleting Fixed-Length Records for Reclaiming Space Dynamically 210
6.2.3 Deleting Variable-Length Records 214
6.2.4 Storage Fragmentation 217
6.2.5 Placement Strategies 220
6.3 **Finding Things Quickly: An Introduction to Internal Sorting and Binary Searching** 222
6.3.1 Finding Things in Simple Field and Record Files 222
6.3.2 Search by Guessing: Binary Search 223
6.3.3 Binary Search versus Sequential Search 225
6.3.4 Sorting a Disk File in Memory 226
6.3.5 The Limitations of Binary Searching and Internal Sorting 226
6.4 **Keysorting** 228
6.4.1 Description of the Method 229
6.4.2 Limitations of the Keysort Method 232
6.4.3 Another Solution: Why Bother to Write the File Back? 232
6.4.4 Pinned Records 234
Summary 234 Key Terms 238 Further Readings 240 Exercises 241
Programming Exercises 243 Programming Project 245

https://hemanthrajhemu.github.io

Chapter 7 Indexing 247

7.1 What Is an Index? 248
7.2 A Simple Index for Entry-Sequenced Files 249
7.3 Using Template Classes in C++ for Object I/O 253
7.4 Object-Oriented Support for Indexed, Entry-Sequenced Files of Data Objects 255
7.4.1 Operations Required to Maintain an Indexed File 256
7.4.2 Class TextIndexedFile 260
7.4.3 Enhancements to Class TextIndexedFile 261
7.5 Indexes That Are Too Large to Hold in Memory 264
7.6 Indexing to Provide Access by Multiple Keys 265
7.7 Retrieval Using Combinations of Secondary Keys 270
7.8 Improving the Secondary Index Structure: Inverted Lists 272
7.8.1 A First Attempt at a Solution 272
7.8.2 A Better Solution: Linking the List of References 274
7.9 Selective Indexes 278
7.10 Binding 279
Summary 280 Key Terms 282 Further Readings 283 Exercises 284
Programming and Design Exercises 285 Programming Project 286

Chapter 8 Cosequential Processing and the Sorting of Large Files 289

8.1 An Object-Oriented Model for Implementing Cosequential Processes 291
8.1.1 Matching Names in Two Lists 292
8.1.2 Merging Two Lists 297
8.1.3 Summary of the Cosequential Processing Model 299
8.2 Application of the Model to a General Ledger Program 301
8.2.1 The Problem 301
8.2.2 Application of the Model to the Ledger Program 304
8.3 Extension of the Model to Include Multiway Merging 309
8.3.1 A *K*-way Merge Algorithm 309
8.3.2 A Selective Tree for Merging Large Numbers of Lists 310
8.4 A Second Look at Sorting in Memory 311
8.4.1 Overlapping Processing and I/O: Heapsort 312
8.4.2 Building the Heap while Reading the File 313
8.4.3 Sorting While Writing to the File 316
8.5 Merging as a Way of Sorting Large Files on Disk 318
8.5.1 How Much Time Does a Merge Sort Take? 320
8.5.2 Sorting a File That Is Ten Times Larger 324

https://hemanthrajhemu.github.io

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*.

https://hemanthrajhemu.github.io

CHAPTER OUTLINE

6.1 Data Compression
- 6.1.1 Using a Different Notation
- 6.1.2 Suppressing Repeating Sequences
- 6.1.3 Assigning Variable-Length Codes
- 6.1.4 Irreversible Compression Techniques
- 6.1.5 Compression in Unix

6.2 Reclaiming Space in Files
- 6.2.1 Record Deletion and Storage Compaction
- 6.2.2 Deleting Fixed-Length Records for Reclaiming Space Dynamically
- 6.2.3 Deleting Variable-Length Records
- 6.2.4 Storage Fragmentation
- 6.2.5 Placement Strategies

6.3 Finding Things Quickly: An Introduction to Internal Sorting and Binary Searching
- 6.3.1 Finding Things in Simple Field and Record Files
- 6.3.2 Search by Guessing: Binary Search
- 6.3.3 Binary Search versus Sequential Search
- 6.3.4 Sorting a Disk File in Memory
- 6.3.5 The Limitations of Binary Searching and Internal Sorting

6.4 Keysorting
- 6.4.1 Description of the Method
- 6.4.2 Limitations of the Keysort Method
- 6.4.3 Another Solution: Why Bother to Write the File Back?
- 6.4.4 Pinned Records

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.

https://hemanthrajhemu.github.io

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.

https://hemanthrajhemu.github.io

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.

https://hemanthrajhemu.github.io

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.

https://hemanthrajhemu.github.io

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

https://hemanthrajhemu.github.io

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.

https://hemanthrajhemu.github.io

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.

https://hemanthrajhemu.github.io

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.

https://hemanthrajhemu.github.io

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

https://hemanthrajhemu.github.io

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

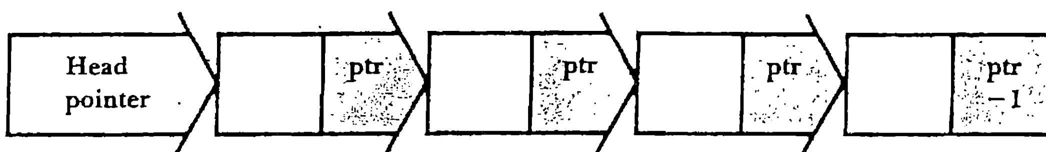
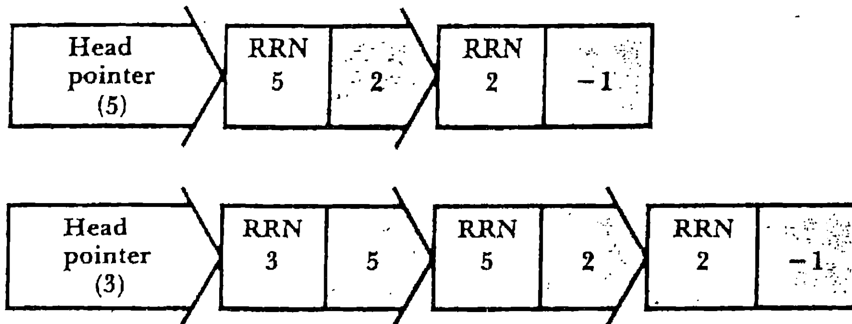


Figure 6.4 A linked list.

https://hemanthrajhemu.github.io

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).

https://hemanthrajhemu.github.io

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

List head (first available record) → 5

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

(a)

List head (first available record) → 1

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

(b)

List head (first available record) → −1

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

(c)

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.

https://hemanthrajhemu.github.io

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

https://hemanthrajhemu.github.io

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

```
        HEAD.FIRST_AVAIL: -1

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

(a)

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

(b)

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).

https://hemanthrajhemu.github.io

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

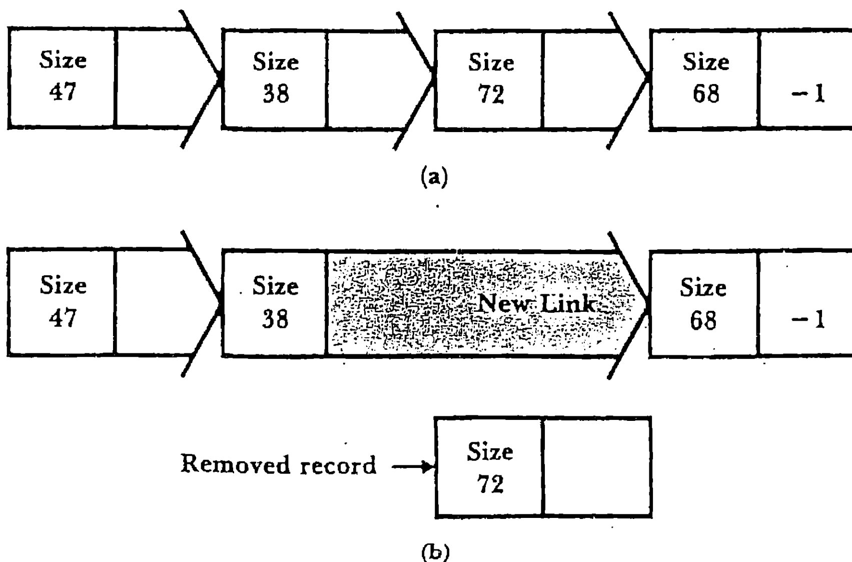


Figure 6.7 Removal of a record from an avail list with variable-length records. (a) Before removal. (b) After removal.

https://hemanthrajhemu.github.io

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.

https://hemanthrajhemu.github.io

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|
```

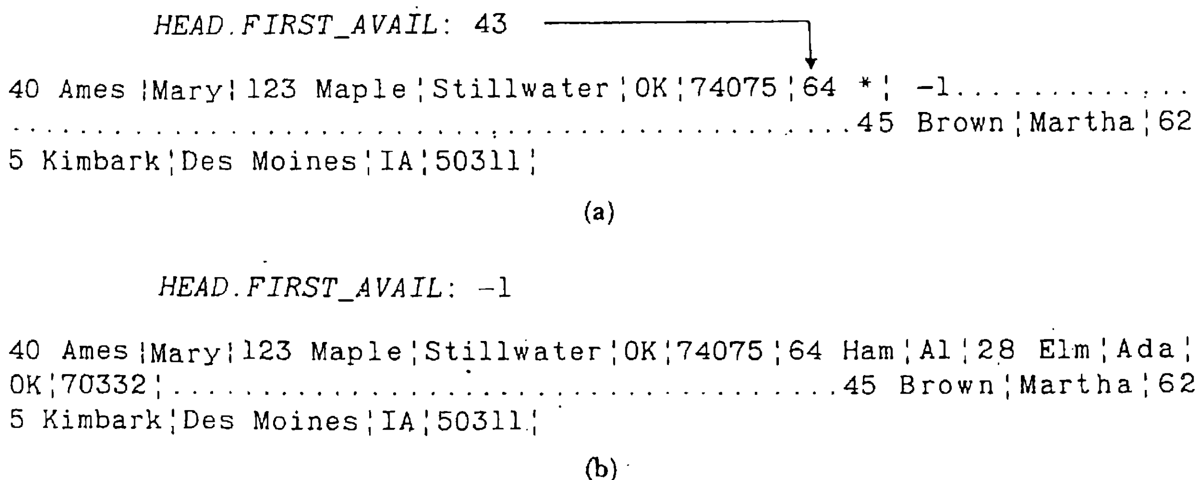


Figure 6.10 Illustration of fragmentation with variable-length records. (a) After deletion of the second record (unused characters in the deleted record are replaced by periods). (b) After the subsequent addition of the record for Al Ham.

https://hemanthrajhemu.github.io

```
        HEAD.FIRST_AVAIL: 43 ──────────────┐
                                           ↓
40 Ames|Mary|123 Maple|Stillwater|OK|74075|35 *| -1..............
...................26 Ham|Al|28 Elm|Ada|OK|70332|45 Brown|Martha|6
25 Kimbark|Des Moines|IA|50311|
```

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.

```
        HEAD.FIRST_AVAIL: 43 ──────────────┐
                                           ↓
40|Ames|Mary|123 Maple|Stillwater|OK|74075|8 *| -1...25 Lee|Ed|
Rt 2|Ada|OK|74820|26 Ham|Al|28 Elm|Ada|OK|70332|45 Brown|Martha|6
25 Kimbark|Des Moines|IA|50311|
```

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

https://hemanthrajhemu.github.io

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

https://hemanthrajhemu.github.io

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.

https://hemanthrajhemu.github.io

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,

https://hemanthrajhemu.github.io

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.

https://hemanthrajhemu.github.io

```
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.

https://hemanthrajhemu.github.io

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.

https://hemanthrajhemu.github.io

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

https://hemanthrajhemu.github.io

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

https://hemanthrajhemu.github.io

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.

https://hemanthrajhemu.github.io

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.

https://hemanthrajhemu.github.io

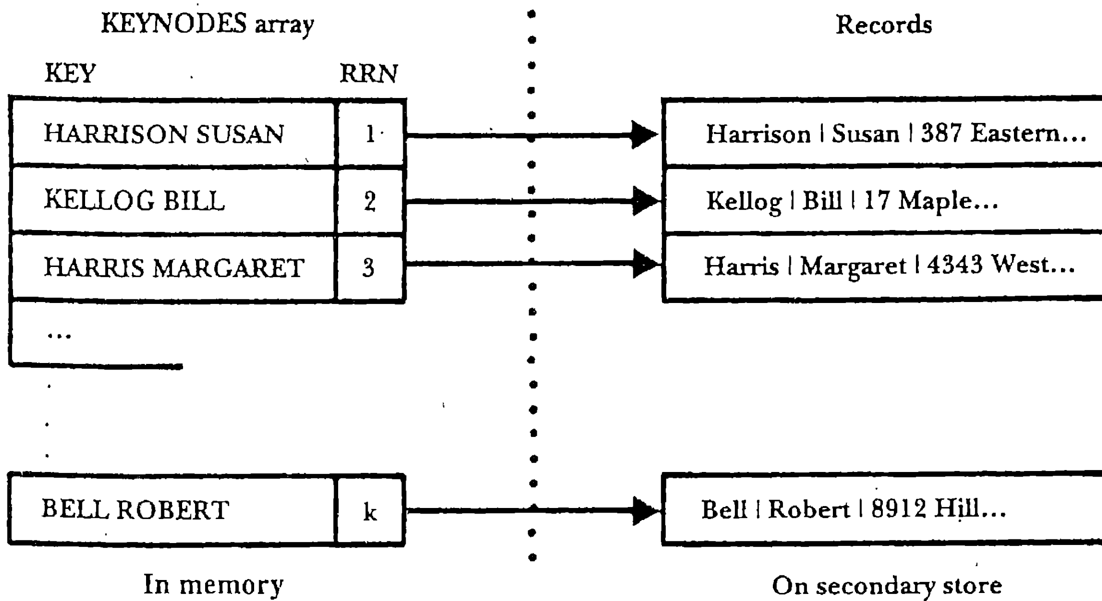


Figure 6.16 Conceptual view of KEYNODES array to be used in memory by internal sort routine and record array on secondary store.

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 KEYfield. 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.

https://hemanthrajhemu.github.io

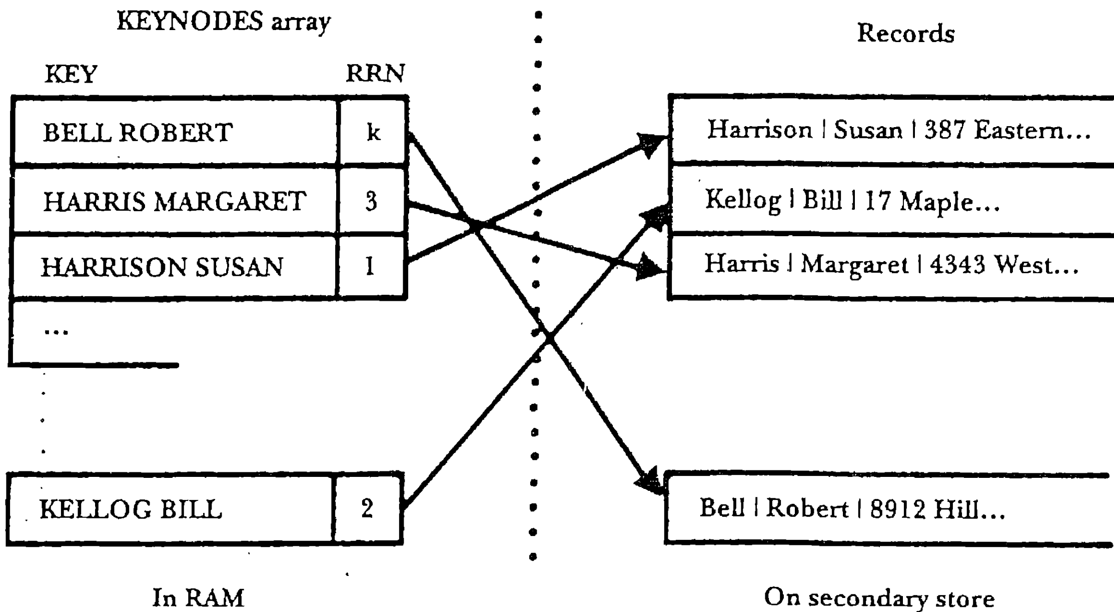


Figure 6.17 Conceptual view of KEYNODES array and file after sorting keys in memory.

```
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

https://hemanthrajhemu.github.io

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.

https://hemanthrajhemu.github.io

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.

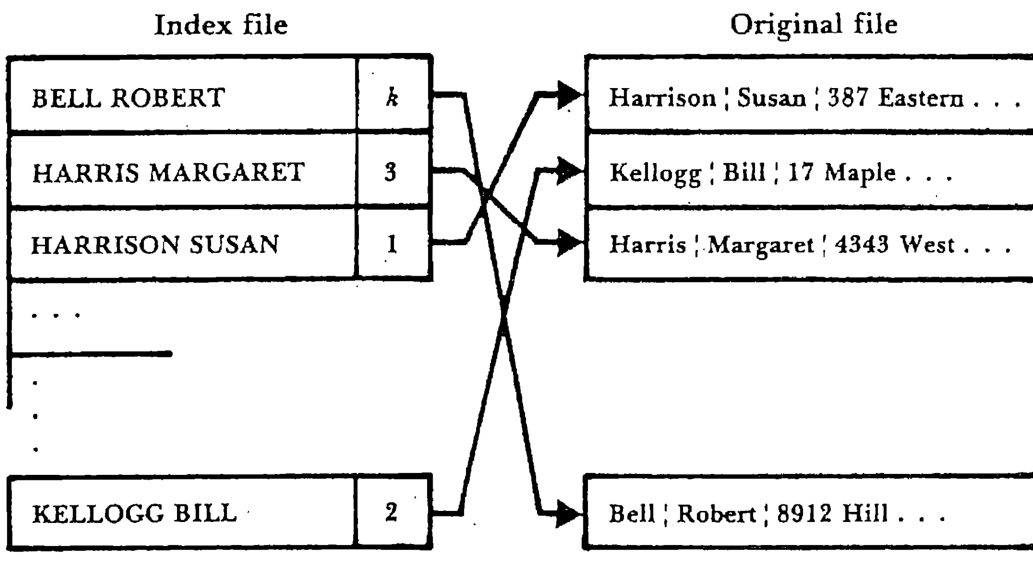


Figure 6.19 Relationship between the index file and the data file.

https://hemanthrajhemu.github.io

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

https://hemanthrajhemu.github.io

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

https://hemanthrajhemu.github.io

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.

https://hemanthrajhemu.github.io

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

https://hemanthrajhemu.github.io

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.

https://hemanthrajhemu.github.io

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.

https://hemanthrajhemu.github.io

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-

https://hemanthrajhemu.github.io

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?

https://hemanthrajhemu.github.io

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.

https://hemanthrajhemu.github.io

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

https://hemanthrajhemu.github.io

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.

https://hemanthrajhemu.github.io

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.

https://hemanthrajhemu.github.io

CHAPTER 7

Indexing

CHAPTER OBJECTIVES

- Introduce concepts of *indexing* that have broad applications in the design of file systems.
- Introduce the use of a *simple linear index* to provide rapid access to records in an entry-sequenced, variable-length record file.
- Investigate the implications of the use of indexes for file maintenance.
- Introduce the template features of C++.
- Discuss the object-oriented approach to indexed sequential files.
- Describe the use of indexes to provide access to records by more than one key.
- Introduce the idea of an *inverted list*, illustrating *Boolean operations* on lists.
- Discuss the issue of *when to bind* an index key to an address in the data file.
- Introduce and investigate the implications of *self-indexing* files.

https://hemanthrajhemu.github.io

CHAPTER OUTLINE

7.1 What Is an Index?
7.2 A Simple Index for Entry-Sequenced Files
7.3 Using Template Classes in C++ for Object I/O
7.4 Object-Oriented Support for Indexed, Entry-Sequenced Files of Data Objects
7.4.1 Operations Required to Maintain an Indexed File
7.4.2 Class TextIndexedFile
7.4.3 Enhancements to Class TextIndexedFile
7.5 Indexes That Are Too Large to Hold in Memory
7.6 Indexing to Provide Access by Multiple Keys
7.7 Retrieval Using Combinations of Secondary Keys
7.8 Improving the Secondary Index Structure: Inverted Lists
7.8.1 A First Attempt at a Solution
7.8.2 A Better Solution: Linking the List of References
7.9 Selective Indexes
7.10 Binding

7.1 What Is an Index?

The last few pages of many books contain an index. Such an index is a table containing a list of topics (keys) and numbers of pages where the topics can be found (reference fields).

All indexes are based on the same basic concept—keys and reference fields. The types of indexes we examine in this chapter are called *simple indexes* because they are represented using *simple arrays* of structures that contain the keys and reference fields. In later chapters we look at indexing schemes that use more complex data structures, especially trees. In this chapter, however, we want to emphasize that indexes can be very simple and still provide powerful tools for file processing.

The index to a book provides a way to find a topic quickly. If you have ever had to use a book that doesn't have a good index, you already know that an index is a desirable alternative to scanning through the book sequentially to find a topic. In general, indexing is another way to handle the problem we explored in Chapter 6: an index is a way to find things.

Consider what would happen if we tried to apply the previous chapter's methods, sorting and binary searching, to the problem of finding things in a book. Rearranging all the words in the book so they were in

https://hemanthrajhemu.github.io

alphabetical order certainly would make finding any particular term easier but would obviously have disastrous effects on the meaning of the book. In a sense, the terms in the book are pinned records. This is an absurd example, but it clearly underscores the power and importance of the index as a conceptual tool. Since it works by indirection, *an index lets you impose order on a file without rearranging the file.* This not only keeps us from disturbing pinned records, but also makes matters such as record addition much less expensive than they are with a sorted file.

Take, as another example, the problem of finding books in a library. We want to be able to locate books by a specific author, title, or subject area. One way of achieving this is to have three copies of each book and three separate library buildings. All of the books in one building would be sorted by author's name, another building would contain books arranged by title, and the third would have them ordered by subject. Again, this is an absurd example, but one that underscores another important advantage of indexing. Instead of using multiple arrangements, a library uses a card catalog. The card catalog is actually a set of three indexes, each using a different *key field,* and all of them using the same catalog number as a *reference field.* Another use of indexing, then, is to provide *multiple access paths* to a file.

We also find that indexing gives us *keyed access to variable-length record files.* Let's begin our discussion of indexing by exploring this problem of access to variable-length records and the simple solution that indexing provides.

One final note: the example data objects used in the following sections are musical recordings. This may cause some confusion as we use the term *record* to refer to an object in a file, and *recording* to refer to a data object. We will see how to get information about recordings by finding records in files. We've tried hard to make a distinction between these two terms. The distinction is between the file system view of the elements that make up files (records), and the user's or application's view of the objects that are being manipulated (recordings).

7.2 A Simple Index for Entry-Sequenced Files

Suppose we own an extensive collection of musical recordings, and we want to keep track of the collection through the use of computer files. For each recording, we keep the information shown in Fig. 7.1. Appendix G includes files `recordng.h` and `recordng.cpp` that define class

https://hemanthrajhemu.github.io

Identification number
Title
Composer or composers
Artist or artists
Label (publisher)

Figure 7.1 Contents of a data record.

`Recording`. Program `makerec.cpp` in Appendix G uses classes `DelimFieldBuffer` and `BufferFile` to create the file of `Recording` objects displayed in Fig. 7.2. The first column of the table contains the record addresses associated with each record in the file.

Suppose we formed a *primary key* for these recordings consisting of the initials for the company label combined with the recording's ID number. This will make a good primary key as it should provide a *unique* key for each entry in the file. We call this key the *Label ID*. The canonical form for the *Label ID* consists of the uppercase form of the Label field followed immediately by the ASCII representation of the `ID` number. For example,

```
LON2312
```

Record address	Label	ID number	Title	Composer(s)	Artist(s)
17	LON	2312	Romeo and Juliet	Prokofiev	Maazel
62	RCA	2626	Quartet in C Sharp Minor	Beethoven	Julliard
117	WAR	23699	Touchstone	Corea	Corea
152	ANG	3795	Symphony No. 9	Beethoven	Giulini
196	COL	38358	Nebraska	Springsteen	Springsteen
241	DG	18807	Symphony No. 9	Beethoven	Karajan
285	MER	75016	Coq d'Or Suite	Rimsky-Korsakov	Leinsdorf
338	COL	31809	Symphony No. 9	Dvorak	Bernstein
382	DG	139201	Violin Concerto	Beethoven	Ferras
427	FF	245	Good News	Sweet Honey in the Rock	Sweet Honey in the Rock

Figure 7.2 Contents of sample recording file.

https://hemanthrajhemu.github.io

How could we organize the file to provide rapid keyed access to individual records? Could we sort the file and then use binary searching? Unfortunately, binary searching depends on being able to jump to the middle record in the file. This is not possible in a variable-length record file because direct access by relative record number is not possible; there is no way to know where the middle record is in any group of records.

An alternative to sorting is to construct an index for the file. Figure 7.3 illustrates such an index. On the right is the data file containing information about our collection of recordings, with one variable-length data record per recording. Only four fields are shown (Label, ID number, Title, and Composer), but it is easy to imagine the other information filling out each record.

On the left is the index, each entry of which contains a *key* corresponding to a certain Label ID in the data file. Each key is associated with a *reference field* giving the address of the first byte of the corresponding data record. ANG3795, for example, corresponds to the reference field containing the number 152, meaning that the record containing full information on the recording with Label ID ANG3795 can be found starting at byte number 152 in the record file.

Index

Key	*Reference field*
ANG3795	152
COL31809	338
COL38358	196
DG139201	382
DG18807	241
FF245	427
LON2312	17
MER75016	285
RCA2626	62
WAR23699	117

Recording file

Address of record	*Actual data record*
17	LON | 2312 | Romeo and Juliet | Prokofiev | ...
62	RCA | 2626| Quartet in C Sharp Minor | Beethoven | ...
117	WAR | 23699 | Touchstone | Corea | ...
152	ANG | 3795 | Symphony No. 9 | Beethoven | ...
196	COL | 38358 | Nebraska | Springsteen | ...
241	DG | 18807 | Symphony No. 9 | Beethoven | ...
285	MER | 75016 | Coq d'Or Suite | Rimsky-Korsakov | ...
338	COL | 31809 | Symphony No. 9 | Dvorak | ...
382	DG | 139201 | Violin Concerto | Beethoven | ...
427	FF | 245 | Good News |Sweet Honey in the Rock | ...

Figure 7.3 Index of the sample recording file.

https://hemanthrajhemu.github.io

The structure of the index object is very simple. It is a list of pairs of fields: a key field and a byte-offset field. There is one entry in the index for each record in the data file. Class `TextIndex` of Fig. 7.4 encapsulates the index data and index operations. The full implementation of class `TextIndex` is given in files `textind.h` and `textind.cpp` of Appendix G. An index is implemented with arrays to hold the keys and record references. Each object is declared with a maximum number of entries and can be used for unique keys (no duplicates) and for nonunique keys (duplicates allowed). The methods `Insert` and `Search` do most of the work of indexing. The protected method `Find` locates the element key and returns its index. If the key is not in the index, `Find` returns -1. This method is used by `Insert`, `Remove`, and `Search`.

A C++ feature used in this class is the *destructor*, method `~TextIndex`. This method is automatically called whenever a `TextIndex` object is deleted, either because of the return from a function that includes the declaration of a `TextIndex` object or because of explicit deletion of an object created dynamically with `new`. The role of the destructor is to clean up the object, especially when it has dynamically created data members. In the case of class `TextIndex`, the protected members `Keys` and `RecAddrs` are created dynamically by the constructor and should be deleted by the destructor to avoid an obvious memory leak:

```
TextIndex::~TextIndex (){delete Keys; delete RecAddrs;}
```

```
class TextIndex
{public:
   TextIndex (int maxKeys = 100, int unique = 1);
   int Insert (const char * key, int recAddr); // add to index
   int Remove (const char * key); // remove key from index
   int Search (const char * key) const;
      // search for key, return recaddr
   void Print (ostream &) const;
protected:
   int MaxKeys; // maximum number of entries
   int NumKeys; // actual number of entries
   char * * Keys; // array of key values
   int * RecAddrs; // array of record references
   int Find (const char * key) const;
   int Init (int maxKeys, int unique);
   int Unique; // if true, each key must be unique in the index
};
```

Figure 7.4 Class TextIndex.

https://hemanthrajhemu.github.io

Note also that the index is sorted, whereas the data file is not. Consequently, although Label ID ANG3795 is the first entry in the index, it is not necessarily the first entry in the data file. In fact, the data file is *entry sequenced,* which means that the records occur in the order they are entered into the file. As we will see, the use of an entry-sequenced file can make record addition and file maintenance much simpler than the case with a data file that is kept sorted by some key.

Using the index to provide access to the data file by Label ID is a simple matter. The code to use our classes to retrieve a single record by key from a recording file is shown in the function `RetrieveRecording`:

```
int RetrieveRecording (Recording & recording, char * key,
       TextIndex & RecordingIndex, BufferFile & RecordingFile)
// read and unpack the recording, return TRUE if succeeds
{ int result;
  result = RecordingFile . Read (RecordingIndex.Search(key));
  if (result == -1) return FALSE;
  result = recording.Unpack (RecordingFile.GetBuffer());
  return result;
}
```

With an open file and an index to the file in memory, `RetrieveRecording` puts together the index search, file read, and buffer unpack operations into a single function.

Keeping the index in memory as the program runs also lets us find records by key more quickly with an indexed file than with a sorted one since the binary searching can be performed entirely in memory. Once the byte offset for the data record is found, a single seek is all that is required to retrieve the record. The use of a sorted data file, on the other hand, requires a seek for each step of the binary search.

7.3 Using Template Classes in C++ for Object I/O

A good object-oriented design for a file of objects should provide operations to read and write data objects without having to go through the intermediate step of packing and unpacking buffers. In Chapter 4, we supported I/O for data with the buffer classes and class `BufferFile`. In order to provide I/O for objects, we added `Pack` and `Unpack` methods to our `Person` object class. This approach gives us the required functionality but

https://hemanthrajhemu.github.io

stops short of providing a read operation whose arguments are a file and a data object. We want a class `RecordFile` that makes the following code possible:

```
Person p; RecordFile pFile;   pFile . Read (p);
Recording r;   RecordFile rFile;   rFile . Read (r);
```

The major difficulty with defining class `RecordFile` is making it possible to support files for different record types without having to modify the class. Is it possible that class `RecordFile` can support read and unpack for a `Person` and a `Recording` without change? Certainly the objects are different; they have different unpacking methods. Virtual function calls do not help because `Person` and `Recording` do not have a common base type. It seems that class `RecordFile` needs to be parameterized so different versions of the class can be constructed for different types of data objects.

It is the C++ *template* feature that supports parameterized function and class definitions, and `RecordFile` is a template class. As shown in Fig. 7.5, class `RecordFile` includes the parameter `RecType`, which is used as the argument type for the read and write methods of the class. Class `RecordFile` is derived from `BufferFile`, which provides most of the functionality. The constructor for `RecordFile` is given inline and simply calls the `BufferFile` constructor.

The definitions of `pFile` and `rFile` just given are not consistent with use of a template class. The actual declarations and calls are:

```
RecordFile <Person> pFile;     pFile . Read (p);
RecordFile <Recording> rFile;  rFile . Read (p);
```

```
template <class RecType>
class RecordFile: public BufferFile
{public:
   int Read (RecType & record, int recaddr = -1);
   int Write (const RecType & record, int recaddr = -1);
   int Append (const RecType & record);
   RecordFile (IOBuffer & buffer): BufferFile (buffer) {}
};
// The template parameter RecType must have the following methods
// int Pack (IOBuffer &); pack record into buffer
// int Unpack (IOBuffer &); unpack record from buffer
```

Figure 7.5 Template Class RecordFile.

https://hemanthrajhemu.github.io

Object rFile is of type RecordFile<Recording>, which is an *instance* of class RecordFile. The call to rFile.Read looks the same as the call to pFile.Read, and the two methods share the same source code, but the implementations of the classes are somewhat different. In particular, the Pack and Unpack methods of class Recording are used for methods of object rFile, but Person methods are used for pFile.

The implementation of method Read of class RecordFile is given in Fig. 7.6; the implementation of all the methods are in file recfile.h in Appendix G. The method makes use of the Read method of BufferFile and the Unpack method of the parameter RecType. A new version of RecordFile::Read is created by the C++ compiler for each instance of RecordFile. The call rFile.Read(r) calls Recording::Unpack, and the call pFile.Read(p) calls Person::Unpack.

Class RecordFile accomplishes the goal of providing object-oriented I/O for data. Adding I/O to an existing class (class Recording, for example) requires three steps:

1. Add methods Pack and Unpack to class Recording.
2. Create a buffer object to use in the I/O:
   ```
   DelimFieldBuffer Buffer;
   ```
3. Declare an object of type RecordFile<Recording>:
   ```
   RecordFile<Recording> rFile (Buffer);
   ```

Now we can directly open a file and read and write objects of class Recording:

```
Recording r1, r2;
rFile . Open ("myfile");
rFile . Read (r1);
rFile . Write (r2);
```

7.4 Object-Oriented Support for Indexed, Entry-Sequenced Files of Data Objects

Continuing with our object-oriented approach to I/O, we will add indexed access to the sequential access provided by class RecordFile. A new class, IndexedFile, extends RecordFile with Update and

https://hemanthrajhemu.github.io

```
template <class RecType>
int RecordFile<RecType>::Read (RecType & record, int recaddr)
{
   int writeAddr, result;
   writeAddr = BufferFile::Read (recaddr);
   if (!writeAddr) return -1;
   result = record . Unpack (Buffer); //RecType::Unpack
   if (!result) return -1;
   return writeAddr;
}
```

Figure 7.6 Implementation of RecordFile::Read.

Append methods that maintain a primary key index of the data file and a Read method that supports access to object by key.

So far, we have classes TextIndex, which supports maintenance and search by primary key, and RecordFile, which supports create, open, and close for files as well as read and write for data objects. We have already seen how to create a primary key index for a data file as a memory object. There are still two issues to address:

- How to make a persistent index of a file. That is, how to store the index in a file when it is not in memory.
- How to guarantee that the index is an accurate reflection of the contents of the data file.

7.4.1 Operations Required to Maintain an Indexed File

The support and maintenance of an entry-sequenced file coupled with a simple index requires the operations to handle a number of different tasks. Besides the RetrieveRecording function described previously, other operations used to find things by means of the index include the following:

- Create the original empty index and data files,
- Load the index file into memory before using it,
- Rewrite the index file from memory after using it,
- Add data records to the data file,
- Delete records from the data file,

https://hemanthrajhemu.github.io

- Update records in the data file, and
- Update the index to reflect changes in the data file.

A great benefit of our object-oriented approach is that everything we need to implement these operations is already available in the methods of our classes. We just need to glue them together. We begin by identifying the methods required for each of these operations. We continue to use class `Recording` as our example data class.

Creating the Files

Two files must be created: a data file to hold the data objects and an index file to hold the primary key index. Both the index file and the data file are created as empty files, with header records and nothing else. This can be accomplished quite easily using the `Create` method implemented in class `BufferFile`. The data file is represented by an object of class `RecordFile<Recording>`. The index file is a `BufferFile` of fixed-size records, as described below. As an example of the manipulation of index files, program `makeind.cpp` of Appendix G creates an index file from a file of recordings.

Loading the Index into Memory

Both loading (reading) and storing (writing) objects is supported in the `IOBuffer` classes. With these buffers, we can make files of index objects. For this example, we are storing the full index in a single object, so our index file needs only one record. As our use of indexes develops in the rest of the book, we will make extensive use of multiple record index files.

We need to choose a particular buffer class to use for our index file. We define class `TextIndexBuffer` as a derived class of `FixedFieldBuffer` to support reading and writing of index objects. `TextIndexBuffer` includes pack and unpack methods for index objects. This style is an alternative to adding these methods to the data class, which in this case is `TextIndexBuffer`. The full implementation of class `TextIndexBuffer` is in files `tindbuff.h` and `tindbuff.cpp` in Appendix G.

Rewriting the Index File from Memory

As part of the `Close` operation on an `IndexedFile`, the index in memory needs to be written to the index file. This is accomplished using the `Rewind` and `Write` operations of class `BufferFile`.

https://hemanthrajhemu.github.io

It is important to consider what happens if this rewriting of the index does not take place or if it takes place incompletely. Programs do not always run to completion. A program designer needs to guard against power failures, the operator turning the machine off at the wrong time, and other such disasters. One of the dangers associated with reading an index into memory and then writing it out when the program is over is that the copy of the index on disk will be out of date and incorrect if the program is interrupted. It is imperative that a program contain at least the following two safeguards to protect against this kind of error:

- There should be a mechanism that permits the program to know when the index is out of date. One possibility involves setting a status flag as soon as the copy of the index in memory is changed. This status flag could be written into the header record of the index file on disk as soon as the index is read into memory and subsequently cleared when the index is rewritten. All programs could check the status flag before using an index. If the flag is found to be set, the program would know that the index is out of date.
- If a program detects that an index is out of date, the program must have access to a procedure that reconstructs the index from the data file. This should happen automatically and take place before any attempt is made to use the index.

Record Addition

Adding a new record to the data file requires that we also add an entry to the index. Adding to the data file itself uses `RecordFile<Recording>::Write`. The record key and the resulting record reference are then inserted into the index record using `TextIndex.Insert`.

Since the index is kept in sorted order by key, insertion of the new index entry probably requires some rearrangement of the index. In a way, the situation is similar to the one we face as we add records to a sorted data file. We have to shift or slide all the entries with keys that come in order after the key of the record we are inserting. The shifting opens up a space for the new entry. The big difference between the work we have to do on the index entries and the work required for a sorted data file is that the index is contained *wholly in memory.* All of the index rearrangement can be done without any file access. The implementation of `TextIndex::Insert` is given in file `textind.cpp` of Appendix G.

https://hemanthrajhemu.github.io

Record Deletion

In Chapter 6 we described a number of approaches to deleting records in variable-length record files that allow for the reuse of the space occupied by these records. These approaches are completely viable for our data file because, unlike a sorted data file, the records in this file need not be moved around to maintain an ordering on the file. This is one of the great advantages of an indexed file organization: we have rapid access to individual records by key without disturbing pinned records. In fact, the indexing itself pins all the records. The implementation of data record deletion is not included in this text but has been left as exercises.

Of course, when we delete a record from the data file, we must also delete the corresponding entry from our index, using `TextIndex::Delete`. Since the index is in memory during program execution, deleting the index entry and shifting the other entries to close up the space may not be an overly expensive operation. Alternatively, we could simply mark the index entry as deleted, just as we might mark the corresponding data record. Again, see `textind.cpp` for the implementation of `TextIndex::Delete`.

Record Updating

Record updating falls into two categories:

- *The update changes the value of the key field.* This kind of update can bring about a reordering of the index file as well as the data file. Conceptually, the easiest way to think of this kind of change is as a deletion followed by an insertion. This delete/insert approach can be implemented while still providing the program user with the view that he or she is merely changing a record.
- *The update does not affect the key field.* This second kind of update does not require rearrangement of the index file but may well involve reordering of the data file. If the record size is unchanged or decreased by the update, the record can be written directly into its old space. But if the record size is increased by the update, a new slot for the record will have to be found. In the latter case the starting address of the rewritten record must replace the old address in the corresponding `RecAddrs` element. Again, the delete/insert approach to maintaining the index can be used. It is also possible to implement an operation simply to change the `RecAddrs` member.

https://hemanthrajhemu.github.io

7.4.2 Class TextIndexedFile

Class `TextIndexedFile` is defined in Fig. 7.7 and in file `indfile.h` in Appendix G. It supports files of data objects with primary keys that are strings. As expected, there are methods: `Create`, `Open`, `Close`, `Read` (sequential and indexed), `Append`, and `Update`. In order to ensure the correlation between the index and the data file, the members that represent the index in memory (`Index`), the index file (`IndexFile`), and the data file (`DataFile`) are protected members. The only access to `these members` for the user is through the methods. `TextIndexedFile` is a template class so that data objects of arbitrary classes can be used.

```
template <class RecType>
class TextIndexedFile
{public:
   int Read (RecType & record); // read next record
   int Read (char * key, RecType & record); // read by key
   int Append (const RecType & record);
   int Update (char * oldKey, const RecType & record);
   int Create (char * name, int mode=ios::in|ios::out);
   int Open (char * name, int mode=ios::in|ios::out);
   int Close ();
   TextIndexedFile (IOBuffer & buffer,
      int keySize, int maxKeys = 100);
   ~TextIndexedFile (); // close and delete
protected:
   TextIndex Index;
   BufferFile IndexFile;
   TextIndexBuffer IndexBuffer;
   RecordFile<RecType> DataFile;
   char * FileName; // base file name for file
   int SetFileName(char * fileName,
      char *& dataFileName, char *& indexFileName);
};
// The template parameter RecType must have the following method
//    char * Key()
```

Figure 7.7 Class TextIndexedFile

https://hemanthrajhemu.github.io

As an example, consider `TextIndexedFile::Append`:

```
template <class RecType>
int TextIndexedFile<RecType>::Append (const RecType &
record)
{
   char * key = record.Key();
   int ref = Index.Search(key);
   if (ref != -1) // key already in file
      return -1;
   ref = DataFile . Append(record);
   int result = Index . Insert (key, ref);
   return ref;
}
```

The `Key` method is used to extract the key value from the record. A search of the index is used to determine if the key is already in the file. If not, the record is appended to the data file, and the resulting address is inserted into the index along with the key.

7.4.3 Enhancements to Class TextIndexedFile

Other Types of Keys

Even though class `TextIndexedFile` is parameterized to support a variety of data object classes, it restricts the key type to string (`char *`). It is not hard to produce a template class `SimpleIndex` with a parameter for the key type. Often, changing a class to a template class requires adding a template parameter and then simply replacing a class name with the parameter name—in this case, replacing `char *` by `keytype`. However, the peculiar way that strings are implemented in C and C++ makes this impossible. Any array in C and C++ is represented by a pointer, and equality and assignment operators are defined accordingly. Since a string is an array, string assignment is merely pointer assignment. If you review the methods of class `TextIndex`, you will see that `strcmp` is used to test for key equality, and `strcpy` is used for key assignment. In order to produce a template index class, the dependencies on `char *` must be removed. The template class `SimpleIndex` is included in files `simpind.h` and `simpind.tc` in Appendix G. It is used as the basis for the advanced indexing strategies of Chapter 9.

In C++, assignment and other operators can be overloaded only for class objects, not for predefined types like `int` and `char *`. In order to

https://hemanthrajhemu.github.io

use a template index class for string keys, a class String is needed. Files strclass.h and strclass.cpp of Appendix G have the definition and implementation of this class, which was first mentioned in Chapter 1. Included in this class are a copy constructor, a constructor with a char * parameter, overloaded assignment and comparison operators, and a conversion operator to char * (operator char*). The following code shows how String objects and C strings become interchangeable:

```
String strObj(10); char * strArray[11]; // strings of <=10 chars
strObj = strArray; // uses String::String(char *)
strArray = strObj; // uses String::operator char * ();
```

The first assignment is implemented by constructing a temporary String object using the char * constructor and then doing String assignment. In this way the constructor acts like a conversion operator to class String. The second assignment uses the conversion operator from class String to convert the String object to a simple C string.

Data Object Class Hierarchies

So far, we have required that every object stored in a RecordFile must be of the same type. Can the I/O classes support objects that are of a variety of types but all from the same type hierarchy? If the type hierarchy supports virtual pack methods, the Append and Update will correctly add records to indexed files. That is, if BaseClass supports Pack, Unpack, and Key, the class TextIndexedFile<BaseClass> will correctly output objects derived from BaseClass, each with its appropriate Pack method.

What about Read? The problem here is that in a virtual function call, it is the type of the calling object that determines which method to call. For example, in this code it is the type of the object referenced by Obj (*Obj) that determines which Pack and Unpack are called:

```
BaseClass * Obj = new Subclass1;
Obj->Pack(Buffer); Obj->Unpack(Buffer); // virtual function calls
```

In the case of the Pack, this is correct. Information from *Obj, of type Subclass1, is transferred to Buffer. However, in the case of Unpack, it is a transfer of information from Buffer to *Obj. If Buffer has been filled from an object of class Subclass2 or BaseClass, the unpacking cannot be done correctly. In essence, it is the source of information (contents of the buffer) that determines the type of

https://hemanthrajhemu.github.io

the object in the `Unpack`, not the memory object. The virtual function call does not work in this case. An object from a file can be read only into a memory object of the correct type.

A reliable solution to the read problem—that is, one that does not attempt to read a record into an object of the wrong type—is not easy to implement in C++. It is not difficult to add a type identifier to each data record. We can add record headers in much the same fashion as file headers. However, the read operation must be able to determine reliably the type of the target object. There is no support in C++ for guaranteeing accurate type identification of memory objects.

Multirecord Index Files

Class `TextIndexedFile` requires that the entire index fit in a single record. The maximum number of records in the file is fixed when the file is created. This is obviously an oversimplification of the index structure and a restriction on its utility. Is it worth the effort to extend the class so that this restriction is eliminated?

It would be easy to modify class `TextIndexedFile` to allow the index to be an array of `TextIndex` objects. We could add protected methods `Insert`, `Delete`, and `Search` to manipulate the arrays of index objects. None of this is much trouble. However, as we will see in the following section and in Chapter 9, a sorted array of index objects, each with keys less than the next, does not provide a very satisfactory index for large files. For files that are restricted to a small number of records, class `TextIndexedFile` will work quite well as it is.

Optimization of Operations

The most obvious optimization is to use binary search in the `Find` method, which is used by `Search`, `Insert`, and `Remove`. This is very reasonable and is left as an exercise.

Another source of some improvement is to avoid writing the index record back to the index file when it has not been changed. The standard way to do this is to add a flag to the index object to signal when it has been changed. This flag is set to *false* when the record is initially loaded into memory and set to *true* whenever the index record is modified, that is, by the `Insert` and `Remove` methods. The `Close` method can check this flag and write the record only when necessary. This optimization gains importance when manipulating multirecord index files.

https://hemanthrajhemu.github.io

7.5 Indexes That Are Too Large to Hold in Memory

The methods we have been discussing—and, unfortunately, many of the advantages associated with them—are tied to the assumption that the index is small enough to be loaded into memory in its entirety. If the index is too large for this approach to be practical, then index access and maintenance must be done on secondary storage. With simple indexes of the kind we have been discussing, accessing the index on a disk has the following disadvantages:

- Binary searching of the index requires several seeks instead of taking place at memory speeds. Binary searching of an index on secondary storage is not substantially faster than the binary searching of a sorted file.
- Index rearrangement due to record addition or deletion requires shifting or sorting records on secondary storage. This is literally millions of times more expensive than performing these same operations in memory.

Although these problems are no worse than those associated with any file that is sorted by key, they are severe enough to warrant the consideration of alternatives. Any time a simple index is too large to hold in memory, you should consider using

- A *hashed* organization if access speed is a top priority; or
- A *tree-structured*, or multilevel, index, such as a *B-tree*, if you need the flexibility of both keyed access and ordered, sequential access.

These alternative file organizations are discussed at length in the chapters that follow. But, before writing off the use of simple indexes on secondary storage altogether, we should note that they provide some important advantages over the use of a data file sorted by key even if the index cannot be held in memory:

- A simple index makes it possible to use a binary search to obtain keyed access to a record in a variable-length record file. The index provides the service of associating a fixed-length and therefore binary-searchable record with each variable-length data record.
- If the index entries are substantially smaller than the data file records, sorting and maintaining the index can be less expensive than sorting and maintaining the data file. There is simply less information to move around in the index file.

https://hemanthrajhemu.github.io

- If there are pinned records in the data file, the use of an index lets us rearrange the keys without moving the data records.

There is another advantage associated with the use of simple indexes, one that we have not yet discussed. By itself, it can be reason enough to use simple indexes even if they do not fit into memory. Remember the analogy between an index and a library card catalog? The card catalog provides multiple views or arrangements of the library's collection, even though there is only one set of books arranged in a single order. Similarly, we can use multiple indexes to provide multiple views of a data file.

7.6 Indexing to Provide Access by Multiple Keys

One question that might reasonably arise at this point is: All this indexing business is pretty interesting, but who would ever want to find a recording using a key such as DG18807? What I want is a recording of Beethoven's Symphony No. 9.

Let's return to our analogy of our index as a library card catalog. Suppose we think of our primary key, the Label ID, as a kind of catalog number. Like the catalog number assigned to a book, we have taken care to make our Label ID unique. Now in a library it is very unusual to begin by looking for a book with a particular catalog number (for example, "I am looking for a book with a catalog number QA331T5 1959."). Instead, one generally begins by looking for a book on a particular subject, with a particular title, or by a particular author (for example, "I am looking for a book on functions," or "I am looking for *The Theory of Functions* by Titchmarsh."). Given the subject, author, or title, one looks in the card catalog to find the *primary key*, the catalog number.

Similarly, we could build a catalog for our record collection consisting of entries for album title, composer, and artist. These fields are *secondary key fields*. Just as the library catalog relates an author entry (secondary key) to a card catalog number (primary key), so can we build an index file that relates Composer to Label ID, as illustrated in Fig. 7.8.

Along with the similarities, there is an important difference between this kind of secondary key index and the card catalog in a library. In a library, once you have the catalog number you can usually go directly to the stacks to find the book since the books are arranged in order by catalog number. In other words, the books are sorted by primary key. The actual data records in our file, on the other hand, are *entry sequenced*.

https://hemanthrajhemu.github.io

Composer index

Secondary key	*Primary key*
BEETHOVEN	ANG3795
BEETHOVEN	DG139201
BEETHOVEN	DG18807
BEETHOVEN	RCA2626
COREA	WAR23699
DVORAK	COL31809
PROKOFIEV	LON2312
RIMSKY-KORSAKOV	MER75016
SPRINGSTEEN	COL38358
SWEET HONEY IN THE R	FF245

Figure 7.8
Secondary key index organized by composer.

Consequently, after consulting the composer index to find the Label ID, you must consult one additional index, our primary key index, to find the actual byte offset of the record that has this particular Label ID. Figure 7.9 shows part of the class definition for a secondary key index and a read function that searches a secondary key index for the primary key. It then uses the primary key to read an `IndexedFile`.

Clearly it is possible to relate secondary key references (for example, Beethoven) directly to a byte offset (241) rather than to a primary key (DG18807). However, there are excellent reasons for postponing this binding of a secondary key to a specific address for as long as possible. These reasons become clear as we discuss the way that fundamental file operations such as record deletion and updating are affected by the use of secondary indexes.

Record Addition

When a secondary index is present, adding a record to the file means adding an entry to the secondary index. The cost of doing this is very simi-

https://hemanthrajhemu.github.io

```
class SecondaryIndex
// An index in which the record reference is a string
{public:
   int Insert (char * secondaryKey, char * primaryKey);
   char * Search (char * secondaryKey); // return primary key
      ...
};
template <class RecType>
int SearchOnSecondary (char * composer, SecondaryIndex index,
      IndexedFile<RecType> dataFile, RecType & rec)
{
   char * Key = index.Search (composer);
   // use primary key index to read file
   return dataFile . Read (Key, rec);
}
```

Figure 7.9 SearchOnSecondary: an algorithm to retrieve a single record from a recording file through a secondary key index.

lar to the cost of adding an entry to the primary index: either records must be shifted, or a vector of pointers to structures needs to be rearranged. As with primary indexes, the cost of doing this decreases greatly if the secondary index can be read into memory and changed there.

Note that the key field in the secondary index file is stored in canonical form (all of the composers' names are capitalized), since this is the form we want to use when we are consulting the secondary index. If we want to print out the name in normal, mixed upper- and lowercase form, we can pick up that form from the original data file. Also note that the secondary keys are held to a fixed length, which means that sometimes they are truncated. The definition of the canonical form should take this length restriction into account if searching the index is to work properly.

One important difference between a secondary index and a primary index is that a secondary index can contain duplicate keys. In the sample index illustrated in Fig. 7.10, there are four records with the key BEETHOVEN. Duplicate keys are, of course, grouped together. Within this group, they should be ordered according to the values of the reference fields. In this example, that means placing them in order by Label ID. The reasons for this second level of ordering become clear a little later, as we discuss retrieval based on combinations of two or more secondary keys.

https://hemanthrajhemu.github.io

Title index

Secondary key	Primary key
COQ D'OR SUITE	MER75016
GOOD NEWS	FF245
NEBRASKA	COL38358
QUARTET IN C SHARP M	RCA2626
ROMEO AND JULIET	LON2312
SYMPHONY NO. 9	ANG3795
SYMPHONY NO. 9	COL31809
SYMPHONY NO. 9	DG18807
TOUCHSTONE	WAR23699
VIOLIN CONCERTO	DG139201

Figure 7.10 Secondary key index organized by recording title.

Record Deletion

Deleting a record usually implies removing all references to that record in the file system. So removing a record from the data file would mean removing not only the corresponding entry in the primary index but also all of the entries in the secondary indexes that refer to this primary index entry. The problem with this is that secondary indexes, like the primary index, are maintained in sorted order by key. Consequently, deleting an entry would involve rearranging the remaining entries to close up the space left open by deletion.

This delete-all-references approach would indeed be advisable if the secondary index referenced the data file directly. If we did not delete the secondary key references and if the secondary keys were associated with actual byte offsets in the data file, it could be difficult to tell when these references were no longer valid. This is another instance of the pinned-record problem. The reference fields associated with the secondary keys would be pointing to byte offsets that could, after deletion and subsequent space reuse in the data file, be associated with different data records.

https://hemanthrajhemu.github.io

But we have carefully avoided referencing actual addresses in the secondary key index. After we search to find the secondary key, we do another search, this time on primary key. Since the primary index *does* reflect changes due to record deletion, a search for the primary key of a record that has been deleted will fail, returning a record-not-found condition. In a sense, the updated primary key index acts as a kind of final check, protecting us from trying to retrieve records that no longer exist.

Consequently, one option that is open to us when we delete a record from the data file is to modify and rearrange only the primary key index. We could safely leave intact the references to the deleted record that exist in the secondary key indexes. Searches starting from a secondary key index that lead to a deleted record are caught when we consult the primary key index.

If there are a number of secondary key indexes, the savings that results from not having to rearrange all of these indexes when a record is deleted can be substantial. This is especially important when the secondary key indexes are kept on secondary storage. It is also important with an interactive system in which the user is waiting at a terminal for the deletion operation to complete.

There is, of course, a cost associated with this shortcut: deleted records take up space in the secondary index files. In a file system that undergoes few deletions, this is not usually a problem. In a somewhat more volatile file structure, it is possible to address the problem by periodically removing from the secondary index files all entries that contain references that are no longer in the primary index. If a file system is so volatile that even periodic purging is not adequate, it is probably time to consider another index structure, such as a B-tree, that allows for deletion without having to rearrange a lot of records.

Record Updating

In our discussion of record deletion, we find that the primary key index serves as a kind of protective buffer, insulating the secondary indexes from changes in the data file. This insulation extends to record updating as well. If our secondary indexes contain references directly to byte offsets in the data file, then updates to the data file that result in changing a record's physical location in the file also require updating the secondary indexes. But, since we are confining such detailed information to the primary index, data file updates affect the secondary index only when they change either the primary or the secondary key. There are three possible situations:

https://hemanthrajhemu.github.io

- *Update changes the secondary key:* if the secondary key is changed, we may have to rearrange the secondary key index so it stays in sorted order. This can be a relatively expensive operation.
- *Update changes the primary key:* this kind of change has a large impact on the primary key index but often requires that we update only the affected reference field (*Label ID* in our example) in all the secondary indexes. This involves searching the secondary indexes (on the unchanged secondary keys) and rewriting the affected fixed-length field. It does not require reordering of the secondary indexes unless the corresponding secondary key occurs more than once in the index. If a secondary key does occur more than once, there may be some local reordering, since records having the same secondary key are ordered by the reference field (primary key).
- *Update confined to other fields:* all updates that do not affect either the primary or secondary key fields do not affect the secondary key index, even if the update is substantial. Note that if there are several secondary key indexes associated with a file, updates to records often affect only a subset of the secondary indexes.

7.7 Retrieval Using Combinations of Secondary Keys

One of the most important applications of secondary keys involves using two or more of them in combination to retrieve special subsets of records from the data file. To provide an example of how this can be done, we will extract another secondary key index from our file of recordings. This one uses the recording's *title* as the key, as illustrated in Fig. 7.10. Now we can respond to requests such as

- Find the recording with Label ID COL38358 (primary key access);
- Find all the recordings of Beethoven's work (secondary keyñcomposer); and
- Find all the recordings titled "Violin Concerto" (secondary keyñtitle).

What is more interesting, however, is that we can also respond to a request that *combines* retrieval on the composer index with retrieval on the title index, such as: Find all recordings of Beethoven's Symphony No. 9. Without the use of secondary indexes, this kind of request requires a sequential search through the entire file. Given a file containing thousands,

https://hemanthrajhemu.github.io

or even hundreds, of records, this is a very expensive process. But, with the aid of secondary indexes, responding to this request is simple and quick.

We begin by recognizing that this request can be rephrased as a Boolean *and* operation, specifying the intersection of two subsets of the data file:

```
Find all data records with:
   composer = 'BEETHOVEN' and title = 'SYMPHONY NO. 9'
```

We begin our response to this request by searching the composer index for the list of Label IDs that identify recordings with Beethoven as the composer. This yields the following list of Label IDs:

```
ANG3795
DG139201
DG18807
RCA2626
```

Next we search the title index for the Label IDs associated with records that have SYMPHONY NO. 9 as the title key:

```
ANG3795
COL31809
DG18807
```

Now we perform the Boolean *and*, which is a match operation, combining the lists so only the members that appear in *both* lists are placed in the output list.

```
Composers           Titles              Matched list
ANG3795 ----------- ANG3795 ----------- ANG3795
DG139201            COL31809      +---- DG18807
DG18807 ----------- DG18807 ------+
RCA2626
```

We give careful attention to algorithms for performing this kind of match operation in Chapter 8. Note that this kind of matching is much easier if the lists that are being combined are in sorted order. That is the reason that, when we have more than one entry for a given secondary key, the records are ordered by the primary key reference fields.

Finally, once we have the list of primary keys occurring in both lists, we can proceed to the primary key index to look up the addresses of the data file records. Then we can retrieve the records:

```
ANG | 3795 | Symphony No. 9 | Beethoven | Guilini
DG | 18807 | Symphony No. 9 | Beethoven | Karajan
```

https://hemanthrajhemu.github.io

This is the kind of operation that makes computer-indexed file systems useful in a way that far exceeds the capabilities of manual systems. We have only one copy of each data file record, and yet, working through the secondary indexes, we have multiple views of these records: we can look at them in order by title, by composer, or by any other field that interests us. Using the computer's ability to combine sorted lists rapidly, we can even combine different views, retrieving *intersections* (Beethoven *and* Symphony No. 9) or *unions* (Beethoven *or* Prokofiev *or* Symphony No. 9) of these views. And since our data file is entry sequenced, we can do all of this without having to sort data file records and can confine our sorting to the smaller index records that can often be held in memory.

Now that we have a general idea of the design and uses of secondary indexes, we can look at ways to improve these indexes so they take less space and require less sorting.

7.8 Improving the Secondary Index Structure: Inverted Lists

The secondary index structures that we have developed so far result in two distinct difficulties:

- We have to rearrange the index file *every time* a new record is added to the file, even if the new record is for an existing secondary key. For example, if we add another recording of Beethoven's Symphony No. 9 to our collection, both the composer and title indexes would have to be rearranged, even though both indexes already contain entries for secondary keys (but not the Label IDs) that are being added.
- If there are duplicate secondary keys, the secondary key field is repeated for each entry. This wastes space because it makes the files larger than necessary. Larger index files are less likely to fit in memory.

7.8.1 A First Attempt at a Solution

One simple response to these difficulties is to change the secondary index structure so it associates an *array* of references with each secondary key.

https://hemanthrajhemu.github.io

For example, we might use a record structure that allows us to associate up to four Label ID reference fields with a single secondary key, as in

BEETHOVEN	ANG3795	DG139201	DG18807	RCA2626

Figure 7.11 provides a schematic example of how such an index would look if used with our sample data file.

The major contribution of this revised index structure is its help in solving our first difficulty: the need to rearrange the secondary index file every time a new record is added to the data file. Looking at Fig. 7.11, we can see that the addition of another recording of a work by Prokofiev does not require the addition of another record to the index. For example, if we add the recording

```
ANG  36193 Piano Concertos 3 and 5  Prokofiev Francois
```

we need to modify only the corresponding secondary index record by inserting a second Label ID:

PROKOFIEV	ANG36193	LON2312		

Since we are not adding another record to the secondary index, there is no need to rearrange any records. All that is required is a rearrangement of the fields in the existing record for Prokofiev.

Although this new structure helps avoid the need to rearrange the secondary index file so often, it does have some problems. For one thing, it provides space for only four Label IDs to be associated with a given key. In the very likely case that more than four Label IDs will go with some key, we need a mechanism for keeping track of the extra Label IDs.

A second problem has to do with space usage. Although the structure does help avoid the waste of space due to the repetition of identical keys, this space savings comes at a potentially high cost. By extending the fixed length of each of the secondary index records to hold more reference fields, we might easily lose more space to internal fragmentation than we gained by not repeating identical keys.

Since we don't want to waste any more space than we have to, we need to ask whether we can improve on this record structure. Ideally, what we would like to do is develop a new design, a revision of our revision, that

https://hemanthrajhemu.github.io

Revised composer index

Secondary key	*Set of primary key references*			
BEETHOVEN	ANG3795	DG139201	DG18807	RCA2626
COREA	WAR23699			
DVORAK	COL31809			
PROKOFIEV	LON2312			
RIMSKY-KORSAKOV	MER75016			
SPRINGSTEEN	COL38358			
SWEET HONEY IN THE R	FF245			

Figure 7.11 Secondary key index containing space for multiple references for each secondary key.

- Retains the attractive feature of not requiring reorganization of the secondary indexes for every new entry to the data file;
- Allows more than four Label IDs to be associated with each secondary key; and
- Eliminates the waste of space due to internal fragmentation.

7.8.2 A Better Solution: Linking the List of References

Files such as our secondary indexes, in which a secondary key leads to a set of one or more primary keys, are called *inverted lists.* The sense in which the list is inverted should be clear if you consider that we are working our way backward from a secondary key to the primary key to the record itself.

The second word in the term "inverted list" also tells us something important: we are, in fact, dealing with a *list* of primary key references. Our revised secondary index, which collects a number of Label IDs for each secondary key, reflects this list aspect of the data more directly than our initial secondary index. Another way of conceiving of this list aspect of our inverted list is illustrated in Fig. 7.12.

As Fig. 7.12 shows, an ideal situation would be to have each secondary key point to a different list of primary key references. Each of these lists

https://hemanthrajhemu.github.io

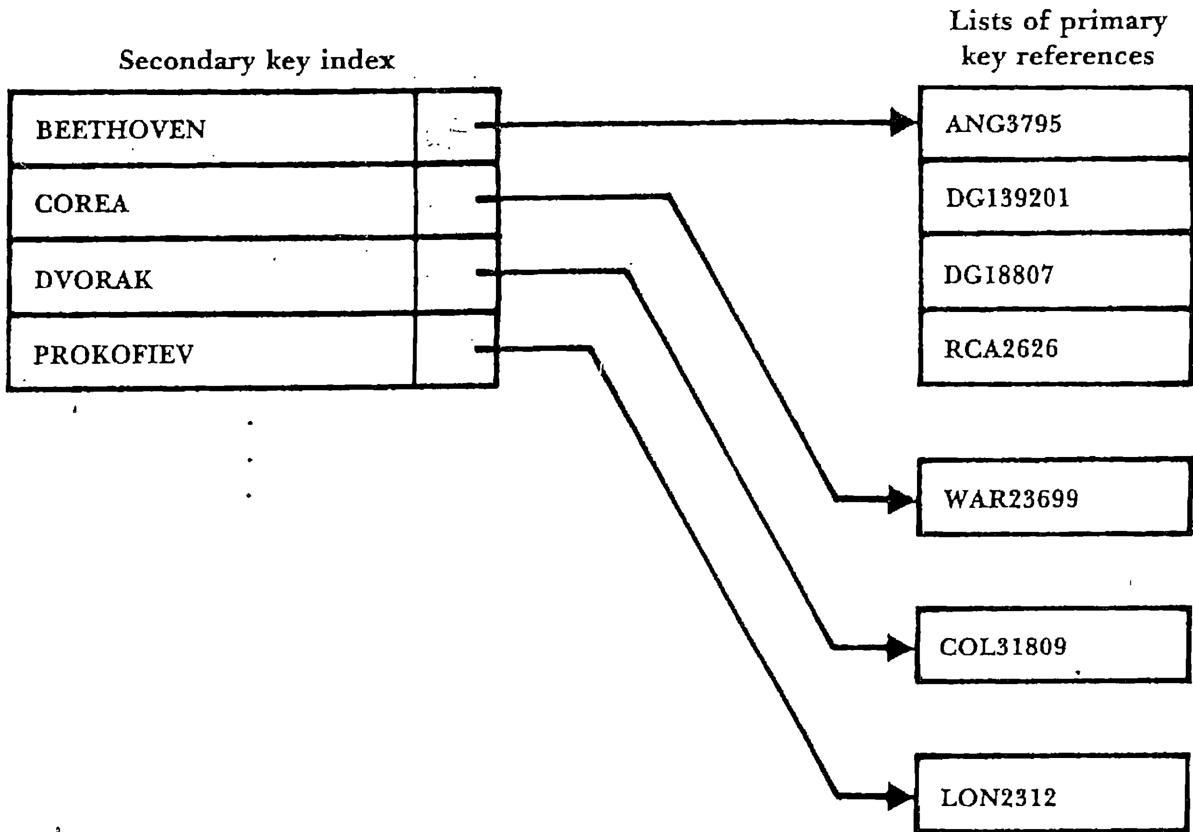
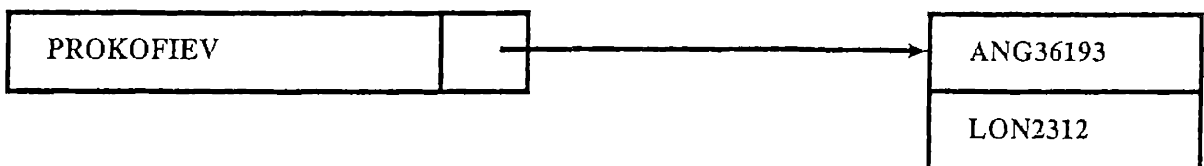


Figure 7.12 Conceptual view of the primary key reference fields as a series of lists.

could grow to be just as long as it needs to be. If we add the new Prokofiev record, the list of Prokofiev references becomes



Similarly, adding two new Beethoven recordings adds just two additional elements to the list of references associated with the Beethoven key. Unlike our record structure which allocates enough space for four Label IDs for each secondary key, the lists could contain hundreds of references, if needed, while still requiring only one instance of a secondary key. On the other hand, if a list requires only one element, then no space is lost to internal fragmentation. Most important, we need to rearrange only the file of secondary keys if a new composer is added to the file.

https://hemanthrajhemu.github.io

How can we set up an unbounded number of different lists, each of varying length, without creating a large number of small files? The simplest way is through the use of linked lists. We could redefine our secondary index so it consists of records with two fields—a secondary key field and a field containing the relative record number of the first corresponding primary key reference (Label ID) in the inverted list. The actual primary key references associated with each secondary key would be stored in a separate, entry-sequenced file.

Given the sample data we have been working with, this new design would result in a secondary key file for composers and an associated Label ID file that are organized as illustrated in Fig. 7.13. Following the links for the list of references associated with Beethoven helps us see how the Label ID List file is organized. We begin, of course, by searching the secondary key index of composers for Beethoven. The record that we find points us to relative record number (RRN) 3 in the Label ID List file. Since this is a fixed-length file, it is easy to jump to RRN 3 and read in its Label ID

Improved revision of the composer index

Secondary Index file

0	BEETHOVEN	3
1	COREA	2
2	DVORAK	7
3	PROKOFIEV	10
4	RIMSKY-KORSAKOV	6
5	SPRINGSTEEN	4
6	SWEET HONEY IN THE R	9

Label ID List file

0	LON2312	−1
1	RCA2626	−1
2	WAR23699	−1
3	ANG3795	8
4	COL38358	−1
5	DG18807	1
6	MER75016	−1
7	COL31809	−1
8	DG139201	5
9	FF245	−1
10	ANG36193	0

Figure 7.13 Secondary key index referencing linked lists of primary key references.

https://hemanthrajhemu.github.io

(ANG3795). Associated with this Label ID is a link to a record with RRN 8. We read in the Label ID for that record, adding it to our list (ANG379 DG139201). We continue following links and collecting Label IDs until the list looks like this:

```
ANG3795    DG139201    DG18807    RCA2626
```

The link field in the last record read from the Label ID List file contains a value of -1. As in our earlier programs, this indicates end-of-list, so we know that we now have all the Label ID references for Beethoven records.

To illustrate how record addition affects the Secondary Index and Label ID List files, we add the Prokofiev recording mentioned earlier:

```
ANG  36193 Piano Concertos 3 and 5  Prokofiev Francois
```

You can see (Fig. 7.13) that the Label ID for this new recording is the last one in the Label ID List file, since this file is entry sequenced. Before this record is added, there is only one Prokofiev recording. It has a Label ID of LON2312. Since we want to keep the Label ID Lists in order by ASCII character values, the new recording is inserted in the list for Prokofiev so it logically precedes the LON2312 recording.

Associating the Secondary Index file with a new file containing linked lists of references provides some advantages over any of the structures considered up to this point:

- The only time we need to rearrange the Secondary Index file is when a new composer's name is added or an existing composer's name is changed (for example, it was misspelled on input). Deleting or adding recordings for a composer who is already in the index involves changing only the Label ID List file. Deleting *all* the recordings for a composer could be handled by modifying the Label ID List file while leaving the entry in the Secondary Index file in place, using a value of -1 in its reference field to indicate that the list of entries for this composer is empty.
- In the event that we need to rearrange the Secondary Index file, the task is quicker now since there are fewer records and each record is smaller.
- Because there is less need for sorting, it follows that there is less of a penalty associated with keeping the Secondary Index files off on secondary storage, leaving more room in memory for other data structures.

https://hemanthrajhemu.github.io

- The Label ID List file is entry sequenced. That means that it *never* needs to be sorted.
- Since the Label ID List file is a fixed-length record file, it would be very easy to implement a mechanism for reusing the space from deleted records, as described in Chapter 6.

There is also at least one potentially significant disadvantage to this kind of file organization: the Label IDs associated with a given composer are no longer guaranteed to be grouped together physically. The technical term for such "togetherness" is *locality*. With a linked, entry-sequenced structure such as this, it is less likely that there will be locality associated with the logical groupings of reference fields for a given secondary key. Note, for example, that our list of Label IDs for Prokofiev consists of the very last and the very first records in the file. This lack of locality means that picking up the references for a composer with a long list of references could involve a large amount of *seeking* back and forth on the disk. Note that this kind of seeking would not be required for our original Secondary Index file structure.

One obvious antidote to this seeking problem is to keep the Label ID List file in memory. This could be expensive and impractical, given many secondary indexes, except for the interesting possibility of using the same Label ID List file to hold the lists for a number of Secondary Index files. Even if the file of reference lists were too large to hold in memory, it might be possible to obtain a performance improvement by holding only a part of the file in memory at a time, paging sections of the file in and out of memory as they are needed.

Several exercises at the end of the chapter explore these possibilities more thoroughly. These are very important problems, as the notion of dividing the index into pages is fundamental to the design of B-trees and other methods for handling large indexes on secondary storage.

7.9 Selective Indexes

Another interesting feature of secondary indexes is that they can be used to divide a file into parts and provide a selective view. For example, it is possible to build a *selective index* that contains only the titles of classical recordings in the record collection. If we have additional information about the recordings in the data file, such as the date the recording was released, we could build selective indexes such as "recordings released prior

https://hemanthrajhemu.github.io

to 1970" and "recordings since 1970." Such selective index information could be combined into Boolean *and* operations to respond to requests such as "List all the recordings of Beethoven's Ninth Symphony released since 1970." Selective indexes are sometimes useful when the contents of a file fall naturally and logically into several broad categories.

7.10 Binding

A recurrent and very important question that emerges in the design of file systems that use indexes is: *At what point is the key bound to the physical address of its associated record?*

In the file system we are designing in the course of this chapter, the *binding* of our primary keys to an address takes place *at the time the files are constructed.* The secondary keys, on the other hand, are bound to an address *at the time that they are used.*

Binding at the time of the file construction results in faster access. Once you have found the right index record, you have in hand the byte offset of the data record you are seeking. If we elected to bind our secondary keys to their associated records at the time of file construction so when we find the DVORAK record in the composer index we would know immediately that the data record begins at byte 338 in the data file, secondary key retrieval would be simpler and faster. The improvement in performance is particularly noticeable if both the primary and secondary index files are used on secondary storage rather than in memory. Given the arrangement we designed, we would have to perform a binary search of the composer index and then a binary search of the primary key index before being able to jump to the data record. Binding early, at file construction time, eliminates the need to search on the primary key.

The disadvantage of binding directly in the file, of *binding tightly,* is that reorganizations of the data file must result in modifications to all bound index files. This reorganization cost can be very expensive, particularly with simple index files in which modification would often mean shifting records. By postponing binding until execution time, when the records are being used, we are able to develop a secondary key system that involves a minimal amount of reorganization when records are added or deleted.

Another important advantage to postponing binding until a record is retrieved is that this approach is safer. As we see in the system that we set

https://hemanthrajhemu.github.io

up, associating the secondary keys with reference fields consisting of primary keys allows the primary key index to act as a kind of final check of whether a record is really in the file. The secondary indexes can afford to be wrong. This situation is very different if the secondary index keys contain addresses. We would then be jumping directly from the secondary key into the data file; the address would need to be right.

This brings up a related safety aspect: it is always more desirable to make important changes in one place rather than in many places. With a bind-at-retrieval-time scheme such as we developed, we need to remember to make a change in only one place, the primary key index, if we move a data record. With a more tightly bound system, we have to make many changes successfully to keep the system internally consistent, braving power failures, user interruptions, and so on.

When designing a new file system, it is better to deal with this question of binding *intentionally* and *early* in the design process rather than letting the binding just happen. In general, tight, in-the-data binding is most attractive when

- The data file is static or nearly so, requiring little or no adding, deleting, or updating of records; and
- Rapid performance during actual retrieval is a high priority.

For example, tight binding is desirable for file organization on a mass-produced, read-only optical disk. The addresses will never change because no new records can ever be added; consequently, there is no reason not to obtain the extra performance associated with tight binding.

For file applications in which record addition, deletion, and updating do occur, however, binding at retrieval time is usually the more desirable option. Postponing binding as long as possible usually makes these operations simpler and safer. If the file structures are carefully designed, and, in particular, if the indexes use more sophisticated organizations such as B-trees, retrieval performance is usually quite acceptable, even given the additional work required by a bind-at-retrieval system.

SUMMARY

We began this chapter with the assertion that indexing as a way of structuring a file is an alternative to sorting because records can be found *by key*. Unlike sorting, indexing permits us to perform *binary searches* for keys in variable-length record files. If the index can be held in memory, record

https://hemanthrajhemu.github.io

addition, deletion, and retrieval can be done much more quickly with an indexed, entry-sequenced file than with a sorted file.

Template classes in C++ provide support for sharing class definitions and code among a number of unrelated classes. Template classes are used in this chapter for class `RecordFile`, which supports I/O of data records without explicit packing and unpacking of buffers, and for general purpose index records in class `SimpleIndex`.

Support for sequential and indexed access to a data file is provided by the template class `TextIndexedFile`. It extends the capabilities of class `RecordFile` by adding indexed read, update, and append operations. Each modification of the data file is accompanied by the proper changes to the index. Each `TextIndexedFile` object is represented by an index record object in memory and two files, a data file and an index file. The `TextIndexedFile::Close` method writes the contents of the index record object into the index file and closes both files.

Indexes can do much more than merely improve on access time: they can provide us with new capabilities that are inconceivable with access methods based on sorted data records. The most exciting new capability involves the use of multiple secondary indexes. Just as a library card catalog allows us to regard a collection of books in author order, title order, or subject order, so index files allow us to maintain different views of the records in a data file. We find that not only can we use secondary indexes to obtain different views of the file but we can also combine the associated lists of primary key references and thereby combine particular views.

In this chapter we address the problem of how to rid our secondary indexes of two liabilities:

- The need to repeat duplicate secondary keys, and
- The need to rearrange the secondary indexes every time a record is added to the data file.

A first solution to these problems involves associating a fixed-size *vector* of reference fields with each secondary key. This solution results in an overly large amount of internal fragmentation but illustrates the attractiveness of handling the reference fields associated with a particular secondary key as a group, or *list*.

Our next iteration of solutions to our secondary index problems is more successful and much more interesting. We can treat the primary key references as an entry-sequenced file, forming the necessary lists through the use of *link fields* associated with each primary record entry. This allows us to create a secondary index file that, in the case of the composer index, needs rearrangement only when we add new composers to the data file.

https://hemanthrajhemu.github.io

The entry-sequenced file of linked reference lists never requires sorting. We call this kind of secondary index structure an *inverted list.*

There are also, of course, disadvantages associated with our new solution. The most serious disadvantage is that our file demonstrates less locality: lists of associated records are less likely to be physically adjacent. A good antidote to this problem is to hold the file of linked lists in memory. We note that this is made more plausible because a single file of primary references can link the lists for a number of secondary indexes.

As indicated by the length and breadth of our consideration of secondary indexing, multiple keys, and inverted lists, these topics are among the most interesting aspects of indexed access to files. The concepts of secondary indexes and inverted lists become even more powerful later, as we develop index structures that are themselves more powerful than the simple indexes we consider here. But, even so, we already see that for small files consisting of no more than a few thousand records, approaches to inverted lists that rely merely on simple indexes can provide a user with a great deal of capability and flexibility.

KEY TERMS

Binding. Binding takes place when a key is associated with a particular physical record in the data file. In general, binding can take place either during the preparation of the data file and indexes or during program execution. In the former case, called *tight binding,* the indexes contain explicit references to the associated physical data record. In the latter case, the connection between a key and a particular physical record is postponed until the record is retrieved in the course of program execution.

Entry-sequenced file. A file in which the records occur in the order that they are entered into the file.

Index. An index is a tool for finding records in a file. It consists of a *key field* on which the index is searched and a *reference field* that tells where to find the data file record associated with a particular key.

Inverted list. The term *inverted list* refers to indexes in which a key may be associated with a *list* of reference fields pointing to documents that contain the key. The secondary indexes developed toward the end of this chapter are examples of inverted lists.

Key field. The key field is the portion of an index record that contains the canonical form of the key that is being sought.

https://hemanthrajhemu.github.io

Locality. Locality exists in a file when records that will be accessed in a given temporal sequence are found in physical proximity to each other on the disk. Increased locality usually results in better performance, as records that are in the same physical area can often be brought into memory with a single *read* request to the disk.

Reference field. The reference field is the portion of an index record that contains information about where to find the data record containing the information listed in the associated key field of the index.

Selective index. A selective index contains keys for only a portion of the records in the data file. Such an index provides the user with a view of a specific subset of the file's records.

Simple index. All the index structures discussed in this chapter are simple indexes insofar as they are all built around the idea of an ordered, linear sequence of index records. All these simple indexes share a common weakness: adding records to the index is expensive. As we see later, tree-structured indexes provide an alternate, more efficient solution to this problem.

Template class. A C++ class that is parameterized, typically with class (or type) parameters. Templates allow a single class definition to be used to construct a family of different classes, each with different arguments for the parameters.

FURTHER READINGS

We have much more to say about indexing in later chapters, where we take up the subjects of tree-structured indexes and indexed sequential file organizations. The topics developed in the current chapter, particularly those relating to secondary indexes and inverted files, are also covered by many other file and data structure texts. The few texts that we list here are of interest because they either develop certain topics in more detail or present the material from a different viewpoint.

Wiederhold (1983) provides a survey of many of the index structures we discuss, along with a number of others. His treatment is more mathematical than that provided in our text. Tremblay and Sorenson (1984) provide a comparison of inverted list structures with an alternative organization called *multilist* files. M. E. S. Loomis (1989) provides a similar discussion, along with some examples oriented toward COBOL users. Kroenke (1998) discuss inverted lists in the context of their application in information retrieval systems.

https://hemanthrajhemu.github.io

EXERCISES

1. Until now, it was not possible to perform a binary search on a variable-length record file. Why does indexing make binary search possible? With a fixed-length record file it *is* possible to perform a binary search. Does this mean that indexing need not be used with fixed-length record files?
2. Why is `Title` not used as a primary key in the `Recording` file described in this chapter? If it were used as a secondary key, what problems would have to be considered in deciding on a canonical form for titles?
3. What is the purpose of keeping an out-of-date-status flag in the header record of an index? In a multiprogramming environment, this flag might be found to be set by one program because another program is in the process of reorganizing the index. How should the first program respond to this situation?
4. Consult a reference book on C++ to determine how template classes like `RecordFile` are implemented. How does the compiler process the method bodies of a template class? How does the compiler process template instantiations?
5. Explain how the use of an index pins the data records in a file.
6. When a record in a data file is updated, corresponding primary and secondary key indexes may or may not have to be altered, depending on whether the file has fixed- or variable-length records, and depending on the type of change made to the data record. Make a list of the different updating situations that can occur, and explain how each affects the indexes.
7. Discuss the problem that occurs when you add the following recording to the recordings file, assuming that the composer index shown in Fig. 7.11 is used. How might you solve the problem without substantially changing the secondary key index structure?

   ```
   LON     1259   Fidelio       Beethoven    Maazel
   ```

8. How are the structures in Fig. 7.13 changed by the addition of the recording

   ```
   LON     1259   Fidelio       Beethoven    Maazel
   ```

9. Suppose you have the data file described in this chapter, but it's greatly expanded, with a primary key index and secondary key indexes

https://hemanthrajhemu.github.io

organized by composer, artist, and title. Suppose that an inverted list structure is used to organize the secondary key indexes. Give step-by-step descriptions of how a program might answer the following queries:

a. List all recordings of Bach or Beethoven, and
b. List all recordings by Perleman of pieces by Mozart or Joplin.

10. Using the program `makerec.cpp`, create a file of recordings. Make a file dump of the file and find the size and contents of the header as well asand the starting address and the size for each record.

11. Use the program `makeind.cpp` to create an index file for the recording file created by program `makerec.cpp`. Using a file dump, find the size and contents of the header, the address and size of the record, and the contents of the record.

12. The method and timing of binding affect two important attributes of a file system—speed and flexibility. Discuss the relevance of these attributes, and the effect of binding time on them, for a hospital patient datainformation system designed to provide information about current patients by patient name, patient ID, location, medication, doctor or doctors, and illness.

PROGRAMMING AND DESIGN EXERCISES

13. Add method(s) to class `TextIndex` to support iterating through the index in key order. One possible strategy is to define two methods:

```
int FirstRecAddr (); // return reference for the smallest key
int NextRecAddr (); // return reference for the next key
```

Implementation of these methods can be supported by adding members to the class.

14. Write a program to print the records of a `Recording` file in key order. One way to implement this program is to read all the records of the file and create an index record in memory and then iterate through the index in key order and read and print the records. Test the program on the file produced by `makerec.cpp`.

15. Write a program to print the records of a file of type `RecordFile<Recording>` in key order. Test the program on the file produced by `makeind.cpp`.

https://hemanthrajhemu.github.io

16. Modify the method `TextIndex::Search` to perform a binary search on the key array.
17. Implement the Remove methods of class `TextIndexedFile`.
18. Extend class `TextIndexedFile` to support the creation of an indexed file from a simple data file. That is, add a method that initializes a `TextIndexedFile` object by opening and reading the existing data file and creating an index from the records in the file.
19. As a major programming project, create a class hierarchy based on `Recording` that has different information for different types of recordings. Develop a class to support input and output of records of these types. The class should be consistent with the style described in the part of Section 7.4.3 about data object class hierarchies. The `Unpack` methods must be sensitive to the type of object that is being initialized by the call.
20. Define and implement a class `SecondaryIndex` to support secondary indexes, as described in Section 7.6. Use this class to create a class `RecordingFile` that uses `RecordFile` as its base class to manage the primary index and the data file and has secondary indexes for the Composer and Artist fields.
21. When searching secondary indexes that contain multiple records for some of the keys, we do not want to find just *any* record for a given secondary key; we want to find the *first* record containing that key. Finding the first record allows us to read ahead, sequentially, extracting all of the records for the given key. Write a variation of a search method that returns the relative record number of the *first* record containing the given key.
22. Identify and eliminate memory leaks in the code of Appendix F.

PROGRAMMING PROJECT

This is the fifth part of the programming project. We add indexes to the data files created by the third part of the project in Chapter 4.

23. Use class `IndexedFile` (or `TextIndexedFile`) to create an index of a file of student objects, using student identifier as key. Write a driver program to create an index file from the student record file created by the program of part three of the programming project in Chapter 4.

https://hemanthrajhemu.github.io

24. Use class `IndexedFile` (or `TextIndexedFile`) to create an index of a file of course registration objects, using student identifier as key. Note that the student identifier is not unique in course registration files. Write a driver program to create an index file from the course registration record file created by the program of part three of the programming project in Chapter 4.
25. Write a program that opens an indexed student file and an indexed course registration file and retrieves information on demand. Prompt a user for a student identifier and print all objects that match it.
26. Develop a class that supports indexed access to course registration files by student identifier and by course identifier (secondary key). See Exercise 20 for an implementation of secondary indexes. Extend the program of Exercise 25 to allow retrieval of information about specific courses.
27. Extend the above projects to support update and deletion of student records and course registration records.

The next part of the programming project is in Chapter 8.

https://hemanthrajhemu.github.io