

FUTURE VISION BIE

One Stop for All Study Materials
& Lab Programs



Future Vision

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>

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

8.5.3	The Cost of Increasing the File Size	326
8.5.4	Hardware-Based Improvements	327
8.5.5	Decreasing the Number of Seeks Using Multiple-Step Merges	329
8.5.6	Increasing Run Lengths Using Replacement Selection	332
8.5.7	Replacement Selection Plus Multistep Merging	338
8.5.8	Using Two Disk Drives with Replacement Selection	341
8.5.9	More Drives? More Processors?	343
8.5.10	Effects of Multiprogramming	344
8.5.11	A Conceptual Toolkit for External Sorting	344
8.6	Sorting Files on Tape	345
8.6.1	The Balanced Merge	346
8.6.2	The K -way Balanced Merge	348
8.6.3	Multiphase Merges	349
8.6.4	Tapes versus Disks for External Sorting	351
8.7	Sort-Merge Packages	352
8.8	Sorting and Cosequential Processing in Unix	352
8.8.1	Sorting and Merging in Unix	352
8.8.2	Cosequential Processing Utilities in Unix	355
	Summary	357
	Key Terms	360
	Further Readings	362
	Exercises	363
	Programming Exercises	366
	Programming Project	367

Chapter 9 Multilevel Indexing and B-Trees

369

9.1	Introduction: The Invention of the B-Tree	370
9.2	Statement of the Problem	372
9.3	Indexing with Binary Search Trees	373
9.3.1	AVL Trees	377
9.3.2	Paged Binary Trees	380
9.3.3	Problems with Paged Trees	382
9.4	Multilevel Indexing, a Better Approach to Tree Indexes	384
9.5	B-trees: Working up from the Bottom	387
9.6	Example of Creating a B-Tree	388
9.7	An Object-Oriented Representation of B-Trees	391
9.7.1	Class BTreeNode: representing B-Tree Nodes in Memory	391
9.7.2	Class BTree: Supporting Files of B-Tree Nodes	393
9.8	B-Tree Methods Search, Insert, and Others	394
9.8.1	Searching	394
9.8.2	Insertion	395
9.8.3	Create, Open, and Close	398
9.8.4	Testing the B-Tree	398
9.9	B-Tree Nomenclature	399
9.10	Formal Definition of B-Tree Properties	401

9.11	Worst-Case Search Depth	401
9.12	Deletion, Merging, and Redistribution	403
9.12.1	Redistribution	406
9.13	Redistribution During Insertion: A Way to Improve Storage Utilization	407
9.14	B* Trees	408
9.15	Buffering of Pages: Virtual B-Trees	409
9.15.1	LRU Replacement	410
9.15.2	Replacement Based on Page Height	411
9.15.3	Importance of Virtual B-Trees	412
9.16	Variable-Length Records and Keys	413
Summary	414	Key Terms 416
		Further Readings 417
		Exercises 419
	Programming Exercises	421
	Programming Project	422

Chapter 10 Indexed Sequential File Access and Prefix B⁺ Trees 423

10.1	Indexed Sequential Access	424
10.2	Maintaining a Sequence Set	425
10.2.1	The Use of Blocks	425
10.2.2	Choice of Block Size	428
10.3	Adding a Simple Index to the Sequence Set	430
10.4	The Content of the Index: Separators Instead of Keys	432
10.5	The Simple Prefix B ⁺ Tree	434
10.6	Simple Prefix B ⁺ Tree Maintenance	435
10.6.1	Changes Localized to Single Blocks in the Sequence Set	435
10.6.2	Changes Involving Multiple Blocks in the Sequence Set	436
10.7	Index Set Block Size	439
10.8	Internal Structure of Index Set Blocks: A Variable-Order B-Tree	440
10.9	Loading a Simple Prefix B ⁺ Tree	443
10.10	B ⁺ Trees	447
10.11	B-Trees, B ⁺ Trees, and Simple Prefix B ⁺ Trees in Perspective	449
Summary	452	Key Terms 455
		Further Readings 456
		Exercises 457
	Programming Exercises	460
	Programming Project	461

Chapter 11 Hashing 463

11.1	Introduction	464
11.1.1	What Is Hashing?	465
11.1.2	Collisions	466
11.2	A Simple Hashing Algorithm	468

Cosequential Processing and the Sorting of Large Files

CHAPTER OBJECTIVES

- ❖ Describe a class of frequently used processing activities known as *cosequential processes*.
- ❖ Provide a general object-oriented model for implementing all varieties of cosequential processes.
- ❖ Illustrate the use of the model to solve a number of different kinds of cosequential processing problems, including problems other than simple merges and matches.
- ❖ Introduce *heapsort* as an approach to overlapping I/O with sorting in memory.
- ❖ Show how merging provides the basis for sorting very large files.
- ❖ Examine the costs of *K*-way merges on disk and find ways to reduce those costs.
- ❖ Introduce the notion of *replacement selection*.
- ❖ Examine some of the fundamental concerns associated with sorting large files using tapes rather than disks.
- ❖ Introduce Unix utilities for sorting, merging, and cosequential processing.

CHAPTER OUTLINE

- 8.1 An Object-Oriented Model for Implementing Cosequential Processes**
 - 8.1.1 Matching Names in Two Lists
 - 8.1.2 Merging Two Lists
 - 8.1.3 Summary of the Cosequential Processing Model
- 8.2 Application of the Model to a General Ledger Program**
 - 8.2.1 The Problem
 - 8.2.2 Application of the Model to the Ledger Program
- 8.3 Extension of the Model to Include Multiway Merging**
 - 8.3.1 A K -way Merge Algorithm
 - 8.3.2 A Selection Tree for Merging Large Numbers of Lists
- 8.4 A Second Look at Sorting in Memory**
 - 8.4.1 Overlapping Processing and I/O: Heapsort
 - 8.4.2 Building the Heap While Reading the File
 - 8.4.3 Sorting While Writing to the File
- 8.5 Merging as a Way of Sorting Large Files on Disk**
 - 8.5.1 How Much Time Does a Merge Sort Take?
 - 8.5.2 Sorting a File That Is Ten Times Larger
 - 8.5.3 The Cost of Increasing the File Size
 - 8.5.4 Hardware-Based Improvements
 - 8.5.5 Decreasing the Number of Seeks Using Multiple-Step Merges
 - 8.5.6 Increasing Run Lengths Using Replacement Selection
 - 8.5.7 Replacement Selection Plus Multistep Merging
 - 8.5.8 Using Two Disk Drives with Replacement Selection
 - 8.5.9 More Drives? More Processors?
 - 8.5.10 Effects of Multiprogramming
 - 8.5.11 A Conceptual Toolkit for External Sorting
- 8.6 Sorting Files on Tape**
 - 8.6.1 The Balanced Merge
 - 8.6.2 The K -way Balanced Merge
 - 8.6.3 Multiphase Merges
 - 8.6.4 Tapes versus Disks for External Sorting
- 8.7 Sort-Merge Packages**
- 8.8 Sorting and Cosequential Processing in Unix**
 - 8.8.1 Sorting and Merging in Unix
 - 8.8.2 Cosequential Processing Utilities in Unix

Cosequential operations involve *the coordinated processing of two or more sequential lists to produce a single output list*. Sometimes the processing results in a *merging*, or *union*, of the items in the input lists; sometimes the goal is a *matching*, or *intersection*, of the items in the lists; and other times the operation is a combination of matching and merging. These kinds of operations on sequential lists are the basis of a great deal of file processing.

In the first half of this chapter we develop a general object-oriented model for performing cosequential operations, illustrate its use for simple matching and merging operations, then apply it to the development of a more complex general ledger program. Next we apply the model to multi-way merging, which is an essential component of external sort-merge operations. We conclude the chapter with a discussion of external sort-merge procedures, strategies, and trade-offs, paying special attention to performance considerations.

8.1 An Object-Oriented Model for Implementing Cosequential Processes

Cosequential operations usually appear to be simple to construct; given the information that we provide in this chapter, this appearance of simplicity can be turned into reality. However, it is also true that approaches to cosequential processing are often confused, poorly organized, and incorrect. These examples of bad practice are by no means limited to student programs: the problems also arise in commercial programs and textbooks. The difficulty with these incorrect programs is usually that they are not organized around a single, clear model for cosequential processing. Instead, they seem to deal with the various exception conditions and problems of a cosequential process in an *ad hoc* rather than systematic way.

This section addresses such lack of organization head on. We present a single, simple model that can be the basis for the construction of any kind of cosequential process. By understanding and adhering to the design principles inherent in the model, you will be able to write cosequential procedures that are simple, short, and robust.

We present this model by defining a class `CosequentialProcess` that supports processing of any type of list, in the same way that class `IOBuffer` supports buffer operations on any type of buffer. Class `CosequentialProcess` includes operations to match and merge lists. It defines the list processing operations required for cosequential processing

as virtual methods. We will then define new subclasses that include the methods for accessing the elements of particular types of lists.

8.1.1 Matching Names in Two Lists

Suppose we want to output the names common to the two lists shown in Fig. 8.1. This operation is usually called a *match operation*, or an *intersection*. We assume, for the moment, that we will not allow duplicate names within a list and that the lists are sorted in ascending order.

We begin by reading in the initial item from each list, and we find that they match. We output this first item as a member of the *match set*, or *intersection set*. We then read in the next item from each list. This time the

List 1	List 2
ADAMS	ADAMS
CARTER	ANDERSON
CHIN	ANDREWS
DAVIS	BECH
FOSTER	BURNS
GARWICK	CARTER
JAMES	DAVIS
JOHNSON	DEMPSEY
KARNS	GRAY
LAMBERT	JAMES
MILLER	JOHNSON
PETERS	KATZ
RESTON	PETERS
ROSEWALD	ROSEWALD
TURNER	SCHMIDT
	THAYER
	WALKER
	WILLIS

Figure 8.1 Sample input lists for cosequential operations.

item in List 2 is less than the item in List 1. When we are processing these lists visually as we are now, we remember that we are trying to match the item CARTER from List 1 and scan down List 2 until we either find it or jump beyond it. In this case, we eventually find a match for CARTER, so we output it, read in the next item from each list, and continue the process. Eventually we come to the end of one of the lists. Since we are looking for items common to both lists, we know we can stop at this point.

Although the match procedure appears to be quite simple, there are a number of matters that have to be dealt with to make it work reasonably well.

- *Initializing*: we need to arrange things in such a way that the procedure gets going properly.
- *Getting and accessing the next list item*: we need simple methods that support getting the next list element and accessing it.
- *Synchronizing*: we have to make sure that the current item from one list is never so far ahead of the current item on the other list that a match will be missed. Sometimes this means getting the next item from List 1, sometimes from List 2, sometimes from both lists.
- *Handling end-of-file conditions*: when we get to the end of either List 1 or List 2, we need to halt the program.
- *Recognizing errors*: when an error occurs in the data (for example, duplicate items or items out of sequence), we want to detect it and take some action.

Finally, we would like our algorithm to be reasonably efficient, simple, and easy to alter to accommodate different kinds of data. The key to accomplishing these objectives in the model we are about to present lies in the way we deal with the third item in our list—synchronizing.

At each step in the processing of the two lists, we can assume that we have two items to compare: a current item from List 1 and a current item from List 2. Let's call these two current items `Item(1)` and `Item(2)`. We can compare the two items to determine whether `Item(1)` is less than, equal to, or greater than `Item(2)`:

- If `Item(1)` is *less than* `Item(2)`, we get the next item from List 1;
- If `Item(1)` is *greater than* `Item(2)`, we get the next item from List 2; and
- If the items are the same, we output the item and get the next items from the two lists.

It turns out that this can be handled very cleanly with a single loop containing one three-way conditional statement, as illustrated in the algorithm of Fig. 8.2. The key feature of this algorithm is that *control always returns to the head of the main loop after every step of the operation*. This means that no extra logic is required within the loop to handle the case when List 1 gets ahead of List 2, or List 2 gets ahead of List 1, or the end-of-file condition is reached on one list before it is on the other. Since each pass through the main loop looks at the next pair of items, the fact that one list may be longer than the other does not require any special logic. Nor does the end-of-file condition—each operation to get a new item resets the `MoreNames` flag that records whether items are available in both lists. The while statement simply checks the value of the flag `MoreNames` on every cycle.

```
int Match (char * List1Name, char * List2Name,
          char * OutputListName)
{
    int MoreItems; // true if items remain in both of the lists

    // initialize input and output lists
    InitializeList (1, List1Name); // initialize List 1
    InitializeList (2, List2Name); // initialize List 2
    InitializeOutput (OutputListName);

    // get first item from both lists
    MoreItems = NextItemInList(1) && NextItemInList(2);

    while (MoreItems) { // loop until no items in one of the lists
        if (Item(1) < Item(2))
            MoreItems = NextItemInList(1);
        else if (Item(1) == Item(2)) // Item1 == Item2
        {
            ProcessItem (1); // match found
            MoreItems = NextItemInList(1) && NextItemInList(2);
        }
        else // Item(1) > Item(2)
            MoreItems = NextItemInList(2);
    }
    FinishUp();
    return 1;
}
```

Figure 8.2 Cosequential match function based on a single loop.

The logic inside the loop is equally simple. Only three possible conditions can exist after reading an item: the *if-then-else* logic handles all of them. Because we are implementing a match process here, output occurs only when the items are the same.

Note that the main program does not concern itself with such matters as getting the next item, sequence checking, and end-of-file detection. Since their presence in the main loop would only obscure the main synchronization logic, they have been relegated to supporting methods. It is also the case that these methods are specific to the particular type of lists being used and must be different for different applications.

Method `NextItemInList` has a single parameter that identifies which list is to be manipulated. Its responsibility is to read the next name from the file, store it somewhere, and return *true* if it was able to read another name and *false* otherwise. It can also check the condition that the list must be in ascending order with no duplicate entries.

Method `Match` must be supported by defining methods `InitializeList`, `InitializeOutput`, `NextItemInList`, `Item`, `ProcessItem`, and `FinishUp`. The `Match` method is perfectly general and is not dependent on the type of the items nor on the way the lists are represented. These details are provided by the supporting methods that need to be defined for the specific needs of particular applications. What follows is a description of a class `CosequentialProcessing` that supports method `Match` and a class `StringListProcess` that defines the supporting operations for the lists like those of Figure 8.1.

Class `CosequentialProcessing`, as given in Fig. 8.3 and in file `coseq.h` and `coseq.cpp` of Appendix H, encapsulates the ideas of cosequential processing that were described in the earlier example of list matching. Note that this is an abstract class, since it does not contain definitions of the supporting methods. This is a template class so the operations that compare list items can be different for different applications. The code of method `Match` in Fig. 8.2 is exactly that of method `Match2Lists` of this class, as you can see in file `coseq.cpp`.

In order to use class `CosequentialProcess` for the application described earlier, we need to create a subclass `StringListProcess` that defines the specific supporting methods. Figure 8.4 (and file `strlist.h` of Appendix H) shows the definition of class `StringListProcess`. The implementations of the methods are given in file `strlist.cpp` of Appendix H. The class definition allows any number of input lists. Protected members are included for the input and

```

template <class ItemType>
class CosequentialProcess
// base class for cosequential processing
{public:
    // The following methods provide basic list processing
    // These must be defined in subclasses
    virtual int InitializeList (int ListNumber, char * ListName)=0;
    virtual int InitializeOutput (char * OutputListName)=0;
    virtual int NextItemInList (int ListNumber)=0;
        //advance to next item in this list
    virtual ItemType Item (int ListNumber) = 0;
        // return current item from this list
    virtual int ProcessItem (int ListNumber)=0;
        // process the item in this list
    virtual int FinishUp()=0; // complete the processing

    // 2-way cosequential match method
    virtual int Match2Lists
        (char * List1Name, char * List2Name, char * OutputListName);
};

```

Figure 8.3 Main members and methods of a general class for cosequential processing.

output files and for the values of the current item of each list. Member `LowValue` is a value that is smaller than any value that can appear in a list—in this case, the null string (" "). `LowValue` is used so that method `NextItemInList` does not have to get the first item in any special way. Member `HighValue` has a similar use, as we will see in the next section.

Given these classes, you should be able to work through the two lists provided in Fig. 8.1, following the code, and demonstrate to yourself that these simple procedures can handle the various resynchronization problems that these sample lists present. A main program (file `match.cpp`) to process the lists stored in files `list1.txt` and `list2.txt` is

```

#include "coseq.h"
int main ()
{
    StringListProcess ListProcess(2); // process with 2 lists
    ListProcess.Match2Lists ("list1.txt", "list2.txt", "match.txt");
}

```

```

class StringListProcess: public CosequentialProcess<String&>
// Class to process lists that are files of strings, one per line
{
public:
    StringListProcess (int NumberOfLists); // constructor

    // Basic list processing methods
    int InitializeList (int ListNumber, char * List1Name);
    int InitializeOutput (char * OutputListName);
    int NextItemInList (int ListNumber); //get next
    String& Item (int ListNumber); //return current
    int ProcessItem (int ListNumber); // process the item
    int FinishUp(); // complete the processing
protected:
    ifstream * Lists; // array of list files
    String * Items; // array of current Item from each list
    ofstream OutputList;
    static const char * LowValue;
    static const char * HighValue;
};

```

Figure 8.4 A subclass to support lists that are files of strings, one per line.

8.1.2 Merging Two Lists

The three-way-test, single-loop model for cosequential processing can easily be modified to handle *merging* of lists simply by producing output for every case of the *if-then-else* construction since a merge is a *union* of the list contents.

An important difference between matching and merging is that with merging we must read *completely* through each of the lists. This necessitates a change in how `MoreNames` is set. We need to keep this flag set to `TRUE` as long as there are records in *either* list. At the same time, we must recognize that one of the lists has been read completely, and we should avoid trying to read from it again. Both of these goals can be achieved if we introduce two `MoreNames` variables, one for each list, and set the stored `Item` value for the completed list to some value (we call it `HighValue`) that

- Cannot possibly occur as a legal input value, and

- Has a *higher* collating sequence value than any possible legal input value. In other words, this special value would come *after* all legal input values in the file's ordered sequence.

For HighValue, we use the string "\xFF" which is a string of only one character and that character has the hex value FF, which is the largest character value.

Method Merge2Lists is given in Fig. 8.5 and in file coseq.cpp of Appendix H. This method has been added to class CosequentialProcess. No modifications are required to class StringListProcess.

```

template <class ItemType>
int CosequentialProcess<ItemType>::Merge2Lists
(char * List1Name, char * List2Name, char * OutputListName)
{
    int MoreItems1, MoreItems2; // true if more items in list
    InitializeList (1, List1Name);
    InitializeList (2, List2Name);
    InitializeOutput (OutputListName);
    MoreItems1 = NextItemInList(1);
    MoreItems2 = NextItemInList(2);

    while (MoreItems1 || MoreItems2){// if either file has more
        if (Item(1) < Item(2))
            // list 1 has next item to be processed
            ProcessItem (1);
            MoreItems1 = NextItemInList(1);
        }
        else if (Item(1) == Item(2))
            // lists have the same item, process from list 1
            ProcessItem (1);
            MoreItems1 = NextItemInList(1);
            MoreItems2 = NextItemInList(2);
        }
        else // Item(1) > Item(2)
            // list 2 has next item to be processed.
            ProcessItem (2);
            MoreItems2 = NextItemInList(2);
        }
    }
    FinishUp();
    return 1;
}

```

Figure 8.5 Cosequential merge procedure based on a single loop.

Once again, you should use this logic to work, step by step, through the lists provided in Fig. 8.1 to see how the resynchronization is handled and how the use of the `HighValue` forces the procedure to finish both lists before terminating.

With these two examples, we have covered all of the pieces of our model. Now let us summarize the model before adapting it to a more complex problem.

8.1.3 Summary of the Cosequential Processing Model

Generally speaking, the model can be applied to problems that involve the performance of set operations (union, intersection, and more complex processes) on two or more sorted input files to produce one or more output files. In this summary of the cosequential processing model, we assume that there are only two input files and one output file. It is important to understand that the model makes certain general assumptions about the nature of the data and type of problem to be solved. Following is a list of the assumptions, together with clarifying comments.

Assumptions

Two or more input files are to be processed in a parallel fashion to produce one or more output files.

Each file is sorted on one or more key fields, and all files are ordered in the same ways on the same fields.

In some cases, there must exist a high-key value that is greater than any legitimate record key and a low-key value that is less than any legitimate record key.

Records are to be processed in logical sorted order.

Comments

In some cases an output file may be the same file as one of the input files.

It is not necessary that all files have the same record structures.

The use of a high-key value and a low-key value is not absolutely necessary, but it can help avoid the need to deal with beginning-of-file and end-of-file conditions as special cases, hence decreasing complexity.

The physical ordering of records is irrelevant to the model, but in practice it may be important to the way the model is implemented. Physical ordering can have a large impact on processing efficiency.

Assumptions (cont.)

For each file there is only one current record. This is the record whose key is accessible within the main synchronization loop.

Records can be manipulated only in internal memory.

Comments (cont.)

The model does not prohibit looking ahead or looking back at records, but such operations should be restricted to subclasses and should not be allowed to affect the structure of the main synchronization loop.

A program cannot alter a record in place on secondary storage.

Given these assumptions, the essential components of the model are:

1. Initialization. Previous item values for all files are set to the low value; then current records for all files are read from the first logical records in the respective files.
2. One main synchronization loop is used, and the loop continues as long as relevant records remain.
3. Within the body of the main synchronization loop is a selection based on comparison of the record keys from respective input file records. If there are two input files, the selection takes the form given in function `Match` of Fig. 8.2.
4. Input files and output files are sequence checked by comparing the previous item value with the new item value when a record is read. After a successful sequence check, the previous item value is set to the new item value to prepare for the next input operation on the corresponding file.
5. High values are substituted for actual key values when end-of-file occurs. The main processing loop terminates when high values have occurred for all relevant input files. The use of high values eliminates the need to add special code to deal with each end-of-file condition. (This step is not needed in a pure match procedure because a match procedure halts when the first end-of-file condition is encountered.)
6. All possible I/O and error detection activities are to be relegated to supporting methods so the details of these activities do not obscure the principal processing logic.

This three-way-test, single-loop model for creating cosequential processes is both simple and robust. You will find very few applications requiring the coordinated sequential processing of two files that cannot be handled neatly and efficiently with the model. We now look at a problem that is much more complex than a simple match or merge but that nevertheless lends itself nicely to solution by means of the model.

8.2 Application of the Model to a General Ledger Program

8.2.1 The Problem

Suppose we are given the problem of designing a general ledger posting program as part of an accounting system. The system includes a journal file and a ledger file. The ledger contains month-by-month summaries of the values associated with each of the bookkeeping accounts. A sample portion of the ledger, containing only checking and expense accounts, is illustrated in Fig. 8.6.

The journal file contains the monthly transactions that are ultimately to be posted to the ledger file. Figure 8.7 shows these journal transactions. Note that the entries in the journal file are paired. This is because every check involves both subtracting an amount from the checking account balance and adding an amount to at least one expense account. The accounting-program package needs procedures for creating this journal file interactively, probably outputting records to the file as checks are keyed in and then printed.

Acct. No.	Account title	Jan	Feb	Mar	Apr
101	Checking account #1	1032.57	2114.56	5219.23	
102	Checking account #2	543.78	3094.17	1321.20	
505	Advertising expense	25.00	25.00	25.00	
510	Auto expenses	195.40	307.92	501.12	
515	Bank charges	0.00	0.00	0.00	
520	Books and publications	27.95	27.95	87.40	
525	Interest expense	103.50	255.20	380.27	
535	Miscellaneous expense	12.45	17.87	23.87	
540	Office expense	57.50	105.25	138.37	
545	Postage and shipping	21.00	27.63	57.45	
550	Rent	500.00	1000.00	1500.00	
555	Supplies	112.00	167.50	2441.80	

Figure 8.6 Sample ledger fragment containing checking and expense accounts.

Acct. No	Check No.	Date	Description	Debit/ credit
101	1271	04/02/86	Auto expense	-78.70
510	1271	04/02/97	Tune-up and minor repair	78.70
101	1272	04/02/97	Rent	-500.00
550	1272	04/02/97	Rent for April	500.00
101	1273	04/04/97	Advertising	-87.50
505	1273	04/04/97	Newspaper ad re: new product	87.50
102	670	04/02/97	Office expense	-32.78
540	670	04/02/97	Printer cartridge	32.78
101	1274	04/02/97	Auto expense	-31.83
510	1274	04/09/97	Oil change	31.83

Figure 8.7 Sample journal entries.

Once the journal file is complete for a given month, meaning that it contains all of the transactions for that month, the journal must be posted to the ledger. *Posting* involves associating each transaction with its account in the ledger. For example, the printed output produced for accounts 101, 102, 505, and 510 during the posting operation, given the journal entries in Fig. 8.7, might look like the output illustrated in Fig. 8.8.

101	Checking account #1			
	1271	04/02/86	Auto expense	-78.70
	1272	04/02/97	Rent	-500.00
	1273	04/04/97	Advertising	-87.50
	1274	04/02/97	Auto expense	-31.83
			Prev. bal: 5219.23	New bal: 4521.20
102	Checking account #2			
	670	04/02/97	Office expense	-32.78
			Prev. bal: 1321.20	New bal: 1288.42
505	Advertising expense			
	1273	04/04/97	Newspaper ad re: new product	87.50
			Prev. bal: 25.00	New bal: 112.50
510	Auto expenses			
	1271	04/02/97	Tune-up and minor repair	78.70
	1274	04/09/97	Oil change	31.83
			Prev. bal: 501.12	New bal: 611.65

Figure 8.8 Sample ledger printout showing the effect of posting from the journal.

How is the posting process implemented? Clearly, it uses the account number as a *key* to relate the journal transactions to the ledger records. One possible solution involves building an index for the ledger so we can work through the journal transactions using the account number in each journal entry to look up the correct ledger record. But this solution involves seeking back and forth across the ledger file as we work through the journal. Moreover, this solution does not really address the issue of creating the output list, in which all the journal entries relating to an account are collected together. Before we could print the ledger balances and collect journal entries for even the first account, 101, we would have to proceed all the way through the journal list. Where would we save the transactions for account 101 as we collect them during this complete pass through the journal?

A much better solution is to begin by collecting all the journal transactions that relate to a given account. This involves sorting the journal transactions by account number, producing a list ordered as in Fig. 8.9.

Now we can create our output list by working through both the ledger and the sorted journal *cosequentially*, meaning that we process the two lists sequentially and in parallel. This concept is illustrated in Fig. 8.10. As we start working through the two lists, we note that we have an initial match on account number. We know that multiple entries are possible in the journal file but not in the ledger, so we move ahead to the next entry in the

Acct. No	Check No.	Date	Description	Debit/ credit
101	1271	04/02/86	Auto expense	-78.70
101	1272	04/02/97	Rent	-500.00
101	1273	04/04/97	Advertising	-87.50
101	1274	04/02/97	Auto expense	-31.83
102	670	04/02/97	Office expense	-32.78
505	1273	04/04/97	Newspaper ad re: new product	87.50
510	1271	04/02/97	Tune-up and minor repair	78.70
510	1274	04/09/97	Oil change	31.83
540	670	04/02/97	Printer cartridge	32.78
550	1272	04/02/97	Rent for April	500.00

Figure 8.9 List of journal transactions sorted by account number.

journal. The account numbers still match. We continue doing this until the account numbers no longer match. We then *resynchronize* the cosequential action by moving ahead in the ledger list. This process is often referred to as a *master-transaction process*. In this case the ledger entry is the master record and the journal entry is the transaction entry.

This matching process seems simple, as in fact it is, as long as every account in one file also appears in another. But there will be ledger accounts for which there is no journal entry, and there can be typographical errors that create journal account numbers that do not exist in the ledger. Such situations can make resynchronization more complicated and can result in erroneous output or infinite loops if the programming is done in an ad hoc way. By using the cosequential processing model, we can guard against these problems. Let us now apply the model to our ledger problem.

8.2.2 Application of the Model to the Ledger Program

The monthly ledger posting program must perform two tasks:

- It needs to update the ledger file with the correct balance for each account for the current month.
- It must produce a printed version of the ledger that not only shows the beginning and current balance for each account but also lists all the journal transactions for the month.

Ledger List		Journal List		
101	Checking account #1	101	1271	Auto expense
		101	1272	Rent
		101	1273	Advertising
		101	1274	Auto expense
102	Checking account #2	102	670	Office expense
505	Advertising expense	505	1273	Newspaper ad re: new product
510	Auto expenses	510	1271	Tune-up and minor repair
		510	1274	Oil change

Figure 8.10 Conceptual view of cosequential matching of the ledger and journal files.

We focus on the second task as it is the more difficult. Let's look again at the form of the printed output, this time extending the output to include a few more accounts as shown in Fig. 8.11. As you can see, the printed output from the monthly ledger posting program shows the balances of all ledger accounts, whether or not there were transactions for the account. From the point of view of the ledger accounts, the process is a *merge*, since even unmatched ledger accounts appear in the output.

What about unmatched journal accounts? The ledger accounts and journal accounts are not equal in authority. The ledger file *defines* the set of legal accounts; the journal file contains entries that are to be *posted* to the accounts listed in the ledger. The existence of a journal account that does not match a ledger account indicates an error. From the point of view of the journal accounts, the posting process is strictly one of *matching*. Our post method needs to implement a kind of combined merging/matching algorithm while simultaneously handling the chores of printing account title lines, individual transactions, and summary balances.

101	Checking account #1			
	1271	04/02/86	Auto expense	-78.70
	1272	04/02/97	Rent	-500.00
	1274	04/02/97	Auto expense	-31.83
	1273	04/04/97	Advertising	-87.50
		Prev. bal:	5219.23	New bal: 4521.20
102	Checking account #2			
	670	04/02/97	Office expense	-32.78
		Prev. bal:	1321.20	New bal: 1288.42
505	Advertising expense			
	1273	04/04/97	Newspaper ad re: new product	87.50
		Prev. bal:	25.00	New bal: 112.50
510	Auto expenses			
	1271	04/02/97	Tune-up and minor repair	78.70
	1274	04/09/97	Oil change	31.83
		Prev. bal:	501.12	New bal: 611.65
515	Bank charges			
		Prev. bal:	0.00	New bal: 0.00
520	Books and publications			
		Prev. bal:	87.40	New bal: 87.40

Figure 8.11 Sample ledger printout for the first six accounts.

In summary, there are three different steps in processing the ledger entries:

1. Immediately after reading a new ledger object, we need to print the header line and initialize the balance for the next month from the previous month's balance.
2. For each transaction object that matches, we need to update the account balance.
3. After the last transaction for the account, the balance line should be printed. This is the place where a new ledger record could be written to create a new ledger file.

This posting operation is encapsulated by defining subclass `MasterTransactionProcess` of `CosequentialProcess` and defining three new pure virtual methods, one for each of the steps in processing ledger entries. Then we can give the full implementation of the posting operation as a method of this class. Figure 8.12 shows the definition of this class. Figure 8.13 has the code for the three-way-test loop of method `PostTransactions`. The new methods of the class are used for processing the master records (in this case the ledger entries). The transaction records (journal entries) can be processed by the `ProcessItem` method that is in the base class.

The reasoning behind the three-way test is as follows:

1. If the ledger (master) account number (`Item[1]`) is less than the journal (transaction) account number (`Item[2]`), then there are no more transactions to add to the ledger account this month (perhaps there were none at all), so we print the ledger account balances (`ProcessEndMaster`) and read in the next ledger account (`NextItemInList(1)`). If the account exists (`MoreMasters` is true), we print the title line for the new account (`ProcessNewMaster`).
2. If the account numbers match, then we have a journal transaction that is to be posted to the current ledger account. We add the transaction amount to the account balance for the new month (`ProcessCurrentMaster`), print the description of the transaction (`ProcessItem(2)`), then read the next journal entry (`NextItemInList(1)`). Note that unlike the match case in either the matching or merging algorithms, we do not read a new entry from both accounts. This is a reflection of our acceptance of more than one journal entry for a single ledger account.

```

template <class ItemType>
class MasterTransactionProcess:
    public CosequentialProcess<ItemType>
// a cosequential process that supports
// master/transaction processing
{public:
    MasterTransactionProcess (); //constructor
    virtual int ProcessNewMaster ()=0;
        // processing when new master read
    virtual int ProcessCurrentMaster ()=0;
        // processing for each transaction for a master
    virtual int ProcessEndMaster ()=0;
        // processing after all transactions for a master
    virtual int ProcessTransactionError ()=0;
        // no master for transaction

    // cosequential processing of master and transaction records
    int PostTransactions (char * MasterFileName,
        char * TransactionFileName, char * OutputListName);
};

```

Figure 8.12 Class MasterTransactionProcess.

```

while (MoreMasters || MoreTransactions)
    if (Item(1) < Item(2)){ // finish this master record
        ProcessEndMaster();
        MoreMasters = NextItemInList(1);
        if (MoreMasters) ProcessNewMaster();
    }
    else if (Item(1) == Item(2)){ // transaction matches master
        ProcessCurrentMaster(); // another transaction for master
        ProcessItem (2); // output transaction record
        MoreTransactions = NextItemInList(2);
    }
    else { // Item(1) > Item(2) transaction with no master
        ProcessTransactionError();
        MoreTransactions = NextItemInList(2);
    }
}

```

Figure 8.13 Three-way-test loop for method PostTransactions of class MasterTransactionProcess.

3. If the journal account is less than the ledger account, then it is an unmatched journal account, perhaps due to an input error. We print an error message (`ProcessTransactionError`) and continue with the next transaction.

In order to complete our implementation of the ledger posting application, we need to create a subclass `LedgerProcess` that includes implementation of the `NextItemInList`, `Item`, and `ProcessItem` methods and the methods for the three steps of master record processing. This new class is given in files `ledgpost.h` and `ledgpost.cpp` of Appendix H. The master processing methods are all very simple, as shown in Fig. 8.14.

The remainder of the code for the ledger posting program, including the simple main program, is given in files `ledger.h`, `ledger.cpp`, and `post.cpp` in Appendix H. This includes the `ostream` formatting that produced Figs. 8.8 and 8.11. The classes `Ledger` and `Journal` make extensive use of the `IOBuffer` and `RecordFile` classes for their file operations.

The development of this ledger posting procedure from our basic cosequential processing model illustrates how the simplicity of the model contributes to its adaptability. We can also generalize the model in an entirely different direction, extending it to enable cosequential processing

```
int LedgerProcess::ProcessNewMaster ()
{
    // print the header and setup last month's balance
    ledger.PrintHeader(OutputList);
    ledger.Balances[MonthNumber] = ledger.Balances[MonthNumber-1];
}

int LedgerProcess::ProcessCurrentMaster ()
{
    // add the transaction amount to the balance for this month
    ledger.Balances[MonthNumber] += journal.Amount;
}

int LedgerProcess::ProcessEndMaster ()
{
    // print the balances line to output
    PrintBalances(OutputList,
        ledger.Balances[MonthNumber-1], ledger.Balances[MonthNumber]);
}

```

Figure 8.14 Master record processing for ledger objects.

of more than two input files at once. To illustrate this, we now extend the model to include multiway merging.

8.3 Extension of the Model to Include Multiway Merging

The most common application of cosequential processes requiring more than two input files is a *K-way merge*, in which we want to merge *K* input lists to create a single, sequentially ordered output list. *K* is often referred to as the *order* of a *K*-way merge.

8.3.1 A *K*-way Merge Algorithm

Recall the synchronizing loop we use to handle a two-way merge of two lists of names. This merging operation can be viewed as a process of deciding which of two input items has the *minimum* value, outputting that item, then moving ahead in the list from which that item is taken. In the event of duplicate input items, we move ahead in each list.

Suppose we keep an array of lists and array of the items (or keys) that are being used from each list in the cosequential process:

```
list[0], list[1], list[2], ... list[k-1]
Item[0], Item[1], Item[3], ... Item[k-1]
```

The main loop for the merge processing requires a call to a *MinIndex* function to find the index of item with the minimum collating sequence value and an inner loop that finds all lists that are using that item:

```
int minItem = MinIndex(Item,k); // find an index of minimum item
ProcessItem(minItem); // Item(minItem) is the next output
for (i = 0; i<k; i++) // look at each list
    if (Item(minItem) == Item(i)) // advance list i
        MoreItems[i] = NextItemInList(i);
```

Clearly, the expensive parts of this procedure are finding the minimum and testing to see in which lists the item occurs and which files therefore need to be read. Note that because the item can occur in several lists, every one of these *if* tests must be executed on every cycle through the loop. However, it is often possible to guarantee that a single item, or key, occurs in only one list. In this case, the procedure becomes simpler and more efficient.

```
int minI = minIndex(Item,k); // find index of minimum item
ProcessItem(minI); // Item[minI] is the next output
MoreItems[minI]=NextItemInList(minI);
```

The resulting merge procedure clearly differs in many ways from our initial three-way-test, single-loop merge for two lists. But, even so, the single-loop parentage is still evident: there is no looping within a list. We determine which lists have the key with the lowest value, output that key, move ahead one key in each of those lists, and loop again. The procedure is as simple as it is powerful.

8.3.2 A Selection Tree for Merging Large Numbers of Lists

The K -way merge described earlier works nicely if K is no larger than 8 or so. When we begin merging a larger number of lists, the set of sequential comparisons to find the key with the minimum value becomes noticeably expensive. We see later that for practical reasons it is rare to want to merge more than eight files at one time, so the use of sequential comparisons is normally a good strategy. If there is a need to merge considerably more than eight lists, we could replace the loop of comparisons with a *selection tree*.

The use of a selection tree is an example of the classic time-versus-space trade-off we so often encounter in computer science. We reduce the time required to find the key with the lowest value by using a data structure to save information about the relative key values across cycles of the procedure's main loop. The concept underlying a selection tree can be readily communicated through a diagram such as that in Fig. 8.15. Here we have used lists in which the keys are numbers rather than strings.

The selection tree is a kind of *tournament tree* in which each higher-level node represents the “winner” (in this case the *minimum* key value) of the comparison between the two descendent keys. The minimum value is always at the root node of the tree. If each key has an associated reference to the list from which it came, it is a simple matter to take the key at the root, read the next element from the associated list, then run the tournament again. Since the tournament tree is a binary tree, its depth is

$$\lceil \log_2 K \rceil$$

for a merge of K lists. The number of comparisons required to establish a new tournament winner is, of course, related to this depth rather than being a linear function of K .

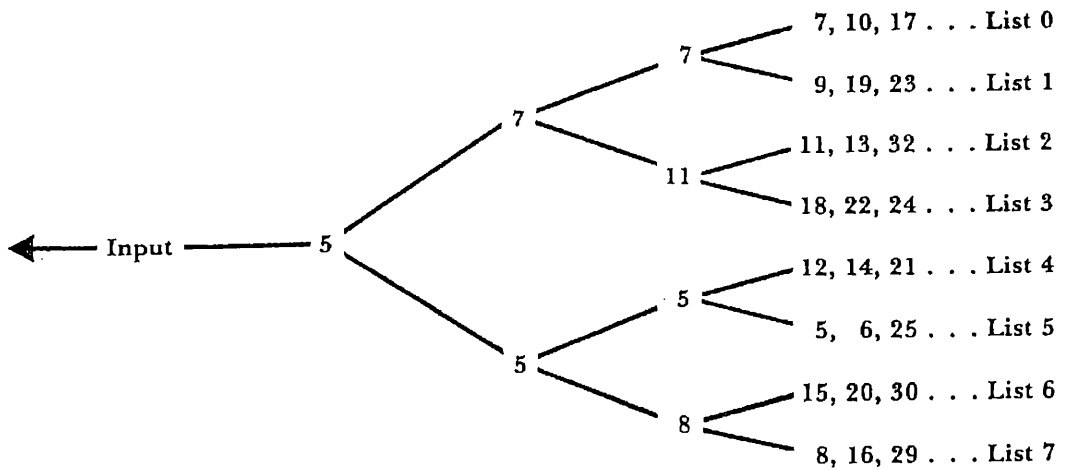


Figure 8.15 Use of a selection tree to assist in the selection of a key with minimum value in a K -way merge.

8.4 A Second Look at Sorting in Memory

In Chapter 6 we considered the problem of sorting a disk file that is small enough to fit in memory. The operation we described involves three separate steps:

1. Read the entire file from disk into memory.
2. Sort the records using a standard sorting procedure, such as shellsort.
3. Write the file back to disk.

The total time taken to sort the file is the sum of the times for the three steps. We see that this procedure is much faster than sorting the file in place, on the disk, because both reading and writing are sequential and each record is read once and written once.

Can we improve on the time that it takes for this memory sort? If we assume that we are reading and writing the file as efficiently as possible and we have chosen the best internal sorting routine available, it would seem not. Fortunately, there is one way that we might speed up an algorithm that has several parts, and that is to perform some of those parts in parallel.

Of the three operations involved in sorting a file that is small enough to fit into memory, is there any way to perform some of them in parallel? If we have only one disk drive, clearly we cannot overlap the reading and writing operations, but how about doing either the reading or writing (or both) at the same time that we sort the file?

8.4.1 Overlapping Processing and I/O: Heapsort

Most of the time when we use an internal sort, we have to wait until we have the whole file in memory before we can start sorting. Is there an internal sorting algorithm that is reasonably fast and that can begin sorting numbers immediately as they are read rather than waiting for the whole file to be in memory? In fact there is, and we have already seen part of it in this chapter. It is called *heapsort*, and it is loosely based on the same principle as the selection tree.

Recall that the selection tree compares keys as it encounters them. Each time a new key arrives, it is compared with the others; and if it is the smallest key, it goes to the root of the tree. This is very useful for our purposes because it means that we can begin sorting keys as they arrive in memory rather than waiting until the entire file is loaded before we start sorting. That is, sorting can occur in parallel with reading.

Unfortunately, in the case of the selection tree, each time a new smallest key is found, it is output to the file. We cannot allow this to happen if we want to sort the whole file because we cannot begin outputting records until we know which one comes first, second, and so on, and we won't know this until we have seen all of the keys.

Heapsort solves this problem by keeping all of the keys in a structure called a *heap*. A heap is a binary tree with the following properties:

1. Each node has a single key, and that key is greater than or equal to the key at its parent node.
2. It is a *complete* binary tree, which means that all of its leaves are on no more than two levels and that all of the keys on the lower level are in the leftmost position.
3. Because of properties 1 and 2, storage for the tree can be allocated sequentially as an array in such a way that the root node is index 1 and the indexes of the left and right children of node i are $2i$ and $2i + 1$, respectively. Conversely, the index of the parent of node j is $\lfloor j/2 \rfloor$.

Figure 8.16 shows a heap in both its tree form and as it would be stored in an array. Note that this is only one of many possible heaps for the given set of keys. In practice, each key has an associated record that is either stored in the array with the key or pointed to by a pointer stored with the key.

Property 3 is very useful for our purposes because it means that a heap is just an array of keys in which the positions of the keys in the array are sufficient to impose an ordering on the entire set of keys. There is no need

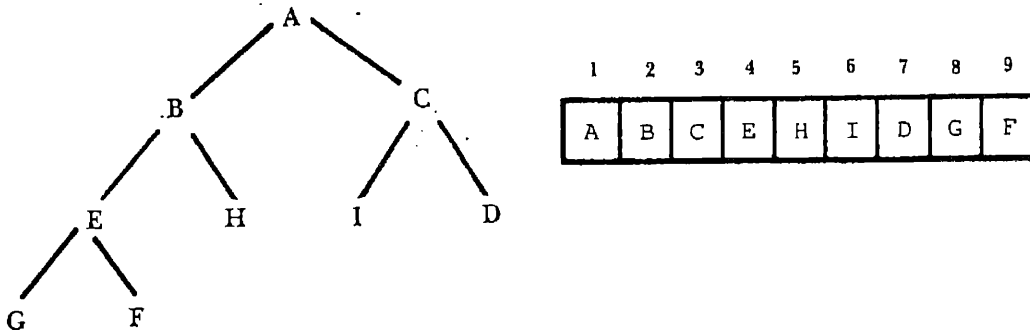


Figure 8.16 A heap in both its tree form and as it would be stored in an array.

for pointers or other dynamic data structuring overhead to create and maintain the heap. (As we pointed out earlier, there may be pointers associating each key with its corresponding record, but this has nothing to do with maintaining the heap.)

8.4.2 Building the Heap While Reading the File

The algorithm for heapsort has two parts. First we build the heap; then we output the keys in sorted order. The first stage can occur at virtually the same time that we read the data, so in terms of elapsed time it comes essentially free. The main members of a simple class `Heap` and its `Insert` method that adds a string to the heap is shown in Fig. 8.17. A full implementation of this class and a test program are in file `heapsort.cpp` in Appendix H. Figure 8.18 contains a sample application of this algorithm.

This shows how to build the heap, but it doesn't tell how to make the input overlap with the heap-building procedure. To solve that problem, we need to look at how we perform the read operation. For starters, we are not going to do a seek every time we want a new record. Instead, we read a block of records at a time into an input buffer and then operate on all of the records in the block before going on to the next block. In terms of memory storage, the input buffer for each new block of keys can be part of the memory area that is set up for the heap. Each time we read a new block, we just append it to the end of the heap (that is, the input buffer "moves" as the heap gets larger). The first new record is then at the end of the heap array, as required by the `Insert` function (Fig. 8.17). Once that record is absorbed into the heap, the next new record is at the end of the heap array, ready to be absorbed into the heap, and so forth.

```

class Heap
{public:
    Heap(int maxElements);
    int Insert (char * newKey);
    char * Remove();
protected:
    int MaxElements; int NumElements;
    char ** HeapArray;
    void Exchange(int i, int j); // exchange element i and j
    int Compare (int i, int j) // compare element i and j
        {return strcmp(HeapArray[i],HeapArray[j]);}
};
int Heap::Insert(char * newKey)
{
    if (NumElements == MaxElements) return FALSE;
    NumElements++; // add the new key at the last position
    HeapArray[NumElements] = newKey;
    // re-order the heap
    int k = NumElements; int parent;
    while (k > 1) // k has a parent
    {
        parent = k / 2;
        if (Compare(k, parent) >= 0) break;
        // HeapArray[k] is in the right place
        // else exchange k and parent
        Exchange(k, parent);
        k = parent;
    }
    return TRUE;
}

```

Figure 8.17 Class Heap and method Insert.

Use of an input buffer avoids an excessive number of seeks, but it still doesn't let input occur at the same time that we build the heap. We saw in Chapter 3 that the way to make processing overlap with I/O is to use more than one buffer. With multiple buffering, as we process the keys in one block from the file, we can simultaneously read later blocks from the file. If we use multiple buffers, how many should we use, and where should we put them? We already answered these questions when we decided to put each new block at the end of the array. Each time we

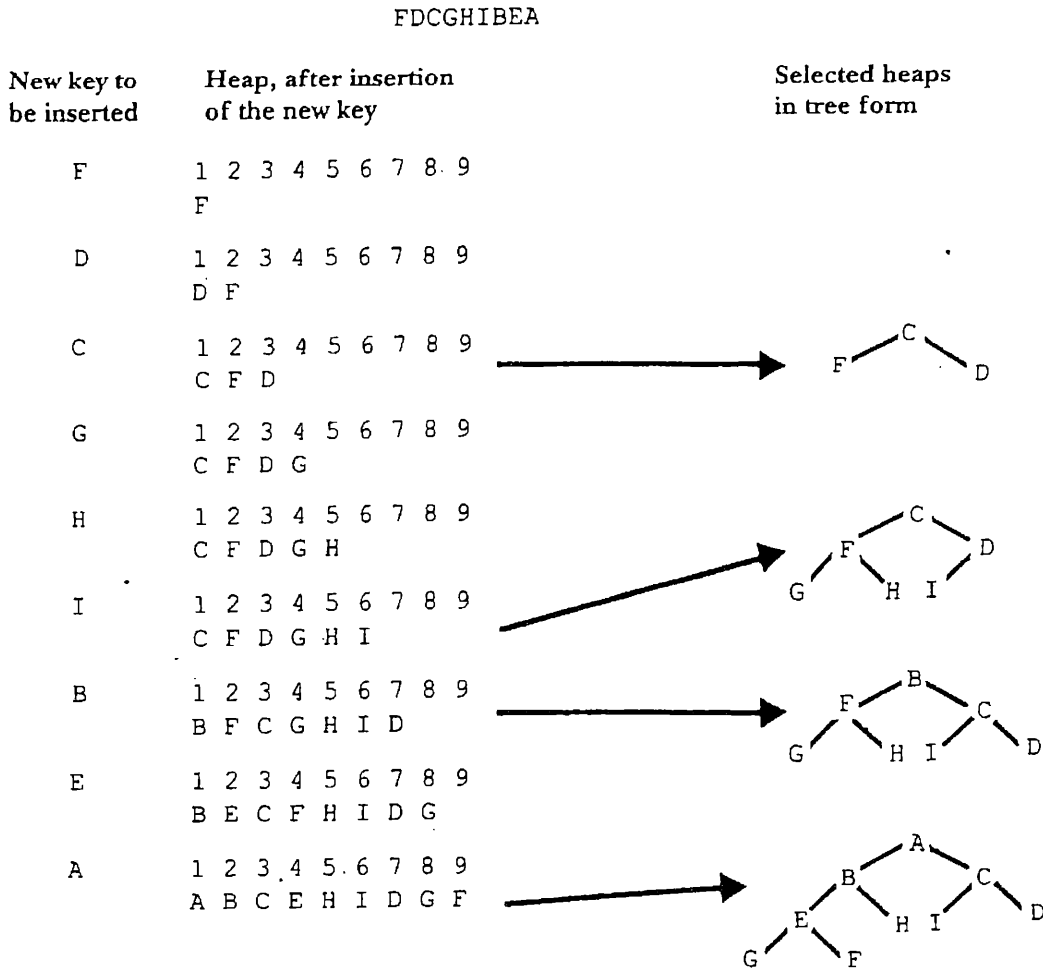


Figure 8.18 Sample application of the heap-building algorithm. The keys F, D, C, G, H, I, B, E, and A are added to the heap in the order shown.

add a new block, the array gets bigger by the size of that block, in effect creating a new input buffer for each block in the file. So the number of buffers is the number of blocks in the file, and they are located in sequence in the array.

Figure 8.19 illustrates the technique that we have just described, in which we append each new block of records to the end of the heap, thereby employing a memory-sized set of input buffers. Now we read new blocks as fast as we can, never having to wait for processing before reading a new block. On the other hand, processing (heap building) cannot occur on a given block until the block to be processed is read, so there *may* be some delay in processing if processing speeds are faster than reading speeds.

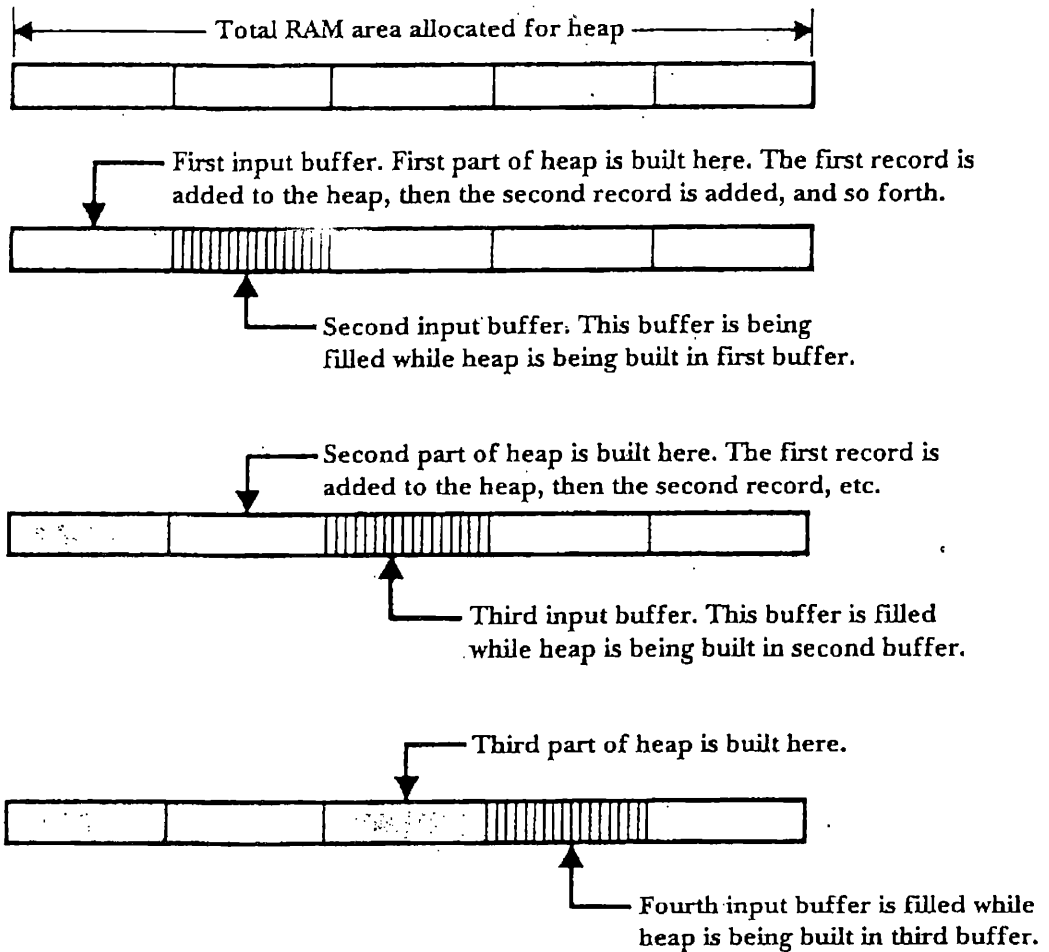


Figure 8.19 Illustration of the technique described in the text for overlapping input with heap building in memory. First read in a block into the first part of memory. The first record is the first record in the heap. Then extend the heap to include the second record, and incorporate that record into the heap, and so forth. While the first block is being processed, read in the second block. When the first block is a heap, extend it to include the first record in the second block, incorporate that record into the heap, and go on to the next record. Continue until all blocks are read in and the heap is completed.

8.4.3 Sorting While Writing to the File

The second and final step involves writing the heap in sorted order. Again, it is possible to overlap I/O (in this case writing) with processing. First, let's look at how to output the sorted keys. Retrieving the keys in order is simply a repetition of the following steps:

1. Determine the value of the key in the first position of the heap. This is the smallest value in the heap.

2. Move the largest value in the heap into the first position, and decrease the number of elements by one. The heap is now out of order at its root.
3. Reorder the heap by exchanging the largest element with the smaller of its children and moving down the tree to the new position of the largest element until the heap is back in order.

Each time these three steps are executed, the smallest value is retrieved and removed from the heap. Figure 8.20 contains the code for method `Remove` that implements these steps. Method `Compare` simply compares two heap elements and returns `-1` if the left element is smaller.

Again, there is nothing inherent in this algorithm that lets it overlap with I/O, but we can take advantage of certain features of the algorithm to

```
char * Heap::Remove()
{
    // remove the smallest element, reorder the heap,
    // and return the smallest element
    // put the smallest value into 'val' for use in return
    char * val = HeapArray[1];

    // put largest value into root
    HeapArray[1] = HeapArray[NumElements];
    // decrease the number of elements
    NumElements--;

    // reorder the heap by exchanging and moving down
    int k = 1; // node of heap that contains the largest value
    int newK; // node to exchange with largest value
    while (2*k <= NumElements) // k has at least one child
    {
        // set newK to the index of smallest child of k
        if (Compare(2*k, 2*k+1) < 0) newK = 2*k;
        else newK = 2*k+1;
        // done if k and newK are in order
        if (Compare(k, newK) < 0) break; // in order
        Exchange(k, newK); // k and newK out of order
        k = newK; // continue down the tree
    }
    return val;
}
```

Figure 8.20 Method `Remove` of class `Heap` removes the smallest element and reorders the heap.

make overlapping happen. First, we see that we know immediately which record will be written first in the sorted file; next, we know what will come second; and so forth. So as soon as we have identified a block of records, we can write that block, and while we are writing that block, we can identify the next block, and so forth.

Furthermore, each time we identify a block to write, we make the heap smaller by exactly the size of a block, freeing that space for a new output buffer. So just as was the case when building the heap, we can have as many output buffers as there are blocks in the file. Again, a little coordination is required between processing and output, but the conditions exist for the two to overlap almost completely.

A final point worth making about this algorithm is that all I/O it performs is essentially sequential. All records are read in the order in which they occur in the file to be sorted, and all records are written in sorted order. The technique could work equally well if the file were kept on tape or disk. More important, since all I/O is sequential, we know that it can be done with a minimum amount of seeking.

8.5 Merging as a Way of Sorting Large Files on Disk

In Chapter 6 we ran into problems when we needed to sort files that were too large to be wholly contained in memory. The chapter offered a partial, but ultimately unsatisfactory, solution to this problem in the form of a *keysort*, in which we needed to hold only the keys in memory, along with pointers to each key's corresponding record. Keysort had two shortcomings:

1. Once the keys were sorted, we then had to bear the substantial cost of seeking to each record in sorted order, reading each record in and then writing it into the new, sorted file.
2. With keysorting, the size of the file that can be sorted is limited by the number of key/pointer pairs that can be contained in memory. Consequently, we still cannot sort really large files.

As an example of the kind of file we cannot sort with either a memory sort or a keysort, suppose we have a file with 8 000 000 records, each of which is 100 bytes long and contains a key field that is 10 bytes long. The total length of this file is about 800 megabytes. Let us further suppose that we have 10 megabytes of memory available as a work area, not counting

memory used to hold the program, operating system, I/O buffers, and so forth. Clearly, we cannot sort the whole file in memory. We cannot even sort all the keys in memory, because it would require 80 megabytes.

The multiway merge algorithm discussed in Section 8.3 provides the beginning of an attractive solution to the problem of sorting large files such as this one. Since memory-sorting algorithms such as heapsort can work in place, using only a small amount of overhead for maintaining pointers and some temporary variables, we can create a sorted subset of our full file by reading records into memory until the memory work area is almost full, sorting the records in this work area, then writing the sorted records back to disk as a sorted subfile. We call such a sorted subfile a *run*. Given the memory constraints and record size in our example, a run could contain approximately

$$\frac{10\,000\,000 \text{ bytes of memory}}{100 \text{ bytes per record}} = 100\,000 \text{ records}$$

Once we create the first run, we then read a new set of records, once again filling memory, and create another run of 100 000 records. In our example, we repeat this process until we have created eighty runs, with each run containing 100 000 sorted records.

Once we have the eighty runs in eighty separate files on disk, we can perform an eighty-way merge of these runs, using the multiway merge logic outlined in Section 8.3, to create a completely sorted file containing all the original records. A schematic view of this run creation and merging process is provided in Fig. 8.21.

This solution to our sorting problem has the following features:

- It can, in fact, sort large files and can be extended to files of any size.
- Reading of the input file during the run-creation step is sequential and hence is much faster than input that requires seeking for every record individually (as in a keysort).
- Reading through each run during merging and writing the sorted records is also sequential. Random accesses are required only as we switch from run to run during the merge operation.
- If a heapsort is used for the in-memory part of the merge, as described in Section 8.4, we can overlap these operations with I/O so the in-memory part does not add appreciably to the total time for the merge.
- Since I/O is largely sequential, tapes can be used if necessary for both input and output operations.

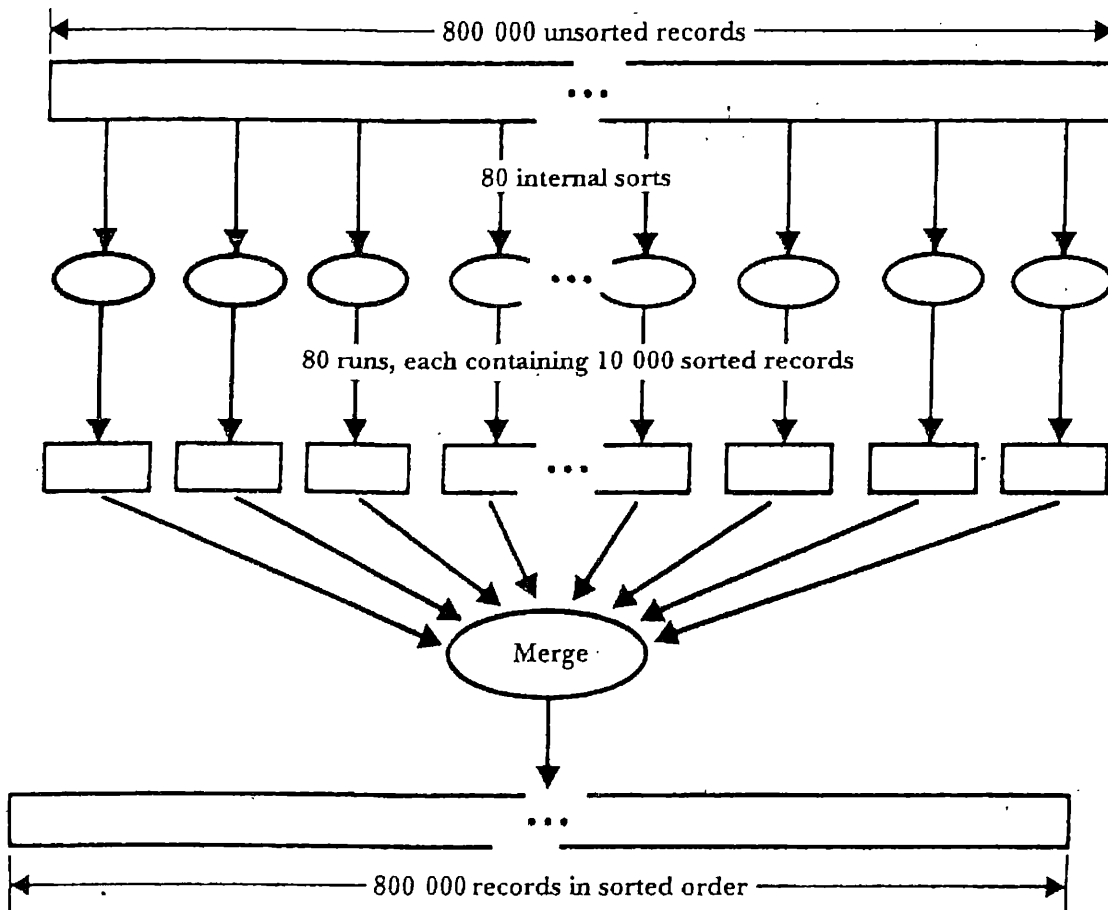


Figure 8.21 Sorting through the creation of runs (sorted subfiles) and subsequent merging of runs.

8.5.1 How Much Time Does a Merge Sort Take?

This general approach to the problem of sorting large files looks promising. To compare this approach with others, we now look at how much time it takes. We do this by taking our 8 million record files and seeing how long it takes to do a merge sort on the Seagate Cheetah 9 disk drive whose specifications are listed in Table 3.1. You might recall that this was the fastest disk available for PCs in early 1997. Please note that our intention here is not to derive time estimates that mean anything in any environment other than the hypothetical environment we have posited. Nor do we want to overwhelm you with numbers or provide you with magic formulas for determining how long a particular sort on a real system will *really* take. Rather, our goal in this section is to derive some benchmarks that we can use to compare several variations on the basic merge sort approach to sorting external files.

We can simplify matters by making the following simplifying assumptions about the computing environment:

- Entire files are always stored in contiguous areas on disk (extents), and a single cylinder-to-cylinder seek takes no time. Hence, *only one seek is required for any single sequential access.*
- Extents that span more than one track are physically staggered in such a way that *only one rotational delay is required per access.*

We see in Fig. 8.21 that there are four times when I/O is performed. During the sort phase:

- Reading all records into memory for sorting and forming runs, and
- Writing sorted runs to disk.

During the merge phase:

- Reading sorted runs into memory for merging, and
- Writing sorted file to disk.

Let's look at each of these in order.

Step 1: Reading Records into Memory for Sorting and Forming Runs

Since we sort the file in 10-megabyte chunks, we read 10 megabytes at a time from the file. In a sense, memory is a 10-megabyte input buffer that we fill up eighty times to form the eighty runs. In computing the total time to input each run, we need to include the amount of time it takes to *access* each block (seek time + rotational delay), plus the amount of time it takes to *transfer* each block. We keep these two times separate because, as we see later in our calculations, the role that each plays can vary significantly depending on the approach used.

From Table 3.1 we see that seek and rotational delay times are 8 msec¹ and 3 msec, respectively, so total time per seek is 11 msec.² The transmission rate is approximately 14 500 bytes per msec. Total input time for the sort phase consists of the time required for 80 seeks, plus the time required to transfer 800 megabytes:

1. Unless the computing environment has many active users pulling the read/write head to other parts of the disk, seek time is likely to be less than the average, since many of the blocks that make up the file are probably going to be physically adjacent to one another on the disk. Many will be on the same cylinder, requiring no seeks at all. However, for simplicity we assume the average seek time.

2. For simplicity, we use the term *seek* even though we really mean *seek and rotational delay*. Hence, the time we give for a seek is the time that it takes to perform an average seek followed by an average rotational delay.

Access :	$80 \text{ seeks} \times 11 \text{ msec} = 1 \text{ sec}$
Transfer :	$800 \text{ megabytes @ } 14\,500 \text{ bytes/msec} = 60 \text{ sec}$
Total :	61 sec

Step 2: Writing Sorted Runs to Disk

In this case, writing is just the reverse of reading—the same number of seeks and the same amount of data to transfer. So it takes another 61 seconds to write the 80 sorted runs.

Step 3: Reading Sorted Runs into Memory for Merging

Since we have 10 megabytes of memory for storing runs, we divide 10 megabytes into 80 parts for buffering the 80 runs. In a sense, we are re-allocating our 10 megabytes of memory as 80 input buffers. Each of the 80 buffers then holds 1/80th of a run (125 000 bytes), so we have to access *each* run 80 times to read all of it. Because there are 80 runs, in order to complete the merge operation (Fig. 8.22) we end up making

$$80 \text{ runs} \times 80 \text{ seeks} = 6400 \text{ seeks.}$$

Total seek and rotation time is then $6400 \times 11 \text{ msec} = 70 \text{ seconds}$. Since 800 megabytes is still transferred, transfer time is still 60 seconds.

Step 4: Writing Sorted File to Disk

To compute the time for writing the file, we need to know how big our output buffers are. Unlike steps 1 and 2, where our big memory sorting

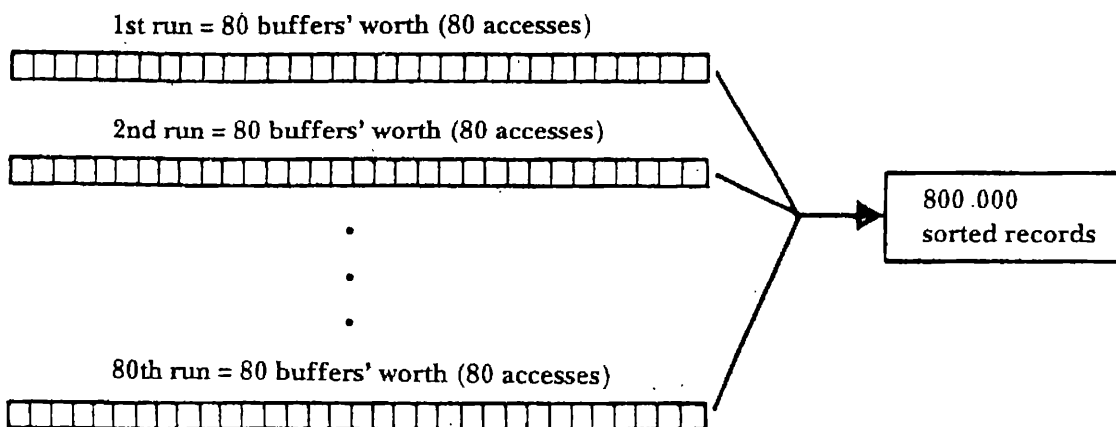


Figure 8.22 Effect of buffering on the number of seeks required, where each run is as large as the available work area in memory.

space doubled as our I/O buffer, we are now using that memory space for storing the data from the runs *before* it is merged. To keep matters simple, let us assume that we can allocate two 200 000-byte output buffers.³ With 200 000-byte buffers, we need to make

$$\frac{800\,000\,000}{200\,000 \text{ bytes per seek}} = 4000 \text{ seeks.}$$

Total seek and rotation time is then $4000 \times 11 \text{ msec} = 44 \text{ seconds}$. Transfer time is still 60 seconds.

The time estimates for the four steps are summarized in the first row in Table 8.1. The total time for this merge sort is 356 seconds, or 5 minutes, 56 seconds. The sort phase takes 122 seconds, and the merge phase takes 234 seconds.

To gain an appreciation of the improvement that this merge sort approach provides us, we need only look at how long it would take us to do one part of a nonmerging method like the key sort method described in Chapter 6. The last part of the key sort algorithm (Fig. 6.16) consists of this for loop:

```
// write new file in key order
for (int j = 0; j < inFile . NumRecs(); j++)
{
    inFile . ReadByRRN (obj, Keys[j] . RRN); // read in key order
    outFile . Append (obj); // write in key order
}
```

3. We use two buffers to allow double buffering; we use 20 000 bytes per buffer because that is approximately the size of a track on our hypothetical disk drive.

Table 8.1 Time estimates for merge sort of 80-megabyte file, assuming use of the Seagate Cheetah 9 disk drive described in Table 3.1. The total time for the sort phase (steps 1 and 2) is 14 seconds, and the total time for the merge phase is 126 seconds.

	Number of seeks	Amount transferred (megabytes)	Seek + rotation time (seconds)	Transfer time (seconds)	Total time (seconds)
Sort: reading	800	800	1	60	61
Sort: writing	800	800	1	60	61
Merge: reading	6400	800	70	60	130
Merge: writing	4000	800	44	60	104
Totals	10 560	3200	116	240	356

This *for* loop requires us to do a separate seek for every record in the file. That is 8 000 000 seeks. At 11 msec per seek, the total time required to perform that one operation works out to 88 000 seconds, or 24 hours, 26 minutes, 40 seconds!

Clearly, for large files the merge sort approach in general is the best option of any that we have seen. Does this mean that we have found the best technique for sorting large files? If sorting is a relatively rare event and files are not too large, the particular approach to merge sorting that we have just looked at produces acceptable results. Let's see how those results stand up as we change some of the parameters of our sorting example.

8.5.2 Sorting a File That Is Ten Times Larger

The first question that comes to mind when we ask about the general applicability of a computing technique is: What happens when we make the problem bigger? In this instance, we need to ask how this approach stands up as we scale up the size of the file.

Before we look at how a bigger file affects the performance of our merge sort, it will help to examine the *kinds* of I/O that are being done in the two different phases—the sort phase and the merge phase. We will see that for the purposes of finding ways to improve on our original approach, we need pay attention only to one of the two phases.

A major difference between the sort phase and the merge phase is in the amount of sequential (versus random) access that each performs. By using heapsort to create runs during the sort phase, we guarantee that all I/O is, in a sense, sequential.⁴ Since sequential access implies minimal seeking, we cannot *algorithmically* speed up I/O during the sort phase. No matter what we do with the records in the file, we have to read them and write them all at least once. Since we cannot improve on this phase by changing the way we do the sort or merge, we ignore the sort phase in the analysis that follows.

The merge phase is a different matter. In particular, the *reading step* of the merge phase is different. Since there is a memory buffer for each run, and these buffers get loaded and reloaded at unpredictable times, the read step of the merge phase is, to a large extent, one in which random accesses

4. It is *not* sequential in the sense that in a multiuser environment there will be other users pulling the read/write head to other parts of the disk between reads and writes, possibly forcing the disk to do a seek each time it reads or writes a block.

are the norm. Furthermore, the number and size of the memory buffers that we read the run data into determine the number of times we have to do random accesses. If we can somehow reconfigure these buffers in ways that reduce the number of random accesses, we can speed up I/O correspondingly. So, if we are going to look for ways to improve performance in a merge sort algorithm, *our best hope is to look for ways to cut down on the number of random accesses that occur while reading runs during the merge phase.*

What about the write step of the merge phase? Like the steps of the sort phase, this step is not influenced by differences in the way we organize runs. Improvements in the way we organize the merge sort do not affect this step. On the other hand, we will see later that it is helpful to include this phase when we measure the results of changes in the organization of the merge sort.

To sum up, since the merge phase is the only one in which we can improve performance by improving the method, we concentrate on it from now on. Now let's get back to the question that we started this section with: What happens when we make the problem bigger? How, for instance, is the time for the merge phase affected if our file is 80 million records rather than 8 million?

If we increase the size of our file by a factor of 10 without increasing the memory space, we clearly need to create more runs. Instead of 80 initial 100 000-record runs, we now have 800 runs. This means we have to do an 800-way merge in our 10 megabytes of memory space. This, in turn, means that during the merge phase we must divide memory into 800 buffers. Each of the 800 buffers holds 1/800th of a run, so we would end up making 800 seeks per run, and

$$800 \text{ runs} \times 800 \text{ seeks/run} = 640\,000 \text{ seeks altogether}$$

The times for the merge phase are summarized in Table 8.2. Note that the total time is more than 2 hours and 24 minutes, almost 25 times greater than for the 800-megabyte file. By increasing the size of our file, we have gotten ourselves back into the situation we had with keysort, in which we can't do the job we need to do without doing a huge amount of seeking. In this instance, by increasing the order of the merge from 80 to 800, we made it necessary to divide our 10-megabyte memory area into 800 tiny buffers for doing I/O; and because the buffers are tiny, each requires many seeks to process its corresponding run.

If we want to improve performance, clearly we need to look for ways to improve on the amount of time spent getting to the data during the merge phase. We will do this shortly, but first let us generalize what we have just observed.

Table 8.2 Time estimates for merge sort of 8000-megabyte file, assuming use of the Seagate Cheetah 9 disk drive described in Table 3.1. The total time for the merge phase is 7600 seconds, or 2 hours, 6 minutes, 40 seconds.

	Number of seeks	Amount transferred (megabytes)	Seek + rotation time (seconds)	Transfer time (seconds)	Total time (seconds)
Merge: reading	640 000	8000	7040	600	7640
Merge: writing	40 000	8000	440	600	1040
Totals	680 000	16 000	7480	1200	8680

8.5.3 The Cost of Increasing the File Size

Obviously, the big difference between the time it took to merge the 800-megabyte file and the 8000-megabyte file was due to the difference in total seek and rotational delay times. You probably noticed that the number of seeks for the larger file is 100 times the number of seeks for the first file, and 100 is the square of the difference in size between the two files. We can formalize this relationship as follows: in general, for a K -way merge of K runs where each run is as large as the memory space available, the buffer size for each of the runs is

$$\left(\frac{1}{K}\right) \times \text{size of memory space} = \left(\frac{1}{K}\right) \times \text{size of each run}$$

so K seeks are required to read all of the records in each individual run. Since there are K runs altogether, the merge operation requires K^2 seeks. Hence, measured in terms of seeks, our sort merge is an $O(K^2)$ operation. Because K is directly proportional to N (if we increase the number of records from 8 000 000 to 80 000 000, K increases from 80 to 800) it also follows that our sort merge is an $O(N^2)$ operation, measured in terms of seeks.

This brief, formal look establishes the principle that as files grow large, we can expect the time required for our merge sort to increase rapidly. It would be very nice if we could find some ways to reduce this time. Fortunately, there are several ways:

- Allocate more hardware, such as disk drives, memory, and I/O channels;

- Perform the merge in more than one step, reducing the order of each merge and increasing the buffer size for each run;
- Algorithmically increase the lengths of the initial sorted runs; and
- Find ways to overlap I/O operations.

In the following sections we look at each of these ways in detail, beginning with the first: invest in more hardware.

8.5.4 Hardware-Based Improvements

We have seen that changes in our sorting algorithm can improve performance. Likewise, we can make changes in our hardware that will also improve performance. In this section we look at three possible changes to a system configuration that could lead to substantial decreases in sort time:

- Increasing the amount of memory,
- Increasing the number of disk drives, and
- Increasing the number of I/O channels.

Increasing the Amount of Memory

It should be clear now that when we have to divide limited buffer space into many small buffers, we increase seek and rotation times to the point where they overwhelm all other sorting operations. Roughly speaking, the increase in the number of seeks is proportional to the square of the increase in file size, given a fixed amount of total buffer space.

It stands to reason, then, that increasing memory space ought to have a substantial effect on total sorting time. A larger memory size means longer and fewer initial runs during the sort phase, and it means fewer seeks per run during the merge phase. The product of fewer runs and fewer seeks per run means a substantial reduction in total seeks.

Let's test this conclusion with our 80 000 000-record file, which took about 2 hours, 6 minutes using 10 megabytes of memory. Suppose we are able to obtain 40 megabytes of memory buffer space for our sort. Each of the initial runs would increase from 100 000 records to 400 000 records, resulting in two hundred 400 000-record runs. For the merge phase, the internal buffer space would be divided into 200 buffers, each capable of holding 1/200th of a run, meaning that there would be $200 \times 200 = 40\,000$ seeks. Using the same time estimates that we used for the previous two

cases, the total time for this merge is 16 minutes, 40 seconds, nearly a sevenfold improvement.

Increasing the Number of Dedicated Disk Drives

If we could have a separate read/write head for every run and no other users contending for use of the same read/write heads, there would be no delay due to seek time after the original runs are generated. The primary source of delay would now be rotational delays and transfers, which would occur every time a new block had to be read.

For example, if each run is on a separate, dedicated drive, our 800-way merge calls for only 800 seeks (one seek per run), down from 640 000, cutting the total seek and rotation times from 7040 seconds to 1 second. Of course, we can't configure 800 separate disk drives every time we want to do a sort, but perhaps something short of this is possible. For instance, if we had two disk drives to dedicate to the merge, we could assign one to input and the other to output, so reading and writing could overlap whenever they occurred simultaneously. (This approach takes some clever buffer management, however. We discuss this later in this chapter.)

Increasing the Number of I/O Channels

If there is only one I/O channel, no two transmissions can occur at the same time, and the total transmission time is the one we have computed. But if there is a separate I/O channel for each disk drive, I/O can overlap completely.

For example, if for our 800-way merge there are 800 channels from 800 disk drives, then transmissions can overlap completely. Practically speaking, it is unlikely that 800 channels and 800 disk drives are available, and even if they were, it is unlikely that all transmissions would overlap because all buffers would not need to be refilled at one time. Nevertheless, increasing the number of I/O channels could improve transmission time substantially.

So we see that there are ways to improve performance if we have some control over how our hardware is configured. In those environments in which external sorting occupies a large percentage of computing time, we are likely to have at least some such control. On the other hand, many times we are not able to expand a system specifically to meet sorting needs that we might have. When this is the case, we need to look for algorithmic ways to improve performance, and this is what we do now.

8.5.5 Decreasing the Number of Seeks Using Multiple-Step Merges

One of the hallmarks of a solution to a file structure problem, as opposed to the solution of a mere data structure problem, is the attention given to the enormous difference in cost between accessing information on disk and accessing information in memory. If our merging problem involved only memory operations, the relevant measure of work, or expense, would be the number of *comparisons* required to complete the merge. The *merge pattern* that would minimize the number of comparisons for our sample problem, in which we want to merge 800 runs, would be the 800-way merge considered. Looked at from a point of view that ignores the cost of seeking, this K -way merge has the following desirable characteristics:

- Each record is read only once.
- If a selection tree is used for the comparisons performed in the merging operation, as described in Section 8.3, then the number of comparisons required for a K -way merge of N records (total) is a function of $N \times \log_2 K$.
- Since K is directly proportional to N , this is an $O(N \times \log_2 N)$ operation (measured in numbers of comparisons), which is to say that it is reasonably efficient even as N grows large.

This would all be very good news were we working exclusively in memory, but the very purpose of this *merge sort* procedure is to be able to sort files that are too large to fit into memory. Given the task at hand, the costs associated with disk seeks are orders of magnitude greater than the costs of operations in memory. Consequently, if we can sacrifice the advantages of an 800-way merge and trade them for savings in access time, we may be able to obtain a net gain in performance.

We have seen that one of the keys to reducing seeks is to reduce the number of runs that we have to merge, thereby giving each run a bigger share of available buffer space. In the previous section we accomplished this by adding more memory. Multiple-step merging provides a way for us to apply the same principle without having to buy more memory.

In multiple-step merging, we do not try to merge all runs at one time. Instead, we break the original set of runs into small groups and merge the runs in these groups separately. On each of these smaller merges, more buffer space is available for each run; hence, fewer seeks are required per run. When all of the smaller merges are completed, a second pass merges the new set of merged runs.

It should be clear that this approach will lead to fewer seeks on the first pass, but now there is a second pass. Not only are a number of seeks required for reading and writing on the second pass, but extra transmission time is used in reading and writing all records in the file. Do the advantages of the two-pass approach outweigh these extra costs? Let's revisit the merge step of our 80 million record sort to find out.

Recall that we began with 800 runs of 100 000 records each. Rather than merging all 800 runs at once, we could merge them as, say, 25 sets of 32 runs each, followed by a 25-way merge of the intermediate runs. This scheme is illustrated in Fig. 8.23.

When compared with our original 800-way merge, this approach has the disadvantage of requiring that we read every record twice: once to form the intermediate runs and again to form the final sorted file. But, since each step of the merge is reading from 25 input files at a time, we are able to use larger buffers and avoid a large number of disk seeks. When we analyzed the seeking required for the 800-way merge, disregarding seeking for the output file, we calculated that the 800-way merge involved 640 000 seeks between the input files. Let's perform similar calculations for our multistep merge.

First Merge Step

For each of the 32-way merges of the initial runs, each input buffer can hold $1/32$ run, so we end up making $32 \times 32 = 1024$ seeks. For all 25 of the

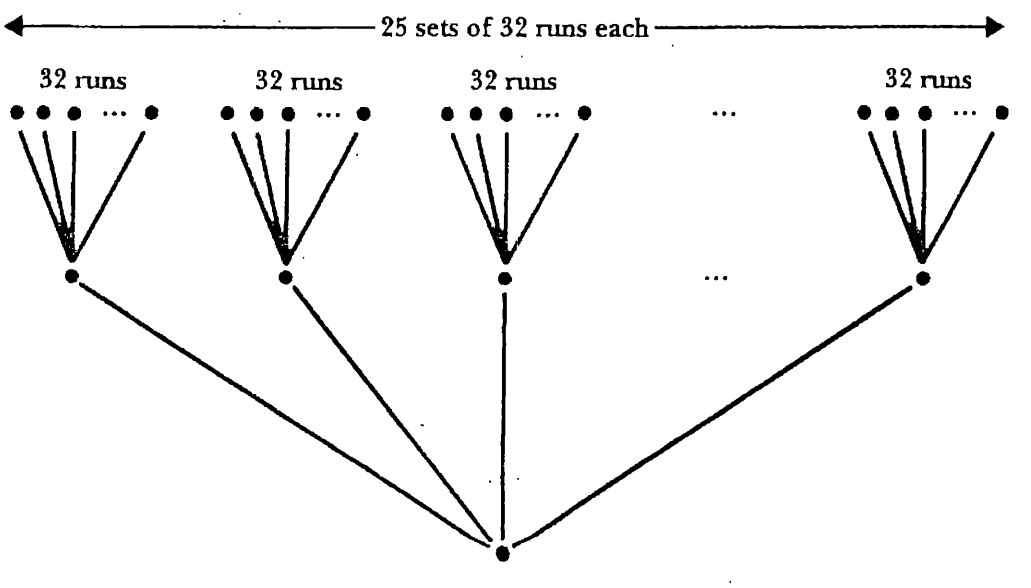


Figure 8.23 Two-step merge of 800 runs.

32-way merges, we make $25 \times 1024 = 25\,600$ seeks. Each of the resulting runs is 3 200 000 records, or 320 megabytes.

Second Merge Step

For each of the 25 final runs, $1/25$ of the total buffer space is allocated, so each input buffer can hold 4000 records, or $1/800$ run. Hence, in this step there are 800 seeks per run, so we end up making $25 \times 800 = 20\,000$ seeks, and

The total number of seeks for the two steps = $25\,600 + 20\,000 = 45\,600$

So, by accepting the cost of processing each record twice, we reduce the number of seeks for reading from 640 000 to 45 600, and we haven't spent a penny for extra memory.

But what about the *total* time for the merge? We save on access times for inputting data, but there are costs. We now have to transmit all of the records four times instead of two, so transmission time increases by 1200 seconds. Also, we write the records twice, rather than once, requiring an extra 40 000 seeks. When we add these extra operations, the total time for the merge is 3782 seconds, or about 1 hour, 3 minutes, compared with 2 hours, 25 minutes for the single-step merge. These results are summarized in Table 8.3.

Once more, note that the essence of what we have done is to find a way to increase the available buffer space for each run. We trade extra passes over the data for a dramatic decrease in random accesses. In this case the trade is certainly a profitable one.

Table 8.3 Time estimates for two-step merge sort of 8000-megabyte file, assuming use of the Seagate Cheetah 9 disk drive described in Table 3.1. The total time is 27 minutes.

	Number of seeks	Amount transferred (megabytes)	Seek + rotation time (seconds)	Transfer time (seconds)	Total time (seconds)
1 st Merge: reading	25 600	8000	282	600	882
1 st Merge: writing	40 000	8000	440	600	1040
2 nd Merge: reading	20 000	8000	220	600	820
2 nd Merge: writing	40 000	8000	440	600	1040
Totals	125 600	32 000	1382	2400	3782

If we can achieve such an improvement with a two-step merge, can we do even better with three steps? Perhaps, but it is important to note in Table 8.3 that we have reduced total seek and rotation times to the point where transmission times are more expensive. Since a three-step merge would require yet another pass over the file, we have reached a point of diminishing returns.

We also could have chosen to distribute our initial runs differently. How would the merge perform if we did 400 two-way merges, followed by one 400-way merge, for instance? A rigorous analysis of the trade-offs between seek and rotation time and transmission time, accounting for different buffer sizes, is beyond the scope of our treatment of the subject.⁵ Our goal is simply to establish the importance of the interacting roles of the major costs in performing merge sorts: seek and rotation time, transmission time, buffer size, and number of runs. In the next section we focus on the pivotal role of the last of these—the number of runs.

8.5.6 Increasing Run Lengths Using Replacement Selection

What would happen if we could somehow increase the size of the initial runs? Consider, for example, our earlier sort of 80 000 000 records in which each record was 100 bytes. Our initial runs were limited to approximately 100 000 records because the memory work area was limited to 10 megabytes. Suppose we are somehow able to create runs of twice this length, containing 200 000 records each. Then, rather than needing to perform an 800-way merge, we need to do only a 400-way merge. The available memory is divided into 400 buffers, each holding 1/800th of a run. Hence, the number of seeks required per run is 800, and the total number of seeks is

$$800 \text{ seeks/run} \times 400 \text{ runs} = 320\,000 \text{ seeks,}$$

half the number required for the 800-way merge of 100 000-byte runs.

In general, if we can somehow increase the size of the initial runs, we decrease the amount of work required during the merge step of the sorting process. A longer initial run means fewer total runs, which means a lower-order merge, which means bigger buffers, which means fewer seeks. But how, short of buying twice as much memory for the computer, can we create initial runs that are twice as large as the number of records that we

5. For more rigorous and detailed analyses of these issues, consult the references cited at the end of this chapter, especially Knuth (1998) and Salzberg (1988, 1990).

can hold in memory? The answer, once again, involves sacrificing some efficiency in our in-memory operations in return for decreasing the amount of work to be done on disk. In particular, the answer involves the use of an algorithm known as *replacement selection*.

Replacement selection is based on the idea of always *selecting* the key from memory that has the lowest value, outputting that key, and then *replacing* it with a new key from the input list. Replacement selection can be implemented as follows:

1. Read a collection of records and sort them using heapsort. This creates a heap of sorted values. Call this heap the *primary heap*.
2. Instead of writing the entire primary heap in sorted order (as we do in a normal heapsort), write only the record whose key has the lowest value.
3. Bring in a new record and compare the value of its key with that of the key that has just been output.
 - a. If the new key value is higher, insert the new record into its proper place in the primary heap along with the other records that are being selected for output. (This makes the new record part of the run that is being created, which means that the run being formed will be larger than the number of keys that can be held in memory at one time.)
 - b. If the new record's key value is lower, place the record in a *secondary heap* of records with key values lower than those already written. (It cannot be put into the primary heap because it cannot be included in the run that is being created.)
4. Repeat step 3 as long as there are records left in the primary heap and there are records to be read. When the primary heap is empty, make the secondary heap into the primary heap, and repeat steps 2 and 3.

To see how this works, let's begin with a simple example, using an input list of only six keys and a memory work area that can hold only three keys. As Fig. 8.24 illustrates, we begin by reading into memory the three keys that fit there and use heapsort to sort them. We select the key with the minimum value, which happens to be 5 in this example, and output that key. We now have room in the heap for another key, so we read one from the input list. The new key, which has a value of 12, now becomes a member of the set of keys to be sorted into the output run. In fact, because it is smaller than the other keys in memory, 12 is the next key that is output. A new key is read into its place, and the process continues. When

Input:
21, 67, 12, 5, 47, 16

↑
Front of input string

Remaining input	Memory ($P = 3$)	Output run
21, 67, 12	5 47 16	—
21, 67	12 47 16	5
21	67 47 16	12, 5
—	67 47 21	16, 12, 5
—	67 47 —	21, 16, 12, 5
—	67 — —	47, 21, 16, 12, 5
—	— — —	67, 47, 21, 16, 12, 5

Figure 8.24 Example of the principle underlying replacement selection.

the process is complete, it produces a sorted list of six keys while using only three memory locations.

In this example the entire file is created using only one heap, but what happens if the fourth key in the input list is 2 rather than 12? This key arrives in memory too late to be output into its proper position relative to the other keys: the 5 has already been written to the output list. Step 3b in the algorithm handles this case by placing such values in a second heap, to be included in the next run. Figure 8.25 illustrates how this process works. During the first run, when keys that are too small to be included in the primary heap are brought in, we mark them with parentheses, indicating that they have to be held for the second run.

It is interesting to use this example to compare the action of replacement selection to the procedure we have been using up to this point, namely that of reading keys into memory, sorting them, and outputting a run that is the size of the memory space. In this example our input list contains thirteen keys. A series of successive memory sorts, given only three memory locations, results in five runs. The replacement selection procedure results in only two runs. Since the disk accesses during a multi-way merge can be a major expense, replacement selection's ability to create longer, and therefore fewer, runs can be an important advantage.

Two questions emerge at this point:

1. Given P locations in memory, how long a run can we expect replacement selection to produce, on the average?
2. What are the costs of using replacement selection?

Input:

33, 18, 24, 58, 14, 17, 7, 21, 67, 12, 5, 47, 16

↑ Front of input string

Remaining input	Memory ($P = 3$)	Output run
33, 18, 24, 58, 14, 17, 7, 21, 67, 12	5 47 16	-
33, 18, 24, 58, 14, 17, 7, 21, 67	12 47 16	5
33, 18, 24, 58, 14, 17, 7, 21	67 47 16	12, 5
33, 18, 24, 58, 14, 17, 7	67 47 21	16, 12, 5
33, 18, 24, 58, 14, 17	67 47 (7)	21, 16, 12, 5
33, 18, 24, 58, 14	67 (17) (7)	47, 21, 16, 12, 5
33, 18, 24, 58	(14) (17) (7)	67, 47, 21, 16, 12, 5
First run complete; start building the second		
33, 18, 24, 58	14 17 7	-
33, 18, 24	14 17 58	7
33, 18	24 17 58	14, 7
33	24 18 58	17, 14, 7
-	24 33 58	18, 17, 14, 7
-	- 33 58	24, 18, 17, 14, 7
-	- - 58	33, 24, 18, 17, 14, 7
		58, 33, 24, 18, 17, 14, 7

Figure 8.25 Step-by-step operation of replacement selection working to form two sorted runs.

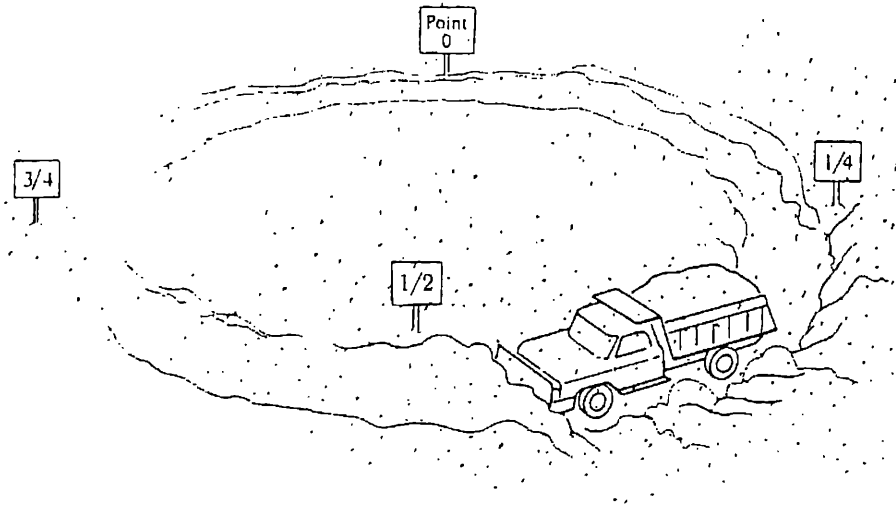
Average Run Length for Replacement Selection

The answer to the first question is that, on the average, we can expect a run length of $2P$, given P memory locations. Knuth⁶ provides an excellent description of an intuitive argument for why this is so:

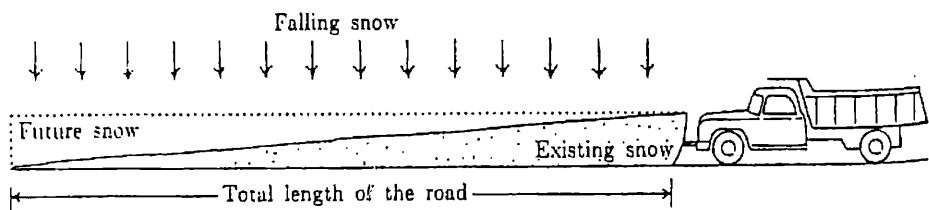
A clever way to show that $2P$ is indeed the expected run length was discovered by E. F. Moore, who compared the situation to a snowplow on a circular track [U.S. Patent 2983904 (1961), Cols. 3–4]. Consider the situation shown [page 336]; flakes of snow are falling uniformly on a circular road, and a lone snowplow is continually clearing the snow. Once the snow has been plowed off the road, it disappears from the system. Points on the road may be designated by real numbers x , $0 \leq x < 1$; a flake of snow falling at position x represents an input record whose key is x ,

6. From Donald Knuth, *The Art of Computer Programming*, vol. 3 1973, Addison-Wesley, Reading, Mass. Pages 254–55 and Figs. 64 and 65. Reprinted with permission.

and the snowplow represents the output of replacement selection. The ground speed of the snowplow is inversely proportional to the height of the snow that it encounters, and the situation is perfectly balanced so that the total amount of snow on the road at all times is exactly P . A new run is formed in the output whenever the plow passes point 0.



After this system has been in operation for a while, it is intuitively clear that it will approach a stable situation in which the snowplow runs at constant speed (because of the circular symmetry of the track). This means that the snow is at constant height when it meets the plow, and the height drops off linearly in front of the plow as shown [below]. It follows that the volume of snow removed in one revolution (namely the run length) is twice the amount present at any one time (namely P).



So, given a random ordering of keys, we can expect replacement selection to form runs that contain about twice as many records as we can hold in memory at one time. It follows that replacement selection creates half as many runs as a series of memory sorts of memory contents, assuming that the replacement selection and the memory sort have access to the same amount of memory. (As we see in a moment, the replacement selection does, in fact, have to make do with less memory than the memory sort.)

It is often possible to create runs that are substantially longer than $2P$. In many applications, the order of the records is *not* wholly random; the keys are often already partially in ascending order. In these cases replacement selection can produce runs that, on the average, exceed $2P$. (Consider what would happen if the input list is already sorted.) Replacement selection becomes an especially valuable tool for such partially ordered input files.

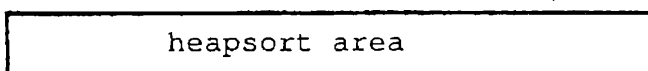
The Costs of Using Replacement Selection

Unfortunately, the no-free-lunch rule applies to replacement selection, as it does to so many other areas of file structure design. In the worked-by-hand examples we have looked at up to this point, we have been inputting records into memory one at a time. We know, in fact, that the cost of seeking for every single input record is prohibitive. Instead, we want to buffer the input, which means, in turn, that we are not able to use *all* of the memory for the operation of replacement selection. Some of it has to be used for input and output buffering. This cost, and the affect it has on available space for sorting, is illustrated in Fig. 8.26.

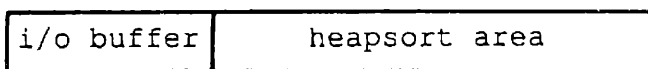
To see the effects of this need for buffering during the replacement selection step, let's return to our example in which we sort 80 000 000 records, given a memory area that can hold 100 000 records.

For the memory sorting methods such as heapsort, which simply read records into memory until it is full, we can perform sequential reads of 100 000 records at a time, until 800 runs have been created. This means that the sort step requires 1600 seeks: 800 for reading and 800 for writing.

For replacement selection we might use an input/output buffer that can hold, for example, 25 000 records; leaving enough space to hold 75 000 records for the replacement selection process. If the I/O buffer holds 2500 records, we can perform sequential reads of 25 000 records at a time, so it



(a) In-RAM sort: all available space used for the sort.



(b) Replacement selection: some of available space is used for I/O.

Figure 8.26 In-memory sort versus replacement selection, in terms of their use of available memory for sorting operation.

takes $80\,000\,000/25\,000 = 3200$ seeks to access all records in the file. This means that the sort step for replacement selection requires 6400 seeks: 3200 for reading and 3200 for writing.

If the records occur in a random key sequence, the average run length using replacement selection will be $2 \times 75\,000 = 150\,000$ records, and there will be about $80\,000\,000/150\,000 = 534$ such runs produced. For the merge step we divide the 10 megabytes of memory into 534 buffers, which hold an average of 187.3 records, so we end up making $150\,000/187.3 = 801$ seeks per run, and

$$801 \text{ seeks per run} \times 534 \text{ runs} = 427\,734 \text{ seeks altogether}$$

Table 8.4 compares the access times required to sort the 80 million records using both a memory sort and replacement selection. The table includes our initial 800-way merge and two replacement selection examples. The second replacement selection example, which produces runs of 400 000 records while using only 75 000 record storage locations in memory, assumes that there is already a good deal of sequential ordering within the input records.

It is clear that, given randomly distributed input data, replacement selection can substantially reduce the number of runs formed. Even though replacement selection requires four times as many seeks to form the runs, the reduction in the amount of seeking effort required to merge the runs more than offsets the extra amount of seeking that is required to form the runs. And when the original data is assumed to possess enough order to make the runs 400 000 records long, replacement selection produces less than one-third as many seeks as memory sorting.

8.5.7 Replacement Selection Plus Multistep Merging

While these comparisons highlight the advantages of replacement selection over memory sorting, we would probably not in reality choose the one-step merge patterns shown in Table 8.4. We have seen that two-step merges can result in much better performance than one-step merges. Table 8.5 shows how these same three sorting schemes compare when two-step merges are used. From Table 8.5 (page 340) we see that the total number of seeks is dramatically less in every case than it was for the one-step merges. Clearly, the method used to form runs is not nearly as important as the use of multistep, rather than one-step, merges.

Furthermore, because the number of seeks required for the merge steps is much smaller in all cases, while the number of seeks required to

Table 8.4 Comparison of access times required to sort 80 million records using both memory sort and replacement selection. Merge order is equal to the number of runs formed.

Approach	Number of records per run	Size of runs formed	Number of runs formed	Number of seeks required to form runs	Merge order used	Total number of seeks	Total seek and rotational delay time
	seek to form runs	formed	formed	to form runs	used	of seeks	(hr) (min)
800 memory sorts followed by an 800-way merge	100 000	100 000	800	1600	800	681 600	2 5
Replacement selection followed by 534-way merge (records in random order)	25 000	150 000	534	6400	534	521 134	1 36
Replacement selection followed by 200-way merge (records partially ordered)	25 000	400 000	200	6400	200	206 400	00 38

Table 8.5 Comparison of access times required to sort 80 million records using both memory sort and replacement selection, each followed by a two-step merge.

Approach	Number of records per run	Size of runs formed	Number of runs formed	Merge pattern used	Number of seeks in merge phases	Total number of seeks	Total seek and rotational delay time
							(hr) (min)
800 memory sorts	100 000	100 000	800	25 × 32-way then 25-way	25 600/20 000	127 200	0 24
Replacement selection (records in random order)	25 000	150 000	534	19 × 28-way then 19-way	22 876/15 162	124 438	0 23
Replacement selection (records partially ordered)	25 000	400 000	200	20 × 10-way then 20-way	8 000/16 000	110 400	0 20

form runs remains the same, the latter have a bigger effect *proportionally* on the final total, and the differences between the memory-sort based method and replacement selection are diminished.

The differences between the one-step and two-step merges are exaggerated by the results in Table 8.5 because they don't take into account the amount of time spent transmitting the data. The two-step merges require that we transfer the data between memory and disk two more times than the one-step merges. Table 8.6 shows the results after adding transmission time to our results. The two-step merges are still better, and replacement selection still wins, but the results are less dramatic.

8.5.8 Using Two Disk Drives with Replacement Selection

Interestingly, and fortunately, replacement selection offers an opportunity to save on both transmission and seek times in ways that memory sort methods do not. As usual, this is at a cost, but if sorting time is expensive, it could well be worth the cost.

Suppose we have two disk drives to which we can assign the separate dedicated tasks of reading and writing during replacement selection. One drive, which contains the original file, does only input, and the other does only output. This has two very nice results: (1) it means that input and output can overlap, reducing transmission time by as much as 50 percent; and (2) seeking is virtually eliminated.

If we have two disks at our disposal, we should also configure memory to take advantage of them. We configure memory as follows: we allocate two buffers each for input and output, permitting double buffering, and allocate the rest of memory for forming the selection tree. This arrangement is illustrated in Fig. 8.27.

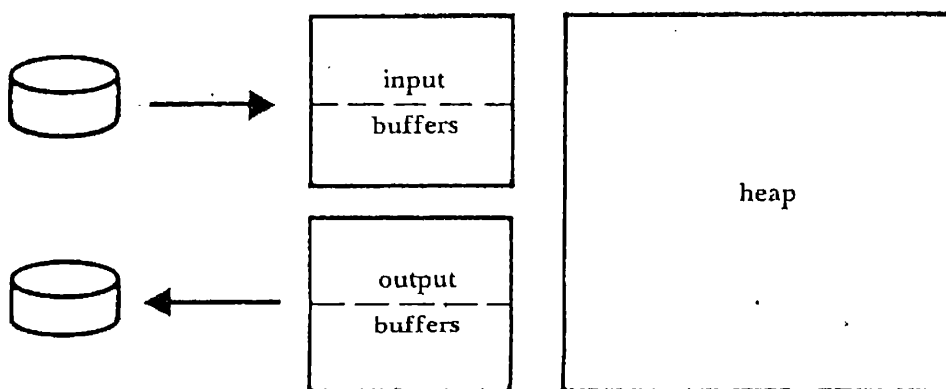


Figure 8.27 Memory organization for replacement selection.

<https://hemanthrajhemu.github.io>

Table 8.6 Comparison of sort merges illustrated in Tables 8.4 and 8.5, taking transmission times into account.

Approach	Number of records	per seek to	form runs	Merge pattern used	Number of seeks for	Sorts and merges	Seek + rotational delay time (min)	Total passes over the file	Total transmission time (min)	Total of seek, rotation, and transmission time (min)
800 memory sorts followed by an 800-way merge	100 000	800-way	100 000	800-way	681/700	125	4	4	40	165
Replacement selection followed by a 534-way merge (records in random order)	25 000	534-way	25 000	534-way	521/134	96	4	4	40	136
Replacement selection followed by a 200-way merge (records partially ordered)	25 000	200-way	25 000	200-way	206/400	38	4	4	40	78
800 memory sorts followed by a two-step merge	100 000	25 × 32-way	127/200	one 25-way	23	6	6	60	83	
Replacement selection followed by a two-step merge (records in random order)	25 000	19 × 28-way	124/438	one 19-way	23	6	6	60	83	
Replacement selection followed by a two-step merge (records partially ordered)	25 000	20 × 10-way	110/400	one 20-way	20	6	6	60	80	

Let's see how the merge sort process might proceed to take advantage of this configuration.

First, the sort phase. We begin by reading enough records to fill up the heap-sized part of memory and form the heap. Next, as we move records from the heap into one of the output buffers, we replace those records with records from one of the input buffers, adjusting the tree in the usual manner. While we empty one input buffer into the tree, we can be filling the other one from the input disk. This permits processing and input to overlap. Similarly, at the same time that we are filling one of the output buffers from the tree, we can transmit the contents of the other to the output disk. In this way, run selection and output can overlap.

During the merge phase, the output disk becomes the input disk, and vice versa. Since the runs are all on the same disk, seeking will occur on the input disk. But output is still sequential, since it goes to a dedicated drive.

Because of the overlapping of so many parts of this procedure, it is difficult to estimate the amount of time the procedure is likely to take. But it should be clear that by substantially reducing seeking and transmission time, we are attacking those parts of the sort merge that are the most costly.

8.5.9 More Drives? More Processors?

If two drives can improve performance, why not three, or four, or more? Isn't it true that the more drives we have to hold runs during the merge phase, the faster we can perform I/O? Up to a point this is true, but of course the number and speed of I/O processors must be sufficient to keep up with the data streaming in and out. And there will also be a point at which I/O becomes so fast that processing can't keep up with it.

But who is to say that we can use only one processor? A decade ago, it would have been farfetched to imagine doing sorting with more than one processor, but now it is very common to be able to dedicate more than one processor to a single job. Possibilities include the following:

- Mainframe computers, many of which spend a great deal of their time sorting, commonly come with two or more processors that can simultaneously work on different parts of the same problem.
- Vector and array processors can be programmed to execute certain kinds of algorithm orders of magnitude faster than scalar processors.
- Massively parallel machines provide thousands, even millions, of processors that can operate independently and at the same time communicate in complex ways with one another.

- Very fast local area networks and communication software make it relatively easy to parcel out different parts of the same process to several different machines.

It is not appropriate, in this text, to cover in detail the implications of these newer architectures for external sorting. But just as the changes over the past decade in the availability and performance of memory and disk storage have altered the way we look at external sorting, we can expect it to change many more times as the current generation of new architectures becomes commonplace.

8.5.10 Effects of Multiprogramming

In our discussions of external sorting on disk we are, of course, making tacit assumptions about the computing environment in which this merging is taking place. We are assuming, for example, that the merge job is running in a dedicated environment (no multiprogramming). If, in fact, the operating system is multiprogrammed, as it normally is, the total time for the I/O might be longer, as our job waits for other jobs to perform their I/O.

On the other hand, one of the reasons for multiprogramming is to allow the operating system to find ways to increase the efficiency of the overall system by overlapping processing and I/O among different jobs. So the system could be performing I/O for our job while it is doing CPU processing on others, and vice versa, diminishing any delays caused by overlap of I/O and CPU processing within our job.

Effects such as these are hard to predict, even when you have much information about your system. Only experimentation can determine what real performance will be like on a busy, multiuser system.

8.5.11 A Conceptual Toolkit for External Sorting

We can now list many tools that can improve external sorting performance. It should be our goal to add these various tools to our conceptual toolkit for designing external sorts and to pull them out and use them whenever they are appropriate. A full listing of our new set of tools would include the following:

- For in-memory sorting, use heapsort for forming the original list of sorted elements in a run. With it and double buffering, we can overlap input and output with internal processing.

- Use as much memory as possible. It makes the runs longer and provides bigger and/or more buffers during the merge phase.
- If the number of initial runs is so large that total seek and rotation time is much greater than total transmission time, use a multistep merge. It increases the amount of transmission time but can decrease the number of seeks enormously.
- Consider using replacement selection for initial run formation, especially if there is a possibility that the runs will be partially ordered.
- Use more than one disk drive and I/O channel so reading and writing can overlap. This is especially true if there are no other users on the system.
- Keep in mind the fundamental elements of external sorting and their relative costs, and look for ways to take advantage of new architectures and systems, such as parallel processing and high-speed local area networks.

8.6 Sorting Files on Tape

There was a time when it was usually faster to perform large external sorts on tape than on disk, but this is much less the case now. Nevertheless, tape is still used in external sorting, and we would be remiss if we did not consider sort merge algorithms designed for tape.

There are a large number of approaches to sorting files on tape. After approximately one hundred pages of closely reasoned discussion of different alternatives for tape sorting, Knuth (1998) summarizes his analysis in the following way:

Theorem A. It is difficult to decide which merge pattern is best in a given situation.

Because of the complexity and number of alternative approaches and because of the way that these alternatives depend so closely on the specific characteristics of the hardware at a particular computer installation, our objective here is merely to communicate some of the fundamental issues associated with tape sorting and merging. For a more comprehensive discussion of specific alternatives, we recommend the work of Knuth (1998) as a starting point.

From a general perspective, the steps involved in sorting on tape resemble those we discussed with regard to sorting on disk:

1. Distribute the unsorted file into sorted *runs*, and
2. Merge the runs into a single sorted file.

Replacement selection is almost always a good choice as a method for creating the initial runs during a tape sort. You will remember that the problem with replacement selection when we are working on disk is that the amount of seeking required during run creation more than offsets the advantage of creating longer runs. This seeking problem disappears when the input is from tape. So, for a tape-to-tape sort, it is almost always advisable to take advantage of the longer runs created by replacement selection.

8.6.1 The Balanced Merge

Given that the question of how to create the initial runs has such a straightforward answer, it is clear that it is in the merging process that we encounter all of the choices and complexities implied by Knuth's tongue-in-cheek theorem. These choices include the question of how to *distribute* the initial runs on tape and questions about the process of merging from this initial distribution. Let's look at some examples to show what we mean.

Suppose we have a file that, after the sort phase, has been divided into ten runs. We look at a number of different methods for merging these runs on tape, assuming that our computer system has four tape drives. Since the initial, unsorted file is read from one of the drives, we have the choice of initially distributing the ten runs on two or three of the other drives. We begin with a method called *two-way balanced merging*, which requires that the initial distribution be on two drives and that at each step of the merge except the last, the output be distributed on two drives. Balanced merging is the simplest tape merging algorithm that we look at; it is also, as you will see, the slowest.

The balanced merge proceeds according to the pattern illustrated in Fig. 8.28.

This balanced merge process is expressed in an alternate, more compact form in Fig. 8.29 (page 348). The numbers inside the table are the run lengths measured in terms of the number of initial runs included in each merged run. For example, in step 1, all the input runs consist of a single initial run. By step 2, the input runs each consist of a pair of initial runs. At the start of step 3, tape drive T1 contains one run consisting of four initial runs followed by a run consisting of two initial runs. This method of illustration more clearly shows the way some of the intermedi-

	Tape	Contains runs				
Step 1	T1	R1	R3	R5	R7	R9
	T2	R2	R4	R6	R8	R10
	T3	—				
	T4	—				
Step 2	T1	—				
	T2	—				
	T3	R1–R2	R5–R6	R9–R10		
	T4	R3–R4	R7–R8			
Step 3	T1	R1–R4	R9–R10			
	T2	R5–R8				
	T3	—				
	T4	—				
Step 4	T1	—				
	T2	—				
	T3	R1–R8				
	T4	R9–R10				
Step 5	T1	R1–R10				
	T2	—				
	T3	—				
	T4	—				

Figure 8.28 Balanced four-tape merge of ten runs.

ate runs combine and grow into runs of lengths 2, 4, and 8, whereas the one run that is copied again and again stays at length 2 until the end. The form used in this illustration is used throughout the following discussions on tape merging.

Since there is no seeking, the cost associated with balanced merging on tape is measured in terms of how much time is spent transmitting the data. In the example, we passed over all of the data four times during the merge phase. In general, given some number of initial runs, how many passes over the data will a two-way balanced merge take? That is, if we start with N runs, how many passes are required to reduce the number of runs to 1? Since each step combines two runs, the number of runs after each

	T1	T2	T3	T4	
Step 1	1 1 1 1 1	1 1 1 1 1	—	—	
Step 2	—	—	2 2 2	2 2	Merge ten runs
Step 3	4 2	4	—	—	Merge ten runs
Step 4	—	—	8	2	Merge ten runs
Step 5	10	—	—	—	

Figure 8.29 Balanced four-tape merge of ten runs expressed in a more compact table notation.

step is half the number for the previous step. If p is the number of passes, then we can express this relationship as

$$\left(\frac{1}{2}\right)^p \cdot N \leq 1$$

from which it can be shown that

$$p = \lceil \log_2 N \rceil$$

In our simple example, $N = 10$, so four passes over the data were required. Recall that for our partially sorted 800-megabyte file there were 200 runs, so $\lceil \log_2 200 \rceil = 8$ passes are required for a balanced merge. If reading and writing overlap perfectly, each pass takes about 11 minutes,⁷ so the total time is 1 hour, 28 minutes. This time is not competitive with our disk-based merges, even when a single disk drive is used. The transmission times far outweigh the savings in seek times.

8.6.2 The K -way Balanced Merge

If we want to improve on this approach, it is clear that we must find ways to reduce the number of passes over the data. A quick look at the formula tells us that we can reduce the number of passes by increasing the order of each merge. Suppose, for instance, that we have 20 tape drives, 10 for input

7. This assumes the 6250 bpi tape used in the examples in Chapter 3. If the transport speed is 200 inches per second, the transmission rate is 1250 kilobytes per second, assuming no blocking. At this rate an 800-megabyte file takes 640 seconds, or 10 minutes 40 seconds to read.

and 10 for output, at each step. Since each step combines 10 runs, the number of runs after each step is one-tenth the number for the previous step. Hence, we have

$$(1/10)^p \cdot N \leq 1$$

and

$$p = \lceil \log_{10} N \rceil$$

In general, a *k-way balanced merge* is one in which the order of the merge at each step (except possibly the last) is *k*. Hence, the number of passes required for a *k-way balanced merge* with *N* initial runs is

$$p = \lceil \log_k N \rceil$$

For a 10-way balanced merge of our 800-megabyte file with 200 runs, $\log_{10} 200 = 3$, so three passes are required. The best estimated time now is reduced to a more respectable 42 minutes. Of course, the cost is quite high: we must keep 20 working tape drives on hand for the merge.

8.6.3 Multiphase Merges

The balanced merging algorithm has the advantage of being very simple; it is easy to write a program to perform this algorithm. Unfortunately, one reason it is simple is that it is “dumb” and cannot take advantage of opportunities to save work. Let’s see how we can improve on it.

We can begin by noting that when we merge the extra run with empty runs in steps 3 and 4, we don’t really accomplish anything. Figure 8.30 shows how we can dramatically reduce the amount of work that has to be done by simply not copying the extra run during step 3. Instead of merging this run with a dummy run, we simply stop tape T3 where it is. Tapes T1 and T2 now each contain a single run made up of four of the initial runs. We rewind all the tapes but T3 and then perform a three-way merge of the runs on tapes T1, T2, and T3, writing the final result on T4. Adding this intelligence to the merging procedure reduces the number of initial runs that must be read and written from forty down to twenty-eight.

The example in Fig. 8.30 clearly indicates that there are ways to improve on the performance of balanced merging. It is important to be able to state, in general terms, what it is about this second merging pattern that saves work:

- We use a higher-order merge. In place of two two-way merges, we use one three-way merge.

	T1	T2	T3	T4	
Step 1	1 1 1 1 1	1 1 1 1 1	—	—	
Step 2	—	—	2 2 2	2 2	Merge ten runs
Step 3	4	4	2	—	Merge eight runs
Step 4	—	—	—	10	Merge ten runs

Figure 8.30 Modification of balanced four-tape merge that does not rewind between steps 2 and 3 to avoid copying runs.

- We extend the merging of runs from one tape over several steps. Specifically, we merge some of the runs from T3 in step 3 and some in step 4. We could say that we merge the runs from T3 in two *phases*.

These ideas, the use of higher-order merge patterns and the merging of runs from a tape in *phases*, are the basis for two well-known approaches to merging called *polyphase merging* and *cascade merging*. In general, these merges share the following characteristics:

- The initial distribution of runs is such that at least the initial merge is a $J-1$ -way merge, where J is the number of available tape drives.
- The distribution of the runs across the tapes is such that the tapes often contain different numbers of runs.

Figure 8.31 illustrates how a polyphase merge can be used to merge ten runs distributed on four tape drives. This merge pattern reduces the number of initial runs that must be read and written from forty (for a balanced two-way merge) to twenty-five. It is easy to see that this reduction is a consequence of the use of several three-way merges in place of two-way merges. It should also be clear that the ability to do these operations as three-way merges is related to the uneven nature of the initial distribution. Consider, for example, what happens if the initial distribution of runs is 4–3–3 rather than 5–3–2. We can perform three three-way merges to open up space on T3, but this also clears all the runs off of T2 and leaves only a single run on T1. Obviously, we are not able to perform another three-way merge as a second step.

Several questions arise at this point:

1. How does one choose an initial distribution that leads readily to an efficient merge pattern?

	T1	T2	T3	T4	
Step 1	1 1 1 1 1	1 1 1	1 1	—	Merge six runs
Step 2	. . 1 1 1	. . 1	—	3 3	Merge five runs
Step 3	. . . 1 1	—	5	. 3	Merge four runs
Step 4 1	4	5	—	Merge ten runs
Step 5	—	—	—	10	

Figure 8.31 Polyphase four-tape merge of ten runs.

2. Are there algorithmic descriptions of the merge patterns, given an initial distribution?
3. Given N runs and J tape drives, is there some way to compute the *optimal* merging performance so we have a yardstick against which to compare the performance of any specific algorithm?

Precise answers to these questions are beyond the scope of this text; in particular, the answer to the last question requires a more mathematical approach to the problem than the one we have taken here. Readers wanting more than an intuitive understanding of how to set up initial distributions should consult Knuth (1998).

8.6.4 Tapes versus Disks for External Sorting

A decade ago 1 megabyte of memory was considered a substantial amount of memory to allocate to any single job, and extra disk drives were very costly. This meant that many of the disk sorting techniques to decrease seeking that we have seen were not available to us or were very limited.

Suppose, for instance, that we want to sort our 8000-megabyte file and there is only 1 megabyte of memory available instead of 10 megabytes. The approach that we used for allocating memory for replacement selection would provide 250 kilobytes for buffering and 750 kilobytes for our selection tree. From this we can expect 5334 runs of 15 000 records each, versus 534 when there is a megabyte of memory. For a one-step merge, this tenfold increase in the number of runs results in a hundredfold increase in the number of seeks. What took three hours with 10 megabytes of memory now takes three hundred hours, just for the seeks! No wonder tapes, which are basically sequential and require no seeking, were preferred.

But now memory is much more readily available. Runs can be longer and fewer, and seeks are much less of a problem. Transmission time is now more important. The best way to decrease transmission time is to reduce the number of passes over the data, and we can do this by increasing the order of the merge. Since disks are random-access devices, very large-order merges can be performed, even if there is only one drive. Tapes, however, are not random-access devices; we need an extra tape drive for every extra run we want to merge. Unless a large number of drives is available, we can perform only low-order merges, and that means large numbers of passes over the data. Disks are better.

8.7 Sort-Merge Packages

Many good utility programs are available for users who need to sort large files. Often the programs have enough intelligence to choose from one of several strategies, depending on the nature of the data to be sorted and the available system configuration. They also often allow users to exert some control (if they want it) over the organization of data and strategies used. Consequently, even if you are using a commercial sort package rather than designing your own sorting procedure, it helps to be familiar with the variety of different ways to design merge sorts. It is especially important to have a good general understanding of the most important factors and trade-offs influencing performance.

8.8 Sorting and Cosequential Processing in Unix

Unix has a number of utilities for performing cosequential processing. It also has sorting routines, but nothing at the level of sophistication that you find in production sort-merge packages. In the following discussion we introduce some of these utilities. For full details, consult the Unix documentation.

8.8.1 Sorting and Merging in Unix

Because Unix is not an environment in which one expects to do frequent sorting of large files of the type we discuss in this chapter, sophisticated

sort-merge packages are not generally available on Unix systems. Still, the sort routines you find in Unix are quick and flexible and quite adequate for the types of applications that are common in a Unix environment. We can divide Unix sorting into two categories: (1) the `sort` command, and (2) callable sorting routines.

The Unix sort Command

The `sort` command has many different options, but the simplest one is to sort the lines in an ASCII file in ascending lexical order. (A line is any sequence of characters ending with the new-line character `.`.) By default, the `sort` utility takes its input file name from the command line and writes the sorted file to standard output. If the file to be sorted is too large to fit in memory, `sort` performs a merge sort. If more than one file is named on the input line, `sort` sorts and merges the files.

As a simple example, suppose we have an ASCII file called `team` with names of members of a basketball team, together with their classes and their scoring averages:

```
Jean Smith Senior 8.8
Chris Mason Junior 9.6
Pat Jones Junior 3.2
Leslie Brown Sophomore 18.2
Pat Jones Freshman 11.4
```

To sort the file, enter

```
$ sort team
Chris Mason Junior 9.6
Jean Smith Senior 8.8
Leslie Brown Sophomore 18.2
Pat Jones Freshman 11.4
Pat Jones Junior 3.2
```

Notice that by default `sort` considers an entire line as the sort key. Hence, of the two players named Pat Jones, the freshman occurs first in the output because “Freshman” is lexically smaller than “Junior.” The assumption that the key is an entire line can be overridden by sorting on specified key fields. For `sort` a key field is assumed to be any sequence of characters delimited by spaces or tabs. You can indicate which key fields to use for sorting by giving their positions:

```
+pos1 [-pos2]
```

where `pos1` tells how many fields to skip before starting the key, and `pos2` tells which field to end with. If `pos2` is omitted, the key extends to the end of the line. Hence, entering

```
$ sort +1 -2 team
```

causes the file *team* to be sorted according to last names. (There is also a form of `pos1` and `pos2` that allows you to specify the character within a field to start a key with.)

The following options, among others, allow you to override the default ASCII ordering used by `sort` :

- d Use “dictionary” ordering: only letters, digits, and blanks are significant in comparisons.
- f “Fold” lowercase letters into uppercase. (This is the canonical form that we defined in Chapter 4.)
- r “Reverse” the sense of comparison: sort in descending ASCII order.

Notice that `sort` sorts lines, and within lines it compares groups of characters delimited by white space. In the language of Chapter 4, records are lines, and fields are groups of characters delimited by white space. This is consistent with the most common Unix view of fields and records within Unix text files.

The `qsort` Library Routine

The Unix library routine `qsort` is a general sorting routine. Given a table of data, `qsort` sorts the elements in the table in place. A table could be the contents of a file, loaded into memory, where the elements of the table are its records. In C, `qsort` is defined as follows:

```
qsort(char *base, int nel, int width, int (*compar())) )
```

The argument `base` is a pointer to the base of the data, `nel` is the number of elements in the table, and `width` is the size of each element. The last argument, `compar()`, is the name of a user-supplied comparison function that `qsort` uses to compare keys. `compar` must have two parameters that are pointers to elements that are to be compared. When `qsort` needs to compare two elements, it passes to `compar` pointers to these elements, and `compar` compares them, returning an integer that is less than, equal to, or greater than zero, depending on whether the first argument is considered less than, equal to, or greater than the second argument. A full explanation of how to

use `qsort` is beyond the scope of this text. Consult the Unix documentation for details.

8.8.2 Cosequential Processing Utilities in Unix

Unix provides a number of utilities for cosequential processing. The `sort` utility, when used to merge files, is one example. In this section we introduce three others: `diff`, `cmp`, and `comm`.

cmp

Suppose you find in your computer that you have two team files, one called `team` and the other called `myteam`. You think that the two files are the same, but you are not sure. You can use the command `cmp` to find out.

`cmp` compares two files. If they differ, it prints the byte and line number where they differ; otherwise it does nothing. If all of one file is identical to the first part of another, it reports that end-of-file was reached on the shorter file before any differences were found.

For example, suppose the file `team` and `myteam` have the following contents:

team

```
Jean Smith Senior 8.8
Chris Mason Junior 9.6
Pat Jones Junior 3.2
Leslie Brown Sophomore 18.2
Pat Jones Freshman 11.4
```

myteam

```
Jean Smith Senior 8.8
Stacy Fox Senior 1.6
Chris Mason Junior 9.6
Pat Jones Junior 5.2
Leslie Brown Sophomore 18.2
Pat Jones Freshman 11.4
```

`cmp` tells you where they differ:

```
$ cmp team myteam
team myteam differ: char 23 line 2
```

Since `cmp` simply compares files on a byte-by-byte basis until it finds a difference, it makes no assumptions about fields or records. It works with both text and nontext files.

diff

`cmp` is useful if you want to know if two files are different, but it doesn't tell you much about how they differ. The command `diff` gives fuller

information. `diff` tells which lines must be changed in two files to bring them into agreement. For example:

```
$ diff team myteam
1a2
> Stacy Fox Senior 1.6
3c4
< Pat Jones Junior 3.2
---
> Pat Jones Junior 5.2
```

The `1a2` indicates that after line 1 in the first file, we need to *add* line 2 from the second file to make them agree. This is followed by the line from the second file that would need to be added. The `3c4` indicates that we need to *change* line 3 in the first file to make it look like line 4 in the second file. This is followed by a listing of the two differing lines, where the leading `<` indicates that the line is from the first file, and the `>` indicates that it is from the second file.

One other indicator that could appear in `diff` output is `d`, meaning that a line in the first file has been *deleted* in the second file. For example, `12d15` means that line 12 in the first file appears to have been deleted from being right after line 15 in the second file. Notice that `diff`, like `sort`, is designed to work with lines of text. It would not work well with non-ASCII text files.

comm

Whereas `diff` tells what is different about two files, `comm` compares two files, which must be ordered in ASCII collating sequence, to see what they have in common. The syntax for `comm` is the following:

```
comm [-123] file1 file2
```

`comm` produces three columns of output. Column 1 lists the lines that are in `file1` only; column 2 lists lines in `file2` only, and column 3 lists lines that are in both files. For example,

```
$ sort team > ts
$ sort myteam > ms
$ comm ts ms
Chris Mason Junior 9.6
Jean Smith Senior 8.8
Leslie Brown Sophomore 18.2
Pat Jones Freshman 11.4
```

Pat Jones Junior 3.2

Pat Jones Junior 5.2

Stacy Fox Senior 1.6

Selecting any of the flags 1, 2, or 3 allows you to print only those columns you are interested in.

The `sort`, `diff`, `comm`, and `cmp` commands (and the `qsort` function) are representative of what is available in Unix for sorting and cosequential processing. As we have said, they have many useful options that we don't cover that you will be interested in reading about.

S U M M A R Y

In the first half of this chapter, we develop a cosequential processing model and apply it to two common problems—updating a general ledger and merge sorting. The model is presented as a class hierarchy, using virtual methods to tailor the model to particular types of lists. In the second half of the chapter we identify the most important factors influencing performance in merge-sorting operations and suggest some strategies for achieving good performance.

The cosequential processing model can be applied to problems that involve operations such as matching and merging (and combinations of these) on two or more sorted input files. We begin the chapter by illustrating the use of the model to perform a simple match of the elements common to two lists and a merge of two lists. The procedures we develop to perform these two operations embody all the basic elements of the model.

In its most complete form, the model depends on certain assumptions about the data in the input files. We enumerate these assumptions in our formal description of the model. Given these assumptions, we can describe the processing components of the model and define pure virtual functions that represent those components.

The real value of the cosequential model is that it can be adapted to more substantial problems than simple matches or merges by extending the class hierarchy. We illustrate this by using the model to design a general ledger accounting program.

All of our early sample applications of the model involve only two input files. We next adapt the model to a multiway merge to show how the model might be extended to deal with more than two input lists. The problem of finding the minimum key value during each pass through the

main loop becomes more complex as the number of input files increases. Its solution involves replacing the three-way selection statement with either a multiway selection or a procedure that keeps current keys in a list structure that can be processed more conveniently.

We see that the application of the model to k -way merging performs well for small values of k , but that for values of k greater than 8 or so, it is more efficient to find the minimum key value by means of a selection tree.

After discussing multiway merging, we shift our attention to a problem that we encountered in a previous chapter—how to sort large files. We begin with files that are small enough to fit into memory and introduce an efficient sorting algorithm, *heapsort*, which makes it possible to overlap I/O with the sorting process.

The generally accepted solution when a file is too large for in-memory sorts is some form of *merge sort*. A merge sort involves two steps:

1. Break the file into two or more sorted subfiles, or runs, using internal sorting methods; and
2. Merge the runs.

Ideally, we would like to keep every run in a separate file so we can perform the merge step with one pass through the runs. Unfortunately, practical considerations sometimes make it difficult to do this effectively.

The critical elements when merging many files on disk are seek and rotational delay times and transmission times. These times depend largely on two interrelated factors: the number of different runs being merged and the amount of internal buffer space available to hold parts of the runs. We can reduce seek and rotational delay times in two ways:

- By performing the merge in more than one step; and/or
- By increasing the sizes of the initial sorted runs.

In both cases, the order of each merge step can be reduced, increasing the sizes of the internal buffers and allowing more data to be processed per seek.

Looking at the first alternative, we see how performing the merge in several steps can decrease the number of seeks dramatically, though it also means that we need to read through the data more than once (increasing total data transmission time).

The second alternative is realized through use of an algorithm called *replacement selection*. Replacement selection, which can be implemented using the selection tree mentioned earlier, involves selecting from memory the key that has the lowest value, outputting that key, and replacing it with a new key from the input list.

With randomly organized files, replacement selection can be expected to produce runs twice as long as the number of internal storage locations available for performing the algorithms. Although this represents a major step toward decreasing the number of runs that need to be merged, it carries an additional cost. The need for a large buffer for performing the replacement selection operation leaves relatively little space for the I/O buffer, which means that many more seeks are involved in forming the runs than are needed when the sort step uses an in-memory sort. If we compare the total number of seeks required by the two different approaches, we find that replacement selection can require more seeks; it performs substantially better only when there is a great deal of order in the initial file.

Next we turn our attention to file sorting on tapes. Since file I/O with tapes does not involve seeking, the problems and solutions associated with tape sorting can differ from those associated with disk sorting, although the fundamental goal of working with fewer, longer runs remains. With tape sorting, the primary measure of performance is the number of times each record must be transmitted. (Other factors, such as tape rewind time, can also be important, but we do not consider them here.)

Since tapes do not require seeking, replacement selection is almost always a good choice for creating initial runs. As the number of drives available to hold run files is limited, the next question is how to distribute the files on the tapes. In most cases, it is necessary to put several runs on each of several tapes, reserving one or more other tapes for the results. This generally leads to merges of several steps, with the total number of runs being decreased after each merge step. Two approaches to doing this are *balanced merges* and *multiphase merges*. In a k -way balanced merge, all input tapes contain approximately the same number of runs, there are the same number of output tapes as there are input tapes, and the input tapes are read through entirely during each step. The number of runs is decreased by a factor of k after each step.

A multiphase merge (such as a *polyphase merge* or a *cascade merge*) requires that the runs initially be distributed unevenly among all but one of the available tapes. This increases the order of the merge and as a result can decrease the number of times each record has to be read. It turns out that the initial distribution of runs among the first set of input tapes has a major effect on the number of times each record has to be read.

Next, we discuss briefly the existence of sort-merge utilities, which are available on most large systems and can be very flexible and effective. We conclude the chapter with a listing of Unix utilities used for sorting and cosequential processing.

KEY TERMS

- Balanced merge.** A multistep merging technique that uses the same number of input devices as output devices. A two-way balanced merge uses two input tapes, each with approximately the same number of runs on it, and produces two output tapes, each with approximately half as many runs as the input tapes. A balanced merge is suitable for merge sorting with tapes, though it is not generally the best method (see *multiphase merging*).
- cmp.** A Unix utility for determining whether two files are identical. Given two files, it reports the first byte where the two files differ, if they differ.
- comm.** A Unix utility for determining which lines two files have in common. Given two files, it reports the lines they have in common, the lines that are in the first file and not in the second, and the lines that are in the second file and not in the first.
- Cosequential operations.** Operations applied to problems that involve the performance of union, intersection, and more complex set operations on two or more sorted input files to produce one or more output files built from some combination of the elements of the input files. Cosequential operations commonly occur in matching, merging, and file-updating problems.
- diff.** A Unix utility for determining all the lines that differ between two files. It reports the lines that need to be added to the first file to make it like the second, the lines that need to be deleted from the second file to make it like the first, and the lines that need to be changed in the first file to make it like the second.
- Heapsort.** A sorting algorithm especially well suited for sorting large files that fit in memory because its execution can overlap with I/O. A variation of heapsort is used to obtain longer runs in the replacement selection algorithm.
- HighValue.** A value used in the cosequential model that is greater than any possible key value. By assigning HighValue as the current key value for files for which an end-of-file condition has been encountered, extra logic for dealing with end-of-file conditions can be simplified.
- k-way merge.** A merge in which k input files are merged to produce one output file.
- LowValue.** A value used in the cosequential model that is less than any possible key value. By assigning LowValue as the previous key value

during initialization, the need for certain other special start-up code is eliminated.

Match. The process of forming a sorted output file consisting of all the elements common to two or more sorted input files.

Merge. The process of forming a sorted output file that consists of the union of the elements from two or more sorted input files.

Multiphase merge. A multistep tape merge in which the initial distribution of runs is such that at least the initial merge is a $J-1$ -way merge (J is the number of available tape drives) and in which the distribution of runs across the tapes is such that the merge performs efficiently at every step. (See *polyphase merge*.)

Multistep merge. A merge in which not all runs are merged in one step. Rather, several sets of runs are merged separately, each set producing one long run consisting of the records from all of its runs. These new, longer sets are then merged, either all together or in several sets. After each step, the number of runs is decreased and the length of the runs is increased. The output of the final step is a single run consisting of the entire file. (Be careful not to confuse our use of the term *multistep merge* with *multiphase merge*.) Although a multistep merge is theoretically more time-consuming than a single-step merge, it can involve much less seeking when performed on a disk, and it may be the only reasonable way to perform a merge on tape if the number of tape drives is limited.

Order of a merge. The number of different files, or runs, being merged. For example, 100 is the order of a 100-way merge.

Polyphase merge. A multiphase merge in which, ideally, the merge order is maximized at every step.

qsort. A general-purpose Unix library routine for sorting files that employs a user-defined comparison function.

Replacement selection. A method of creating initial runs based on the idea of always *selecting* from memory the record whose key has the lowest value, outputting that record, and then *replacing* it in memory with a new record from the input list. When new records are brought in with keys that are greater than those of the most recently output records, they eventually become part of the run being created. When new records have keys that are less than those of the most recently output records, they are held over for the next run. Replacement selection generally produces runs that are substantially longer than runs

that can be created by in-memory sorts and hence can help improve performance in merge sorting. When using replacement selection with merge sorts on disk, however, one must be careful that the extra seeking required for replacement selection does not outweigh the benefits of having longer runs to merge.

Run. A sorted subset of a file resulting from the sort step of a sort merge or one of the steps of a multistep merge.

Selection tree. A binary tree in which each higher-level node represents the winner of the comparison between the two descendent keys. The minimum (or maximum) value in a selection tree is always at the root node, making the selection tree a good data structure for merging several lists. It is also a key structure in replacement selection algorithms, which can be used for producing long runs for merge sorts. (*Tournament sort*, an internal sort, is also based on the use of a selection tree.)

Sequence checking. Checking that records in a file are in the expected order. It is recommended that all files used in a cosequential operation be sequence checked.

sort. A Unix utility for sorting and merging files.

Synchronization loop. The main loop in the cosequential processing model. A primary feature of the model is to do all synchronization within a single loop rather than in multiple nested loops. A second objective is to keep the main synchronization loop as simple as possible. This is done by restricting the operations that occur within the loop to those that involve current keys and by relegating as much special logic as possible (such as error checking and end-of-file checking) to subprocedures.

Theorem A (Knuth). It is difficult to decide which merge pattern is best in a given situation.

FURTHER READINGS

The subject matter treated in this chapter can be divided into two separate topics: the presentation of a model for cosequential processing and discussion of external merging procedures on tape and disk. Although most file processing texts discuss cosequential processing, they usually do it in the context of specific applications, rather than presenting a general model

that can be adapted to a variety of applications. We found this useful and flexible model through Dr. James VanDoren who developed this form of the model himself for presentation in the file structures course that he teaches. We are not aware of any discussion of the cosequential model elsewhere in the literature.

Quite a bit of work has been done toward developing simple and effective algorithms to do sequential file updating, which is an important instance of cosequential processing. The results deal with some of the same problems the cosequential model deals with, and some of the solutions are similar. See Levy (1982) and Dwyer (1981) for more.

Unlike cosequential processing, external sorting is a topic that is covered widely in the literature. The most complete discussion of the subject, by far, is in Knuth (1998). Students interested in the topic of external sorting must, at some point, familiarize themselves with Knuth's definitive summary of the subject. Knuth also describes replacement selection, as evidenced by our quoting from his book in this chapter.

Salzberg (1990) describes an approach to external sorting that takes advantage of replacement selection, parallelism, distributed computing, and large amounts of memory. Cormen, Leiserson, and Rivest (1990) and Loomis (1989) also have chapters on external sorting.

EXERCISES

1. Consider the cosequential `Merge2Lists` method of Fig. 8.5 and the supporting methods of class `CosequentialProcess` in Appendix H. Comment on how they handle the following initial conditions. If they do not correctly handle a situation, indicate how they might be altered to do so.
 - a. List 1 empty and List 2 not empty
 - b. List 1 not empty and List 2 empty
 - c. List 1 empty and List 2 empty
2. Section 8.3.1 includes the body of a loop for doing a k -way merge, assuming that there are no duplicate names. If duplicate names are allowed, one could add to the procedure a facility for keeping a list of subscripts of duplicate lowest names. Modify the body of the loop to implement this. Describe the changes required to the supporting methods.

3. In Section 8.3, two methods are presented for choosing the lowest of K keys at each step in a K -way merge: a linear search and use of a selection tree. Compare the performances of the two approaches in terms of numbers of comparisons for $K = 2, 4, 8, 16, 32,$ and 100 . Why do you think the linear approach is recommended for values of K less than 8?
4. Suppose you have 80 megabytes of memory available for sorting the 8 000 000-record file described in Section 8.5.
 - a. How long does it take to sort the file using the merge-sort algorithm described in Section 8.5.1?
 - b. How long does it take to sort the file using the keysort algorithm described in Chapter 6?
 - c. Why will keysort not work if there are ten megabytes of memory available for the sorting phase?
5. How much seek time is required to perform a one-step merge such as the one described in Section 8.5 if the time for an average seek is 10 msec and the amount of available internal buffer space is 5000 K? 1000 K?
6. Performance in sorting is often measured in terms of the number of comparisons. Explain why the number of comparisons is not adequate for measuring performance in sorting large files.
7. In our computations involving the merge sorts, we made the simplifying assumption that only one seek and one rotational delay are required for any single sequential access. If this were not the case, a great deal more time would be required to perform I/O. For example, for the 800-megabyte file used in the example in Section 8.5.1, for the input step of the sort phase (“reading all records into memory for sorting and forming runs”), each individual run could require many accesses. Now let’s assume that the extent size for our hypothetical drive is 80 000 bytes (approximately one track) and that all files are stored in track-sized blocks that must be accessed separately (one seek and one rotational delay per block).
 - a. How many seeks does step 1 now require?
 - b. How long do steps 1, 2, 3, and 4 now take?
 - c. How does increasing the file size by a factor of 10 now affect the total time required for the merge sort?
8. Derive two formulas for the number of seeks required to perform the merge step of a one-step k -way sort merge of a file with r records

divided into k runs, where the amount of available memory is equivalent to M records. If an internal sort is used for the sort phase, you can assume that the length of each run is M , but if replacement selection is used, you can assume that the length of each run is about $2M$. Why?

9. Assume a quiet system with four separately addressable disk drives, each of which is able to hold several gigabytes. Assume that the 800-megabyte file described in Section 8.5 is already on one of the drives. Design a sorting procedure for this sample file that uses the separate drives to minimize the amount of seeking required. Assume that the final sorted file is written off to tape and that buffering for this tape output is handled invisibly by the operating system. Is there any advantage to be gained by using replacement selection?
10. Use replacement selection to produce runs from the following files, assuming $P = 4$.
 - a. 2329517955413513318241147
 - b. 3591117182324293341475155
 - c. 5551474133292423181711953
11. Suppose you have a disk drive that has 10 read/write heads per surface, so 10 cylinders may be accessed at any one time without having to move the actuator arm. If you could control the physical organization of runs stored on disk, how might you be able to exploit this arrangement in performing a sort merge?
12. Assume we need to merge 14 runs on four tape drives. Develop merge patterns starting from each of these initial distributions:
 - a. 8—4—2
 - b. 7—4—3
 - c. 6—5—3
 - d. 5—5—4.
13. A four-tape polyphase merge is to be performed to sort the list 24 36 13 25 16 45 29 38 23 50 22 19 43 30 11 27 48. The original list is on tape 4. Initial runs are of length 1. After initial sorting, tapes 1, 2, and 3 contain the following runs (a slash separates runs):

Tape 1: 24 / 36 / 13 / 25

Tape 2: 16 / 45 / 29 / 38 / 23 / 50 Tape 3: 22 / 19 / 43 / 30 / 11 / 27 / 47

 - a. Show the contents of tape 4 after one merge phase.
 - b. Show the contents of all four tapes after the second and fourth phases.

- c. Comment on the appropriateness of the original 4—6—7 distribution for performing a polyphase merge.
14. Obtain a copy of the manual for one or more commercially available sort-merge packages. Identify the different kinds of choices available to users of the packages. Relate the options to the performance issues discussed in this chapter.
15. A join operation matches two files by matching field values in the two files. In the ledger example, a join could be used to match master and transaction records that have the same account numbers. The ledger posting operation could be implemented with a sorted ledger file and an indexed, entry-sequenced transaction file by reading a master record and then using the index to find all corresponding transaction records.

Compare the speed of this join operation with the cosequential processing method of this chapter. Don't forget to include the cost of sorting the transaction file.

PROGRAMMING EXERCISES

16. Modify method `LedgerProcess::ProcessEndMaster` so it updates the ledger file with the new account balances for the month.
17. Implement the k -way merge in class `CosequentialProcessing` using an object of class `Heap` to perform the merge selection.
18. Implement a k -way match in class `CosequentialProcessing`.
19. Implement the sort merge operation using class `Heap` to perform replacement selection to create the initial sorted runs and class `CosequentialProcessing` to perform the merge phases.

PROGRAMMING PROJECT

This is the sixth part of the programming project. We develop applications that produce student transcripts and student grade reports from information contained in files produced by the programming project of Chapter 4.

20. Use class `CosequentialProcesses` and `MasterTransactionProcess` to develop an application that produces student transcripts. For each student record (master) print the student information and a list of all courses (transaction) taken by the student. As input, use a file of student records sorted by student identifier and a file of course registration records sorted by student identifiers.
21. Use class `CosequentialProcesses` and `MasterTransactionProcess` to develop an application that produces student grade reports. As input, use a file of student records sorted by student identifier and a file of course registrations with grades for a single semester.

The next part of the programming project is in Chapter 9.

Multilevel Indexing and B-Trees

CHAPTER OBJECTIVES

- ❖ Place the development of B-trees in the historical context of the problems they were designed to solve.
- ❖ Look briefly at other tree structures that might be used on secondary storage, such as paged AVL trees.
- ❖ Introduce multirecord and multilevel indexes and evaluate the speed of the search operation.
- ❖ Provide an understanding of the important properties possessed by B-trees and show how these properties are especially well suited to secondary storage applications.
- ❖ Present the object-oriented design of B-trees
 - Define class BTreeNode, the in-memory representation of the nodes of B-trees.
 - Define class BTree, the full representation of B-trees including all operations.
- ❖ Explain the implementation of the fundamental operations on B-trees.
- ❖ Introduce the notion of page buffering and virtual B-trees.
- ❖ Describe variations of the fundamental B-tree algorithms, such as those used to build B* trees and B-trees with variable-length records.*

CHAPTER OUTLINE

- 9.1 Introduction: The Invention of the B-Tree
- 9.2 Statement of the Problem
- 9.3 Indexing with Binary Search Trees
 - 9.3.1 AVL Trees
 - 9.3.2 Paged Binary Trees
 - 9.3.3 Problems with Paged Trees
- 9.4 Multilevel Indexing: A Better Approach to Tree Indexes
- 9.5 B-Trees: Working up from the Bottom
- 9.6 Example of Creating a B-Tree
- 9.7 An Object-Oriented Representation of B-Trees
 - 9.7.1 Class BTreeNode: Representing B-Tree Nodes in Memory
 - 9.7.2 Class BTree: Supporting Files of B-Tree Nodes
- 9.8 B-Tree Methods Search, Insert, and Others
 - 9.8.1 Searching
 - 9.8.2 Insertion
 - 9.8.3 Create, Open, and Close
 - 9.8.4 Testing the B-Tree
- 9.9 B-Tree Nomenclature
- 9.10 Formal Definition of B-Tree Properties
- 9.11 Worst-Case Search Depth
- 9.12 Deletion, Merging, and Redistribution
 - 9.12.1 Redistribution
- 9.13 Redistribution During Insertion: A Way to Improve Storage Utilization
- 9.14 B* Trees
- 9.15 Buffering of Pages: Virtual B-Trees
 - 9.15.1 LRU Replacement
 - 9.15.2 Replacement Based on Page Height
 - 9.15.3 Importance of Virtual B-Trees
- 9.16 Variable-Length Records and Keys

9.1 Introduction: The Invention of the B-Tree

Computer science is a young discipline. As evidence of this youth, consider that at the start of 1970, after astronauts had twice traveled to the moon, B-trees did not yet exist. Today, twenty-seven years later, it is hard to think of a major, general-purpose file system that is not built around a B-tree design.

Douglas Comer, in his excellent survey article, “The Ubiquitous B-Tree” (1979), recounts the competition among computer manufacturers and independent research groups in the late 1960s. The goal was the discovery of a general method for storing and retrieving data in large file systems that would provide rapid access to the data with minimal overhead cost. Among the competitors were R. Bayer and E. McCreight, who were working for Boeing Corporation. In 1972 they published an article, “Organization and Maintenance of Large Ordered Indexes,” which announced B-trees to the world. By 1979, when Comer published his survey article, B-trees had already become so widely used that Comer was able to state that “the B-tree is, *de facto*, the standard organization for indexes in a database system.”

We have reprinted the first few paragraphs of the 1972 Bayer and McCreight article¹ because it so concisely describes the facets of the problem that B-trees were designed to solve: how to access and efficiently maintain an index that is too large to hold in memory. You will remember that this is the same problem that is left unresolved in Chapter 7, on simple index structures. It will be clear as you read Bayer and McCreight’s introduction that their work goes straight to the heart of the issues we raised in the indexing chapter.

In this paper we consider the problem of organizing and maintaining an index for a dynamically changing random access file. By an *index* we mean a collection of index elements which are pairs (x, a) of fixed size physically adjacent data items, namely a key x and some associated information a . The key x identifies a unique element in the index, the associated information is typically a pointer to a record or a collection of records in a random access file. For this paper the associated information is of no further interest.

We assume that the index itself is so voluminous that only rather small parts of it can be kept in main store at one time. Thus the bulk of the index must be kept on some backup store. The class of backup stores considered are *pseudo random access devices* which have rather long access or wait time—as opposed to a true random access device like core store—and a rather high data rate once the transmission of physically sequential data has been initiated. Typical pseudo random access devices are: fixed and moving head disks, drums, and data cells.

Since the data file itself changes, it must be possible not only to search the index and to retrieve elements, but also to delete and to insert

1. From *Acta-Informatica*, 1:173–189, ©1972, Springer Verlag, New York. Reprinted with permission.

keys—more accurately index elements—economically. The index organization described in this paper allows retrieval, insertion, and deletion of keys in time proportional to $\log_k I$ or better, where I is the size of the index, and k is a device dependent natural number which describes the page size such that the performance of the maintenance and retrieval scheme becomes near optimal.

Bayer and McCreight's statement that they have developed a scheme with retrieval time proportional to $\log_k I$, where k is related to the page size, is very significant. As we will see, the use of a B-tree with a page size of sixty-four to index an file with 1 million records results in being able to find the key for any record in no more than three seeks to the disk. A binary search on the same file can require as many as twenty seeks. Moreover, we are talking about getting this kind of performance from a system that requires only minimal overhead as keys are inserted and deleted.

Before looking in detail at Bayer and McCreight's solution, let's first return to a more careful look at the problem, picking up where we left off in Chapter 7. We will also look at some of the data and file structures that were routinely used to attack the problem before the invention of B-trees. Given this background, it will be easier to appreciate the contribution made by Bayer and McCreight's work.

One last matter before we begin: why the name *B-tree*? Comer (1979) provides this footnote:

The origin of "B-tree" has never been explained by [Bayer and McCreight]. As we shall see, "balanced," "broad," or "bushy" might apply. Others suggest that the "B" stands for Boeing. Because of his contributions, however, it seems appropriate to think of B-trees as "Bayer"-trees.

9.2 Statement of the Problem

The fundamental problem with keeping an index on secondary storage is, of course, that accessing secondary storage is slow. This can be broken down into two more specific problems:

- *Searching the index must be faster than binary searching.* Searching for a key on a disk often involves seeking to different disk tracks. Since seeks are expensive, a search that has to look in more than three or four locations before finding the key often requires more time than is desirable. If we are using a binary search, four seeks is enough only to differentiate among fifteen items. An average of about 9.5 seeks is

required to find a key in an index of one thousand items using a binary search. We need to find a way to home in on a key using fewer seeks.

- *Insertion and deletion must be as fast as search.* As we saw in Chapter 7, if inserting a key into an index involves moving a large number of the other keys in the index, index maintenance is very nearly impractical on secondary storage for indexes consisting of only a few hundred keys, much less thousands of keys. We need to find a way to make insertions and deletions that have only local effects in the index rather than requiring massive reorganization.

These were the two critical problems that confronted Bayer and McCreight in 1970. They serve as guideposts for steering our discussion of the use of tree structures and multilevel indexes for secondary storage retrieval.

9.3 Indexing with Binary Search Trees

Let's begin by addressing the second of these two problems: looking at the cost of keeping a list in sorted order so we can perform binary searches. Given the sorted list in Fig. 9.1, we can express a binary search of this list as a *binary search tree*, as shown in Fig. 9.2.

Using elementary data structure techniques, it is a simple matter to create nodes that contain right and left link fields so the binary search tree can be constructed as a linked structure. Figure 9.3 illustrates a linked representation of the first two levels of the binary search tree shown in Fig. 9.2. In each node, the left and right links point to the left and right *children* of the node.

What is wrong with binary search trees? We have already said that binary search is not fast enough for disk resident indexing. Hence, a binary search tree cannot solve our problem as stated earlier. However, this is not the only problem with binary search trees. Chief among these is the lack of an effective strategy of balancing the tree. That is, making sure that the height of the leaves of the tree is uniform: no leaf is much farther from the root than any other leaf. Historically, a number of attempts were made to solve these problems, and we will look at two of them: AVL trees and paged binary trees.

AX CL DE FB FT HN JD KF NR PA RF SD TK WS YJ

Figure 9.1 Sorted list of keys.

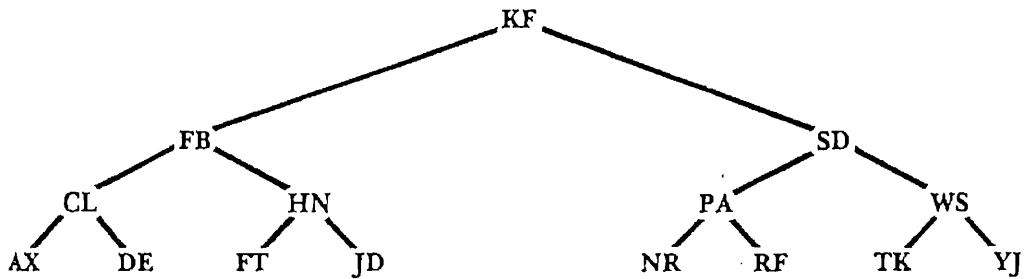


Figure 9.2 Binary search tree representation of the list of keys.

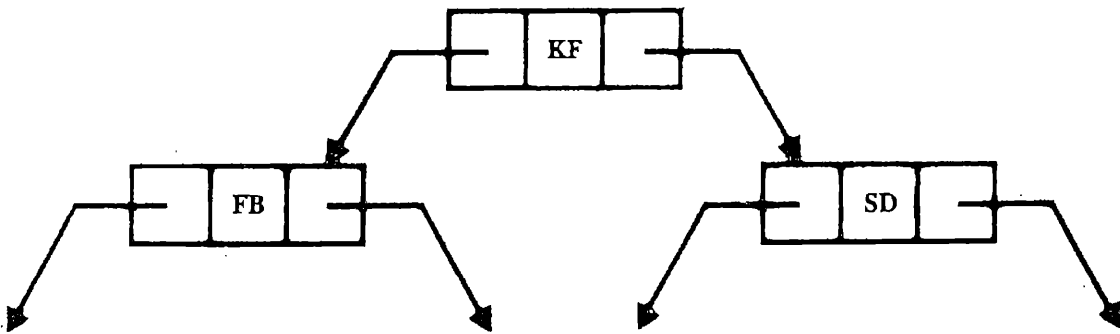
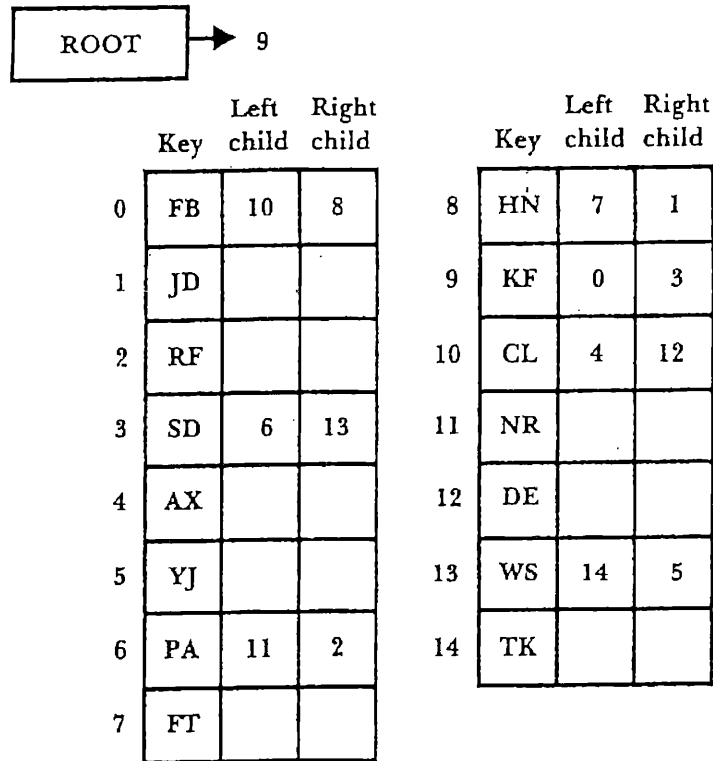


Figure 9.3 Linked representation of part of a binary search tree.

However, to focus on the costs and not the advantages is to miss the important new capability that this tree structure gives us: we no longer have to sort the file to perform a binary search. Note that the records in the file illustrated in Fig. 9.4 appear in random rather than sorted order. The sequence of the records in the file has no necessary relation to the structure of the tree; all the information about the logical structure is carried in the link fields. The very positive consequence that follows from this is that if we add a new key to the file, such as LV, we need only link it to the appropriate leaf node to create a tree that provides search performance that is as good as we would get with a binary search on a sorted list. The tree with LV added is illustrated in Fig. 9.5 (page 376).

Search performance on this tree is still good because the tree is in a *balanced* state. By balanced we mean that the height of the shortest path to a leaf does not differ from the height of the longest path by more than one level. For the tree in Fig. 9.5, this difference of one is as close as we can get to *complete balance*, in which all the paths from root to leaf are exactly the same length.

Figure 9.4
Record contents for a linked representation of the binary tree in Figure 9.2.



Consider what happens if we go on to enter the following eight keys to the tree in the sequence in which they appear:

NP MB TM LA UF ND TS NK

Just searching down through the tree and adding each key at its correct position in the search tree results in the tree shown in Fig. 9.6.

The tree is now out of balance. This is a typical result for trees that are built by placing keys into the tree as they occur without rearrangement. The resulting disparity between the length of various search paths is undesirable in any binary search tree, but it is especially troublesome if the nodes of the tree are being kept on secondary storage. There are now keys that require seven, eight, or nine seeks for retrieval. A binary search on a sorted list of these twenty-four keys requires only five seeks in the worst case. Although the use of a tree lets us avoid sorting, we are paying for this convenience in terms of extra seeks at retrieval time. For trees with hundreds of keys, in which an out-of-balance search path might extend to thirty, forty, or more seeks, this price is too high.

If each node is treated as a fixed-length record in which the link fields contain relative record numbers (RRNs) pointing to other nodes, then it is possible to place such a tree structure on secondary storage. Figure 9.4

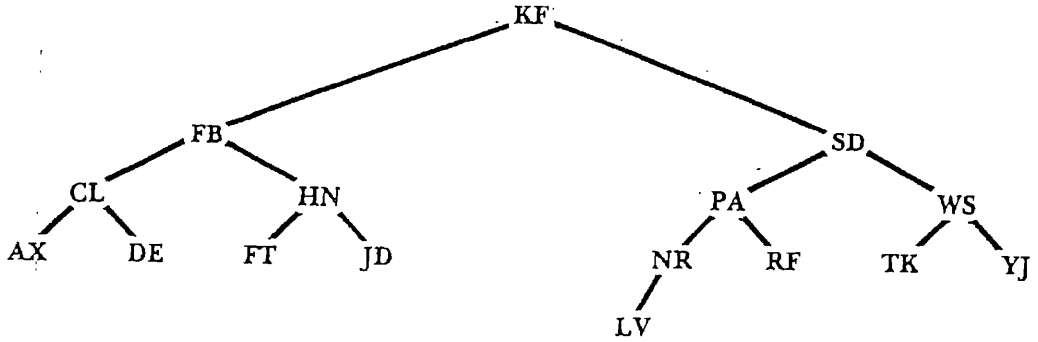


Figure 9.5 Binary search tree with LV added.

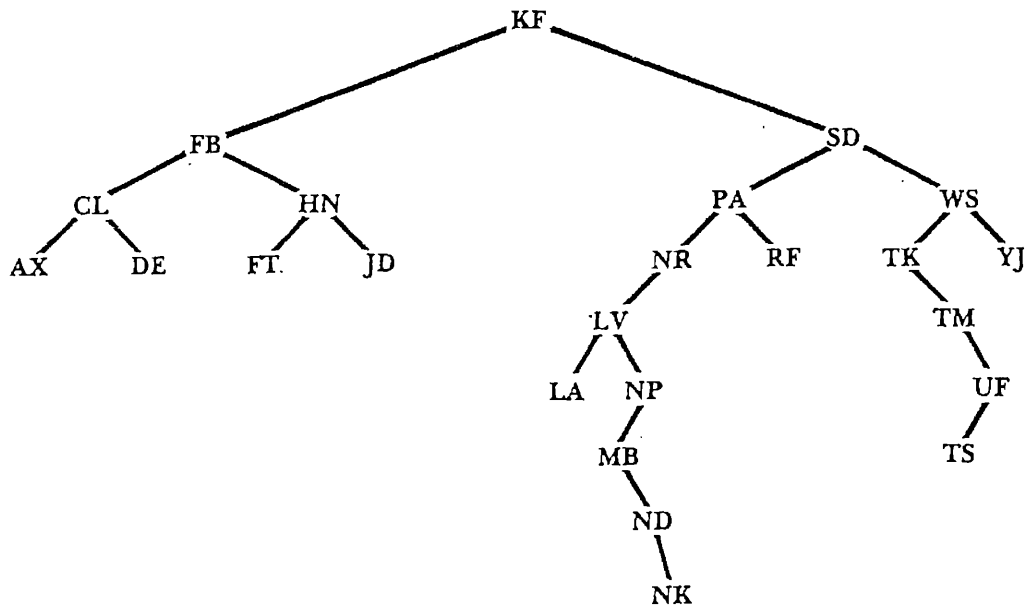


Figure 9.6 Binary search tree showing the effect of added keys.

illustrates the contents of the fifteen records that would be required to form the binary tree depicted in Fig. 9.2.

Note that more than half of the link fields in the file are empty because they are leaf nodes with no children. In practice, leaf nodes need to contain some special character, such as -1, to indicate that the search through the tree has reached the leaf level and that there are no more nodes on the search path. We leave the fields blank in this figure to make them more noticeable, illustrating the potentially substantial cost in terms of space utilization incurred by this kind of linked representation of a tree.

9.3.1 AVL Trees

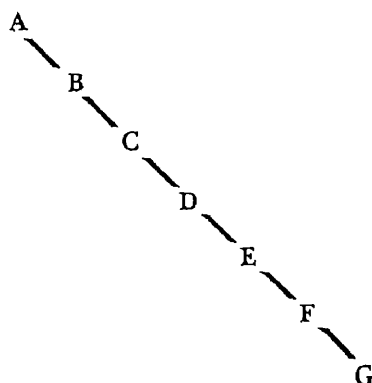
Earlier we said that there is no *necessary* relationship between the order in which keys are entered and the structure of the tree. We stress the word *necessary* because it is clear that order of entry is, in fact, important in determining the structure of the sample tree illustrated in Fig. 9.6. The reason for this sensitivity to the order of entry is that, so far, we have just been linking the newest nodes at the leaf levels of the tree. This approach can result in some very undesirable tree organizations. Suppose, for example, that our keys consist of the letters A–G and that we receive these keys in alphabetical order. Linking the nodes as we receive them produces a degenerate tree that is, in fact, nothing more than a linked list, as illustrated in Fig. 9.7.

The solution to this problem is somehow to reorganize the nodes of the tree as we receive new keys, maintaining a near optimal tree structure. One elegant method for handling such reorganization results in a class of trees known as *AVL trees*, in honor of the pair of Russian mathematicians, G. M. Adel'son-Vel'skii and E. M. Landis, who first defined them. An AVL tree is a *height-balanced* tree. This means that there is a limit placed on the amount of difference allowed between the heights of any two subtrees sharing a common root. In an AVL tree the maximum allowable difference is one. An AVL tree is therefore called a *height-balanced 1-tree* or *HB(1) tree*. It is a member of a more general class of height-balanced trees known as *HB(k) trees*, which are permitted to be *k* levels out of balance.

The trees illustrated in Fig. 9.8 have the AVL, or HB(1) property. Note that no two subtrees of any root differ by more than one level. The trees in Fig. 9.9 are *not* AVL trees. In each of these trees, the root of the subtree that is not in balance is marked with an X.

Figure 9.7

A degenerate tree.



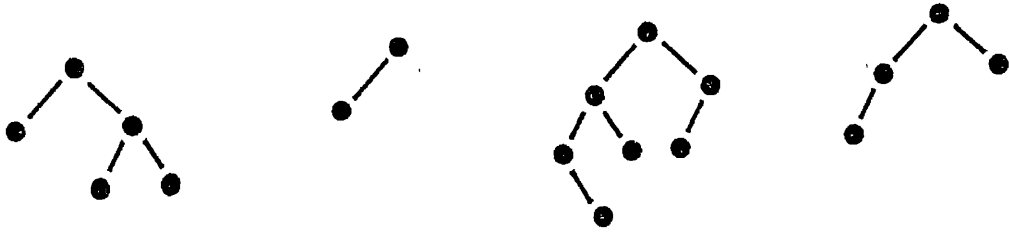


Figure 9.8 AVL trees.

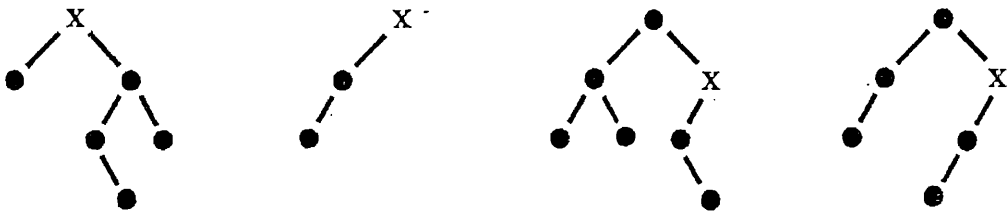


Figure 9.9 Trees that are not AVL trees.

The two features that make AVL trees important are

- By setting a maximum allowable difference in the height of any two subtrees, AVL trees guarantee a minimum level of performance in searching; and
- Maintaining a tree in AVL form as new nodes are inserted involves the use of one of a set of four possible rotations. Each of the rotations is confined to a single, local area of the tree. The most complex of the rotations requires only five pointer reassignments.

AVL trees are an important class of data structure. The operations used to build and maintain AVL trees are described in Knuth (1998), Standish (1980), and elsewhere. AVL trees are not themselves directly applicable to most file structure problems because, like all strictly *binary* trees, they have too many levels—they are too *deep*. However, in the context of our general discussion of the problem of accessing and maintaining indexes that are too large to fit in memory, AVL trees are interesting because they suggest that it is possible to define procedures that maintain height balance.

The fact that an AVL tree is height-balanced guarantees that search performance approximates that of a *completely balanced* tree. For example, the completely balanced form of a tree made up from the input keys

B C G E F D A

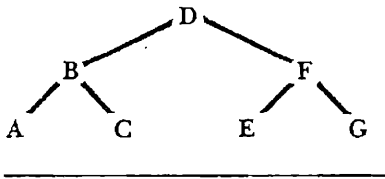


Figure 9.10

A completely balanced search tree.

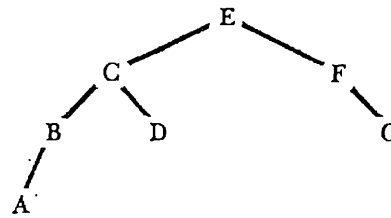


Figure 9.11 A search tree constructed using AVL procedures.

is illustrated in Fig. 9.10, and the AVL tree resulting from the same input keys, arriving in the same sequence, is illustrated in Fig. 9.11.

For a completely balanced tree, the worst-case search to find a key, given N possible keys, looks at

$$\log_2 (N + 1)$$

levels of the tree. For an AVL tree, the worst-case search could look at

$$1.44 \log_2 (N + 2)$$

levels. So, given 1 000 000 keys, a completely balanced tree requires seeking to 20 levels for some of the keys, but never to 21 levels. If the tree is an AVL tree, the maximum number of levels increases to only 29. This is a very interesting result, given that the AVL procedures guarantee that a single reorganization requires no more than five pointer reassignments. Empirical studies by VanDoren and Gray (1974), among others, have shown that such local reorganizations are required for approximately every other insertion into the tree and for approximately every fourth deletion. So height balancing using AVL methods guarantees that we will obtain a reasonable approximation of optimal binary tree performance at a cost that is acceptable in most applications using primary, random-access memory.

When we are using secondary storage, a procedure that requires more than five or six seeks to find a key is less than desirable; twenty or twenty-eight seeks is unacceptable. Returning to the two problems that we identified earlier in this chapter,

- Binary searching requires too many seeks, and
- Keeping an index in sorted order is expensive,

we can see that height-balanced trees provide an acceptable solution to the second problem. Now we need to turn our attention to the first problem.

of the 63 nodes in the tree with no more than two disk accesses. Note that every page holds 7 nodes and can branch to eight new pages. If we extend the tree to one additional level of paging, we add sixty-four new pages; we can then find any one of 511 nodes in only three seeks. Adding yet another level of paging lets us find any one of 4095 nodes in only four seeks. A binary search of a list of 4095 items can take as many as twelve seeks.

Clearly, breaking the tree into pages has the potential to result in faster searching on secondary storage, providing us with much faster retrieval than any other form of keyed access that we have considered up to this point. Moreover, our use of a page size of seven in Fig. 9.12 is dictated more by the constraints of the printed page than by anything having to do with secondary storage devices. A more typical example of a page size might be 8 kilobytes, capable of holding 511 key/reference field pairs. Given this page size and assuming that each page contains a completely balanced full tree and that the pages are organized as a completely balanced full tree, it is then possible to find any one of 134 217 727 keys with only three seeks. That is the kind of performance we are looking for. Note that, while the number of seeks required for a worst-case search of a completely full, balanced binary tree is

$$\log_2 (N + 1)$$

where N is the number of keys in the tree, the number of seeks required for the *paged* versions of a completely full, balanced tree is

$$\log_{k+1} (N + 1)$$

where N is, once again, the number of keys. The new variable, k , is the number of keys held in a single page. The second formula is actually a generalization of the first, since the number of keys in a page of a purely binary tree is 1. It is the logarithmic effect of the page size that makes the impact of paging so dramatic:

$$\begin{aligned} \log_2 (134\,217\,727 + 1) &= 27 \text{ seeks} \\ \log_{511+1} (134\,217\,727 + 1) &= 3 \text{ seeks} \end{aligned}$$

The use of large pages does not come free. Every access to a page requires the transmission of a large amount of data, most of which is not used. This extra transmission time is well worth the cost, however, because it saves so many seeks, which are far more time-consuming than the extra transmissions. A much more serious problem, which we look at next, has to do with keeping the paged tree organized.

9.3.3 Problems with Paged Trees

The major problem with paged trees is still inefficient disk usage. In the example in Fig. 9.12, there are seven tree nodes per page. Of the fourteen reference fields in a single page, six of them are reference nodes within the page. That is, we are using fourteen reference fields to distinguish between eight subtrees. We could represent the same information with seven key fields and eight subtree references. A significant amount of the space in the node is still being wasted.

Is there any advantage to storing a binary search tree within the page? It's true that in doing so we can perform binary search. However, if the keys are stored in an array, we can still do our binary search. The only problem here is that insertion requires a linear number of operations. We have to remember, however, that the factor that determines the cost of search is the number of disk accesses. We can do almost anything in memory in the time it takes to read a page. The bottom line is that there is no compelling reason to produce a tree inside the page.

The second problem, if we decide to implement a paged tree, is how to build it. If we have the entire set of keys in hand before the tree is built, the solution to the problem is relatively straightforward: we can sort the list of keys and build the tree from this sorted list. Most important, if we plan to start building the tree from the root, we know that the middle key in the sorted list of keys should be the *root key* within the *root page* of the tree. In short, we know where to begin and are assured that this beginning point will divide the set of keys in a balanced manner.

Unfortunately, the problem is much more complicated if we are receiving keys in random order and inserting them as soon as we receive them. Assume that we must build a paged tree as we receive the following sequence of single-letter keys:

C S D T A M P I B W N G U R K E H O L J Y Q Z F X V

We will build a paged binary tree that contains a maximum of three keys per page. As we insert the keys, we rotate them within a page as necessary to keep each page as balanced as possible. The resulting tree is illustrated in Fig. 9.13. Evaluated in terms of the depth of the tree (measured in pages), this tree does not turn out too badly. (Consider, for example, what happens if the keys arrive in alphabetical order.)

Even though this tree is not dramatically misshapen, it clearly illustrates the difficulties inherent in building a paged binary tree from the top down. When you start from the root, the initial keys must, of necessity, go into the root. In this example at least two of these keys, C and D, are not

keys that we want there. They are adjacent in sequence and tend toward the beginning of the total set of keys. Consequently, they force the tree out of balance.

Once the wrong keys are placed in the root of the tree (or in the root of any subtree farther down the tree), what can you do about it? Unfortunately, there is no easy answer to this. We cannot simply rotate entire pages of the tree in the same way that we would rotate individual keys in an unpagged tree. If we rotate the tree so the initial root page moves down to the left, moving the C and D keys into a better position, then the S key is out of place. So we must break up the pages. This opens up a whole world of possibilities and difficulties. Breaking up the pages implies rearranging them to create new pages that are both internally balanced and well arranged relative to other pages. Try creating a page rearrangement algorithm for the simple, three-keys-per-page tree from Fig. 9.13. You will find it very difficult to create an algorithm that has only local effects, rearranging just a few pages. The tendency is for rearrangements and adjustments to spread out through a large part of the tree. This situation grows even more complex with larger page sizes.

So, although we have determined that collecting keys into pages is a very good idea from the standpoint of reducing seeks to the disk, we have

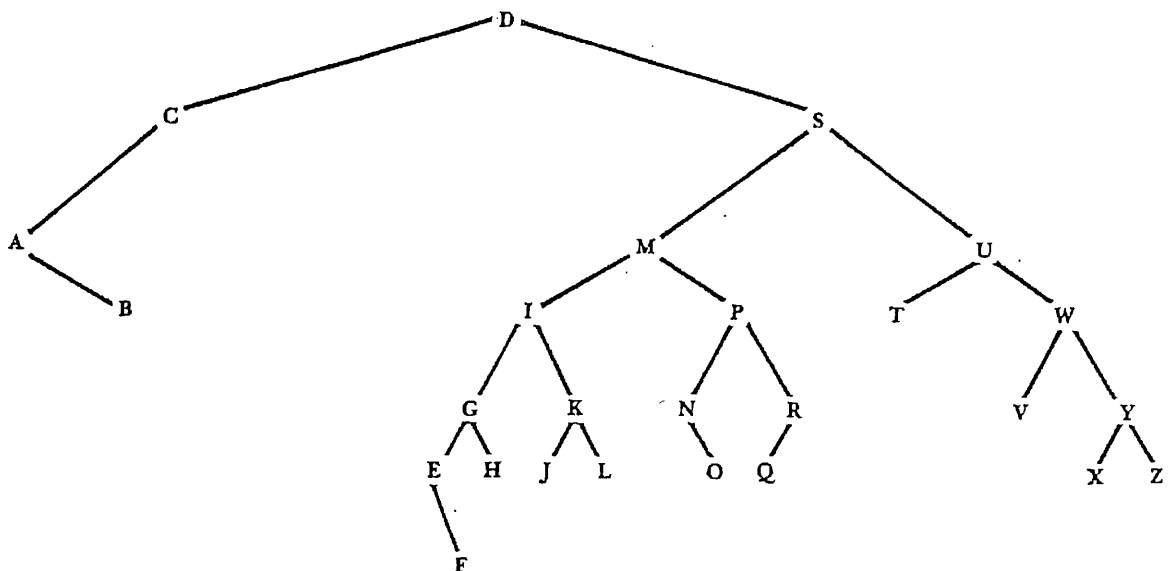


Figure 9.13 Paged tree constructed from keys arriving in random input sequence.

not yet found a way to collect the right keys. We are still confronting at least two unresolved questions:

- How do we ensure that the keys in the root page turn out to be good *separator* keys, dividing up the set of other keys more or less evenly?
- How do we avoid grouping keys, such as C, D, and S in our example, that should not share a page?

There is, in addition, a third question that we have not yet had to confront because of the small page size of our sample tree:

- How can we guarantee that each of the pages contains at least some minimum number of keys? If we are working with a larger page size, such as 8191 keys per page, we want to avoid situations in which a large number of pages each contains only a few dozen keys.

Bayer and McCreight's 1972 B-tree article provides a solution directed precisely at these questions.

A number of the elegant, powerful ideas used in computer science have grown out of looking at a problem from a different viewpoint. B-trees are an example of this viewpoint-shift phenomenon.

The key insight required to make the leap from the kinds of trees we have been considering to a new solution, B-trees, is that we can choose to *build trees upward from the bottom instead of downward from the top*. So far, we have assumed the necessity of starting construction from the root as a given. Then, as we found that we had the wrong keys in the root, we tried to find ways to repair the problem with rearrangement algorithms. Bayer and McCreight recognized that the decision to work down from the root was, of itself, the problem. Rather than finding ways to undo a bad situation, they decided to avoid the difficulty altogether. With B-trees, you allow the root to *emerge*, rather than set it up and then find ways to change it.

9.4 Multilevel Indexing: A Better Approach to Tree Indexes

The previous section attempted to develop an ideal strategy for indexing large files based on building search trees, but serious flaws were uncovered. In this section we take a different approach. Instead of basing our strategy on binary tree searches, we start with the single record indexing strategy of

Chapter 7. We extend this to multirecord indexes and then multilevel indexes. Ultimately, this approach, too, is flawed, but it is the source of the primary efficiency of searching and leads us directly to B-trees.

In Chapter 7, we noted that a single record index puts a limit on the number of keys allowed and that large files need multirecord indexes. A multirecord index consists of a sequence of simple index records. The keys in one record in the list are all smaller than the keys of the next record. A binary search is possible on a file that consists of an ordered sequence of index records, but we already know that binary search is too expensive.

To illustrate the benefits of an indexed approach, we use the large example file of Chapter 8, an 80-megabyte file of 8 000 000 records, 100 bytes each, with 10-byte keys. An index of this file has 8 000 000 key-reference pairs divided among a sequence of index records. Let's suppose that we can put 100 key-reference pairs in a single index record. Hence there are 80 000 records in the index. In order to build the index, we need to read the original file, extract the key from each record, and sort the keys. The strategies outlined in Chapter 8 can be used for this sorting. The 100 largest keys are inserted into an index record, and that record is written to the index file. The next largest 100 keys go into the next record of the file, and so on. This continues until we have 80 000 index records in the index file. Although we have reduced the number of records to be searched by a factor of 100, we still must find a way to speed up the search of this 80 000-record file.

Can we build an index of the index file, and how big will it be? Since the index records form a sorted list of keys, we can choose one of the keys (for example, the largest) in each index record as the key of that whole record. These *second-level* keys can be used to build a second-level index with 80 000 keys, or 800 index records. In searching the second-level index for a key k , we choose the smallest second-level key that is greater than or equal to k . If k is in the first-level index, it must be in the block referenced by that second-level key.

Continuing to a third level, we need just 8 index records to index the largest keys in the 800 second-level records. Finally, the fourth level consists of a single index record with only 8 keys. These four levels together form an index tree with a fan-out of 100 and can be stored in a single index file. Each node of the tree is an index record with 100 children. Each of the children of a node is itself an index node, except at the leaves. The children of the leaf nodes are data records.

A single index file containing the full four-level index of 8 000 000 records requires 80 809 index records, each with 100 key-reference

pairs. The lowest level index is an index to the data file, and its reference fields are record addresses in the data file. The other indexes use their reference fields for index record addresses, that is, addresses within the index file.

The costs associated with this multilevel index file are the space overhead of maintaining the extra levels, the search time, and the time to insert and delete elements. The space overhead is 809 more records than the 80 000 minimum for an index of the data file. This is just 1 percent. Certainly this is not a burden.

The search time is simple to calculate—it's three disk reads! An analysis of search time always has multiple parts: the minimum search time, the maximum search time, and the average search time for keys that are in the index and for keys that are not in the index. For this multilevel index, all of these cases require searching four index records. That is, each level of the index must be searched. For a key that is in the index, we need to search all the way to the bottom level to get the data record address. For a key not in the index, we need to search all the way to the bottom to determine that it is missing. The average, minimum, and maximum number of index blocks to search are all four, that is, the number of levels in the index. Since there is only one block at the top level, we can keep that block in memory. Hence, a maximum of three disk accesses are required for any key search. It might require fewer disk reads if any of the other index records are already in memory.

Look how far we've come: an arbitrary record in an 80-megabyte file can be read with just four disk accesses—three to search the index and one to read the data record. The total space overhead, including the primary index, is well below 10 percent of the data file size. This tree is not full, since the root node has only eight children and can accommodate one hundred. This four-level tree will accommodate twelve times this many data records, or a total of 100 million records in a file of 10 gigabytes. Any one of these records can be found with only three disk accesses. This is what we need to produce efficient indexed access!

The final factor in the cost of multilevel indexes is the hardest one. How can we insert keys into the index? Recall that the first-level index is an ordered sequence of records. Does this imply that the index file must be sorted? The search strategy relies on indexes and record addresses, not on record placement in the file. As with the simple indexes of Chapter 7, this indexed search supports entry-sequenced records. As long as the location of the highest level index record is known, the other records can be anywhere in the file.

Having an entry-sequenced index file does not eliminate the possibility of linear insertion time. For instance, suppose a new key is added that will be the smallest key in the index. This key must be inserted into the first record of the first-level index. Since that record is already full with one hundred elements, its largest key must be inserted into the second record, and so on. Every record in the first-level index must be changed. This requires 80 000 reads and writes. This is truly a fatal flaw in simple multi-level indexing.

9.5 B-Trees: Working up from the Bottom

B-trees are multilevel indexes that solve the problem of linear cost of insertion and deletion. This is what makes B-trees so good, and why they are now the standard way to represent indexes. The solution is twofold. First, don't require that the index records be full. Second, don't shift the overflow keys to the next record; instead split an overfull record into two records, each half full. Deletion takes a similar strategy of merging two records into a single record when necessary.

Each node of a B-tree is an index record. Each of these records has the same maximum number of key-reference pairs, called the *order* of the B-tree. The records also have a minimum number of key-reference pairs, typically half of the order. A B-tree of order one hundred has a minimum of fifty keys and a maximum of one hundred keys per record. The only exception is the single root node, which can have a minimum of two keys.

An attempt to insert a new key into an index record that is not full is cheap. Simply update the index record. If the new key is the new largest key in the index record, it is the new higher-level key of that record, and the next higher level of the index must be updated. The cost is bounded by the height of the tree.

When insertion into an index record causes it to be overfull, it is split into two records, each with half of the keys. Since a new index node has been created at this level, the largest key in this new node must be inserted into the next higher level node. We call this the *promotion* of the key. This promotion may cause an overflow at that level. This in turn causes that node to be split, and a key promoted to the next level. This continues as far as necessary. If the index record at the highest level overflows, it must be split. This causes another level to be added to the multilevel index. In this way, a B-tree grows up from the leaves. Again the cost of insertion is bounded by the height of the tree.

The rest of the secrets of B-trees are just working out the details. How to split nodes, how to promote keys, how to increase the height of the tree, and how to delete keys.

9.6 Example of Creating a B-Tree

Let's see how a B-tree grows given the key sequence that produces the paged binary tree illustrated in Fig. 9.13. The sequence is

C S D T A M P I B W N G U R K E H O L J Y Q Z F X V

We use an order four B-tree (maximum of four key-reference pairs per node). Using such a small node size has the advantage of causing pages to split more frequently, providing us with more examples of splitting. We omit explicit indication of the reference fields so we can fit a larger tree on the printed page.

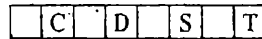
Figure 9.14 illustrates the growth of the tree up to the point where it is about to split the root for the second time. The tree starts with a single empty record. In Fig. 9.14(a), the first four keys are inserted into that record. When the fifth key, A, is added in Fig. 9.14(b), the original node is split and the tree grows by one level as a new root is created. The keys in the root are the largest key in the left leaf, D, and the largest key in the right leaf, T.

The keys M, P, and I all belong in the rightmost leaf node, since they are larger than the largest key in the right node. However, inserting I makes the rightmost leaf node overfull, and it must be split, as shown in Fig. 9.14(c). The largest key in the new node, P, is inserted into the root. This process continues in Figs. 9.14(d) and (e), where B, W, N, G, and U are inserted.

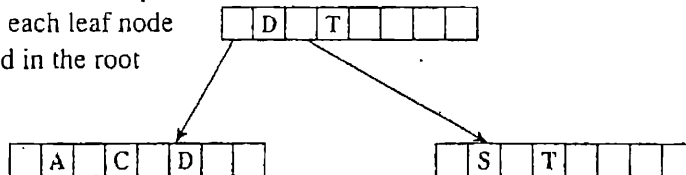
In the tree of Fig. 9.14(e), the next key in the list, R, should be put into the rightmost leaf node, since it is greater than the largest key in the previous node, P, and less than or equal to the largest key in that node, W. However, the rightmost leaf node is full, and so is the root. Splitting that leaf node will overfill the root node. At this point a new root must be created, and the height of the tree increased to three.

Figure 9.15 shows the tree as it grows to height three. The figure also shows how the tree continues to grow as the remaining keys in the sequence are added. Figure 9.15(b) stops after Z is added. The next key in the sequence, F, requires splitting the second-leaf node, as shown in Fig. 9.15(c). Although the leaf level of the tree is not shown in a single line, it is still a single level. Insertions of X and V causes the rightmost leaf to be

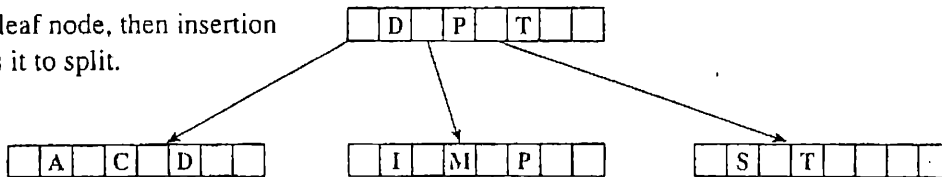
a) Insertions of C, S, D, T into the initial node.



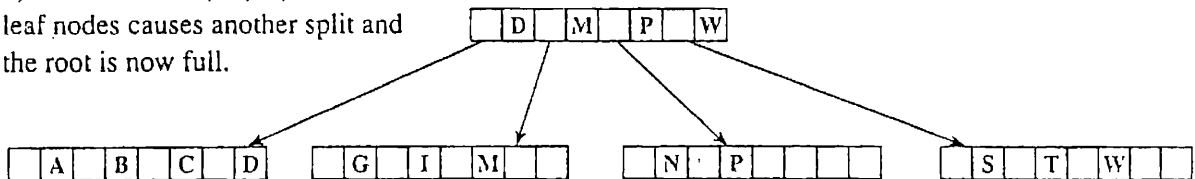
b) Insertion of A causes node to split and the largest key in each leaf node (D and T) to be placed in the root node.



c) M and P are inserted into the rightmost leaf node, then insertion of I causes it to split.



d) Insertions of B, W, N, and G into leaf nodes causes another split and the root is now full.



e) Insertion of U proceeds without incident, but R would have to be inserted into the rightmost leaf, which is full.

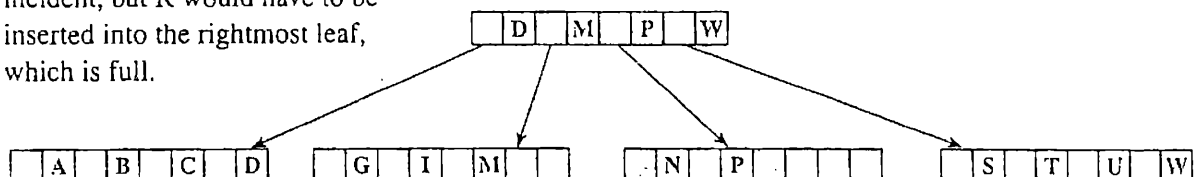
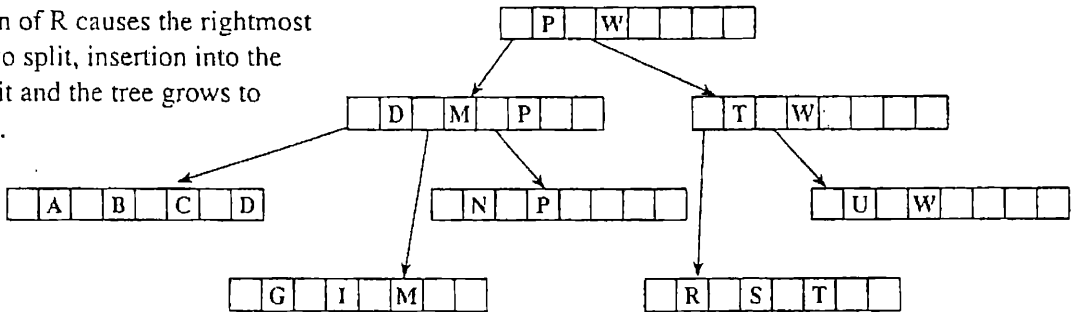
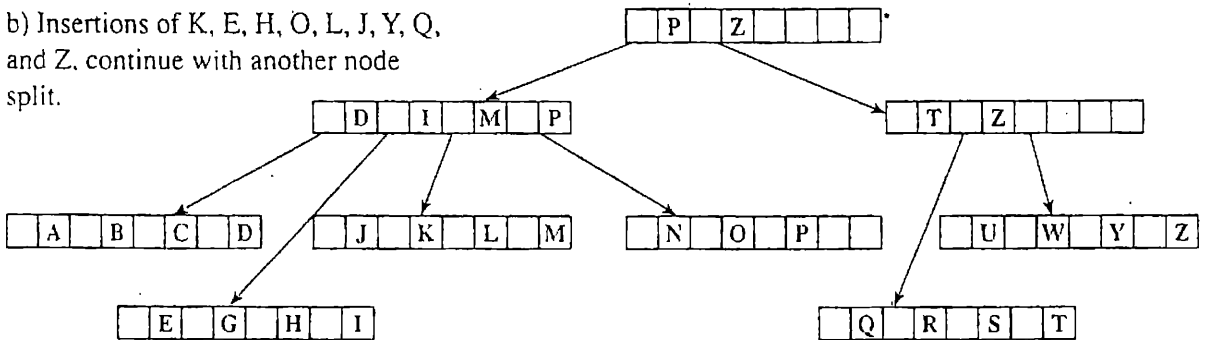


Figure 9.14 Growth of a B-tree, part 1. The tree grows to a point at which the root needs to be split the second time.

a) Insertion of R causes the rightmost leaf node to split, insertion into the root to split and the tree grows to level three.



b) Insertions of K, E, H, O, L, J, Y, Q, and Z. continue with another node split.



c) Insertions of F, X, and V finish the insertion of the alphabet.

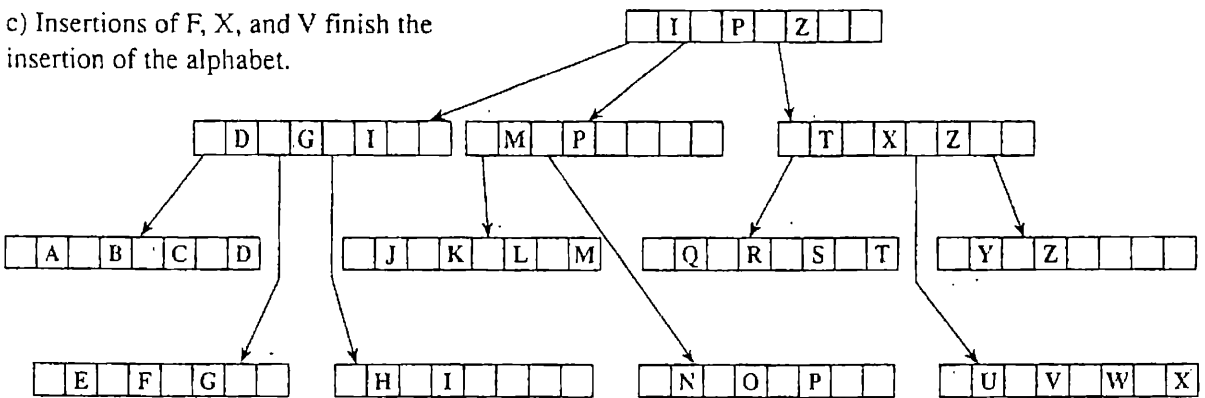


Figure 9.15 Growth of a B-tree, part 2. The root splits to level three; remaining keys are inserted.

overfull and split. The rightmost leaf of the middle level is also overfull and is split. All twenty-six letters are inserted into a tree of height three and order four.

Note that the number of nodes affected by any insertion is never more than two nodes per level (one changed and a new one created by a split), so the insertion cost is strictly linear in the height of the tree.

9.7 An Object-Oriented Representation of B-Trees

9.7.1 Class `BTreeNode`: Representing B-Tree Nodes in Memory

As we have seen, a B-tree is an index file associated with a data file. Most of the operations on B-trees, including insertion and deletion, are applied to the B-tree nodes in memory. The B-tree file simply stores the nodes when they are not in memory. Hence, we need a class to represent the memory resident B-tree nodes. Class `BTreeNode`, given in Fig. 9.16 and in file `btnode.h` of Appendix I, is a template class based on the `SimpleIndex` template class that was described in Section 7.4.3. Note that a `BTreeNode` object has methods to insert and remove a key and to split and merge nodes. There are also protected members that store the file address of the node and the minimum and maximum number of keys. You may notice that there is no search method defined in the class. The search method of the base class `SimpleIndex` works perfectly well.

It is important to note that not every data member of a `BTreeNode` has to be stored when the object is not in memory. The difference between the memory and the disk representations of `BTreeNode` objects is managed by the `pack` and `unpack` operations.

Class `BTreeNode` is designed to support some optimizations of the in-memory operations. For example, the number of keys is actually one more than the order of the tree, as shown in the constructor. The call to the `SimpleIndex` constructor creates an index record with `maxKeys+1` elements:

```
template <class keyType>
BTreeNode<keyType>::BTreeNode(int maxKeys, int unique)
    : SimpleIndex<keyType>(maxKeys+1, unique)
{ Init ();}
```

```

template <class keyType>
class BTreeNode: public SimpleIndex <keyType>
// this is the in-memory version of the BTreeNode
(public:
    BTreeNode(int maxKeys, int unique = 1);
    int Insert (const keyType key, int recAddr);
    int Remove (const keyType key, int recAddr = -1);
    int LargestKey (); // returns value of Largest key
    int Split (BTreeNode<keyType>*newNode); //move into newNode
    int Pack (IOBuffer& buffer) const;
    int Unpack (IOBuffer& buffer);
protected:
    int MaxBKeys; // maximum number of keys in a node
    int Init ();
    friend class BTree<keyType>;
};

```

Figure 9.16 The main members and methods of class BTreeNode: template class for B-tree node in memory.

For this class, the order of the B-tree node (member MaxBKeys) is one less than the value of MaxKeys, which is a member of the base class SimpleIndex. Making the index record larger allows the Insert method to create an overflow node. The caller of BTreeNode::Insert needs to respond to the overflow in an appropriate fashion. Similarly, the Remove method can create an underfull node.

Method Insert simply calls SimpleIndex::Insert and then checks for overflow. The value returned is 1 for success, 0 for failure, and -1 for overflow:

```

template <class keyType>
int BTreeNode<keyType>::Insert (const keyType key, int recAddr)
{
    int result = SimpleIndex<keyType>::Insert (key, recAddr);
    if (!result) return 0; // insert failed
    if (NumKeys > MaxBKeys) return -1; // node overflow
    return 1;
}

```

9.7.2 Class BTree: Supporting Files of B-Tree Nodes

We now look at class `BTree` which uses in-memory `BTreeNode` objects, adds the file access portion, and enforces the consistent size of the nodes. Figure 9.17 and file `btree.h` of Appendix I contain the definition of class `BTree`. Here are methods to create, open, and close a B-tree and to search; insert, and remove key-reference pairs. In the protected area of the class, we find methods to transfer nodes from disk to memory (`Fetch`) and back to disk (`Store`). There are members that hold the root node in memory and represent the height of the tree and the file of index records. Member `Nodes` is used to keep a collection of tree nodes in memory and reduce disk accesses, as will be explained later.

```

template <class keyType>
class BTree
{public:
    BTree(int order, int keySize=sizeof(keyType), int unique=1);
    int Open (char * name, int mode);
    int Create (char * name, int mode);
    int Close ();
    int Insert (const keyType key, const int recAddr);
    int Remove (const keyType key, const int recAddr = -1);
    int Search (const keyType key, const int recAddr = -1);
protected:
    typedef BTreeNode<keyType> BTreeNode; // necessary shorthand
    BTreeNode * FindLeaf (const keyType key);
    // load a branch into memory down to the leaf with key
    BTreeNode * Fetch(const int recaddr);//load node into memory
    int Store (BTreeNode *);// store node into file
    BTreeNode Root;
    int Height; // height of tree
    int Order; // order of tree
    BTreeNode ** Nodes; // storage for a branch
    // Nodes[1] is level 1, etc. (see FindLeaf)
    // Nodes[Height-1] is leaf
    RecordFile<BTreeNode> BTreeFile;
};

```

Figure 9.17 Main members and methods of class `BTree`: whole B-tree implementation—including methods `Create`, `Open`, `Search`, `Insert`, and `Remove`.

9.8 B-Tree Methods Search, Insert, and Others

Now that we have seen the principles of B-tree operations and we have the class definitions and the single node operations, we are ready to consider the details of the B-tree methods.

9.8.1 Searching

The first B-tree method we examine is a tree-searching procedure. Searching is a good place to begin because it is relatively simple, yet it still illustrates the characteristic aspects of most B-tree algorithms:

- They are iterative, and
- They work in two stages, operating alternatively on entire pages (class `BTree`) and then *within* pages (class `BTreeNode`).

The searching procedure is iterative, loading a page into memory and then searching through the page, looking for the key at successively lower levels of the tree until it reaches the leaf level. Figure 9.18 contains the code for method `Search` and the protected method `FindLeaf` that does almost all of the work. Let's work through the methods by hand, searching for the key `L` in the tree illustrated in Fig. 9.15(a). For an object `btree` of type `BTree<char>` and an integer `recAddr`, the following code finds that there is no data file record with key `L`:

```
recAddr = btree.Search ('L');
```

Method `Search` calls method `FindLeaf`, which searches down a branch of the tree, beginning at the root, which is referenced by the pointer value `Nodes[0]`. In the first iteration, with `level = 1`, the line

```
recAddr = Nodes[level-1]->Search(key, -1, 0);
```

is an inexact search and finds that `L` is less than `P`, the first key in the record. Hence, `recAddr` is set to the first reference in the root node, which is the index file address of the first node in the second level of the tree of Fig. 9.15(a). The line

```
Nodes[level]=Fetch(recAddr);
```

reads that second-level node into a new `BTreeNode` object and makes `Nodes[1]` point to this new object. The second iteration, with `level = 2`, searches for `L` in this node. Since `L` is less than `M`, the second key in the

```

template <class keyType>
int BTree<keyType>::Search (const keyType key, const int recAddr)
{
    BTreeNode<keyType>.* leafNode;
    leafNode = FindLeaf (key);
    return leafNode -> Search (key, recAddr);
}

template <class keyType>
BTreeNode<keyType> * BTree<keyType>::FindLeaf (const keyType key)
// load a branch into memory down to the leaf with key
{
    int recAddr, level;
    for (level = 1; level < Height; level++)
    {
        recAddr = Nodes[level-1]->Search(key, -1, 0); //inexact search
        Nodes[level]=Fetch(recAddr);
    }
    return Nodes[level-1];
}

```

Figure 9.18 Method BTree::Search and BTree::FindLeaf.

record, the second reference is selected, and the second node in the leaf level of the tree is loaded into `Nodes[2]`. After the for loop increments `level`, the iteration stops, and `FindLeaf` returns the address of this leaf node. At the end of this method, the array `Nodes` contains pointers to the complete branch of the tree.

After `FindLeaf` returns, method `Search` uses an exact search of the leaf node to find that there is no data record that has key `L`. The value returned is `-1`.

Now let's use method `Search` to look for `G`, which *is* in the tree of Fig. 9.15(a). It follows the same downward path that it did for `L`, but this time the exact search in method `Search` finds `G` in position 1 of the second-leaf node. It returns the first reference field in the node, which is the data file address of the record with key `G`.

9.8.2 Insertion

There are two important observations we can make about the insertion, splitting, and promotion process:

- It begins with a search that proceeds all the way down to the leaf level, and
- After finding the insertion location at the leaf level, the work of insertion, overflow detection, and splitting proceeds upward from the bottom.

Consequently, we can conceive of our iterative procedure as having three phases:

1. Search to the leaf level, using method `FindLeaf`, before the iteration;
2. Insertion, overflow detection, and splitting on the upward path;
3. Creation of a new root node, if the current root was split.

Let's use the example of inserting *R* and its data record address (called `recAddr`) into the tree of Fig. 9.14(e) so we can watch the insertion procedure work through these phases. The result of this insertion is shown in Fig. 9.15(a). Method `Insert` is the most complicated of the methods included in file `btree.tc` in Appendix I. We will look at some of its code here.

The first operation in method `Insert` is to search to the root for key *R* using `FindLeaf`:

```
thisNode = FindLeaf (key);
```

As described above, `FindLeaf` loads a complete branch into memory. In this case, `Nodes [0]` is the root node, and `Nodes [1]` is the rightmost leaf node (containing *S*, *T*, *U*, and *W*).

The next step is to insert *R* into the leaf node

```
result = thisNode -> Insert (key, recAddr);
```

The result here is that an overflow is detected. The object `thisNode` now has five keys. The node must be split into two nodes, using the following code:

```
newNode = NewNode();
thisNode -> Split (newNode);
Store(thisNode); Store(newNode);
```

Now the two nodes, one with keys *R*, *S*, and *T*, and one with *U* and *W*, have been stored back in the file. We are done with the leaf level and are ready to move up the tree.

The next step is to update the parent node. Since the largest key in `thisNode` has changed, method `UpdateKey` is used to record the change (`largestKey` has been set to the previous largest key in `thisNode`):

```
parentNode->UpdateKey(largestKey, thisNode->LargestKey());
```

Hence the value W in the root is changed to T. Then the largest value in the new node is inserted into the root of the tree:

```
parentNode->Insert(newNode->LargestKey(), newNode->RecAddr);
```

The value W is inserted into the root. This is often called *promoting* the key W. This causes the root to overflow with five keys. Again, the node is split, resulting in a node with keys D, M, and P, and one with T and W.

There is no higher level of the tree, so a new root node is created, and the keys P and W are inserted into it. This is accomplished by the following code:

```
int newAddr = BTreeFile.Append(Root); //put previous root into file
// insert 2 keys in new root node
Root.Keys[0]=thisNode->LargestKey();
Root.RecAddrs[0]=newAddr;
Root.Keys[1]=newNode->LargestKey();
Root.RecAddrs[1]=newNode->RecAddr;
Root.NumKeys=2;
Height++;
```

It begins by appending the old root node into the B-tree file. The very first index record in the file is always the root node, so the old root node, which is no longer the root, must be put somewhere else. Then the insertions are performed. Finally the height of the tree is increased by one.

Insert uses a number of support functions. The most obvious one is method `BTreeNode::Split` which distributes the keys between the original page and the new page. Figure 9.19 contains an implementation of this method. Some additional error checking is included in the full implementation in Appendix I. Method `Split` simply removes some of the keys and references from the overfull node and puts them into the new node.

The full implementation of `BTree::Insert` in Appendix I includes code to handle the special case of the insertion of a new largest key in the tree. This is the only case where an insertion adds a new largest key to a node. This can be verified by looking at method `FindLeaf`, which is used to determine the leaf node to be used in insertion. `FindLeaf` always chooses a node whose largest key is greater than or equal to the search key. Hence, the only case where `FindLeaf` returns a leaf node in which the search key is greater than the largest key is where that leaf node is the rightmost node in the tree and the search key is greater than any key in the tree. In this case, the insertion of the new key

```

template <class keyType>
int BTreeNode<keyType>::Split (BTreeNode<keyType> * newNode)
{
    // find the first Key to be moved into the new node
    int midpt = (NumKeys+1)/2;
    int numNewKeys = NumKeys - midpt;
    // move the keys and recaddrs from this to newNode
    for (int i = midpt; i < NumKeys; i++)
    {
        newNode->Keys[i-midpt] = Keys[i];
        newNode->RecAddrs[i-midpt] = RecAddrs[i];
    }
    // set number of keys in the two Nodes
    newNode->NumKeys = numNewKeys;
    NumKeys = midpt;
    return 1;
}

```

Figure 9.19 Method `Split` of class `BTreeNode`.

requires changing the largest key in the rightmost node in every level of the index. The code to handle this special case is included in `BTree::Insert`.

9.8.3 Create, Open, and Close

We need methods to create, open, and close B-tree files. Our object-oriented design and the use of objects from previous classes have made these methods quite simple, as you can see in file `btree.tc` of Appendix I. Method `Create` has to write the empty root node into the file `BTreeFile` so that its first record is reserved for that root node. Method `Open` has to open `BTreeFile` and load the root node into memory from the first record in the file. Method `Close` simply stores the root node into `BTreeFile` and closes it.

9.8.4 Testing the B-Tree

The file `tstbtree.cpp` in Appendix I has the full code of a program to test creation and insertion of a B-tree. Figure 9.20 contains most of the code. As you can see, this program uses a single character key (class

```

const char * keys="CSDTAMPIBWNGURKEHOLJYQZFXV";
const int BTreeSize = 4;
main (int argc, char * argv)
{
    int result, i;
    BTree <char> bt (BTreeSize);
    result = bt.Create ("testbt.dat", ios::in|ios::out);
    for (i = 0; i<26; i++)
    {
        cout<<"Inserting "<<keys[i]<<endl;
        result = bt.Insert(keys[i],i);
        bt.Print(cout); // print after each insert
    }
    return 1;
}

```

Figure 9.20 Program tstbtree.cpp.

BTree<char>) and inserts the alphabet in the same order as in Fig. 9.14 and 9.15. The tree that is created is identical in form to those pictured in the figures.

9.9 B-Tree Nomenclature

Before moving on to discuss B-tree performance and variations on the basic B-tree algorithms, we need to formalize our B-tree terminology. Providing careful definitions of terms such as *order* and *leaf* enables us to state precisely the properties that must be present for a data structure to qualify as a B-tree.

This definition of B-tree properties, in turn, informs our discussion of matters such as the procedure for deleting keys from a B-tree.

Unfortunately, the literature on B-trees is not uniform in its use of terms. Reading that literature and keeping up with new developments therefore require some flexibility and some background: the reader needs to be aware of the different uses of some of the fundamental terms.

For example, Bayer and McCreight (1972), Comer (1979), and a few others refer to the *order* of a B-tree as the *minimum* number of *keys* that can be in a page of a tree. So, our initial sample B-tree (Fig. 9.14), which

can hold a *maximum* of four keys per page, has an *order* of two, using Bayer and McCreight's terminology. The problem with this definition of order is that it becomes clumsy when you try to account for pages that hold an *odd*, maximum number of keys. For example, consider the following question: Within the Bayer and McCreight framework, is the page of an order three B-tree full when it contains six keys or when it contains seven keys?

Knuth (1998) and others have addressed the odd/even confusion by defining the *order* of a B-tree to be the *maximum* number of *descendants* that a page can have. This is the definition of *order* that we use in this text. Note that this definition differs from Bayer and McCreight's in two ways: it references a *maximum*, not a *minimum*, and it counts *descendants* rather than *keys*.

When you split the page of a B-tree, the descendants are divided as evenly as possible between the new page and the old page. Consequently, every page except the root and the leaves has at *least* $m/2$ descendants. Expressed in terms of a ceiling function, we can say that the minimum number of descendants is $\lceil m/2 \rceil$.

Another term that is used differently by different authors is *leaf*. Bayer and McCreight refer to the lowest level of keys in a B-tree as the leaf level. This is consistent with the nomenclature we have used in this text. Other authors, including Knuth, consider the leaves of a B-tree to be one level *below* the lowest level of keys. In other words, they consider the leaves to be the actual data records that might be pointed to by the lowest level of keys in the tree. We do *not* use this definition; instead we stick with the notion of leaf as the lowest level of B-tree nodes.

Finally, many authors call our definition of B-tree a *B⁺ tree*. The term *B-tree* is often used for a version of the B-tree that has data record references in all of the nodes, instead of only in the leaf nodes. A major difference is that our version has the full index in the leaf nodes and uses the interior nodes as higher level indexes. This results in a duplication of keys, since each key in an interior node is duplicated at each lower level. The other version eliminates this duplication of key values, and instead includes data record references in interior nodes. While it seems that this will save space and reduce search times, in fact it often does neither. The major deficiency of this version is that the size of the interior nodes is much larger for the same order B-tree. Another way to look at the difference is that for the same amount of space in the interior nodes, by eliminating the data references, we could significantly increase the order of the tree, resulting in shallower trees. Of course, the shallower the tree, the shorter the search.

In this book, we use the term *B⁺ tree* to refer to a somewhat more complex situation in which the data file is not entry sequenced but is organized into a linked list of sorted blocks of records. The data file is organized in much the same way as the leaf nodes of a B-tree. The great advantage of the B⁺ tree organization is that both indexed access and sequential access are optimized. This technique is explained in detail in the next chapter.

You may have recognized that the largest key in each interior B-tree node is not needed in the searching. That is, in method `FindLeaf`, whenever the search key is bigger than any key in the node, the search proceeds to the rightmost child. It is possible and common to implement B-trees with one less key than reference in each interior node. However, the insertion method is made more complicated by this optimization, so it has been omitted in the B-tree classes and is included as a programming exercise.

9.10 Formal Definition of B-Tree Properties

Given these definitions of order and leaf, we can formulate a precise statement of the properties of a B-tree of order m :

- Every page has a maximum of m descendants.
- Every page, except for the root and the leaves, has at least $\lceil m/2 \rceil$ descendants.
- The root has at least two descendants (unless it is a leaf).
- All the leaves appear on the same level.
- The leaf level forms a complete, ordered index of the associated data file.

9.11 Worst-Case Search Depth

It is important to have a quantitative understanding of the relationship between the page size of a B-tree, the number of keys to be stored in the tree, and the number of levels that the tree can extend. For example, you might know that you need to store 1 000 000 keys and that, given the nature of your storage hardware and the size of your keys, it is reasonable

to consider using a B-tree of order 512 (maximum of 511 keys per page). Given these two facts, you need to be able to answer the question: In the worst case, what will be the maximum number of disk accesses required to locate a key in the tree? This is the same as asking how deep the tree will be.

We can answer this question by noting that every key appears in the leaf level. Hence, we need to calculate the maximum height of a tree with 1 000 000 keys in the leaves.

Next we need to observe that we can use the formal definition of B-tree properties to calculate the *minimum* number of descendants that can extend from any level of a B-tree of some given order. This is of interest because we are interested in the *worst-case* depth of the tree. The worst case occurs when every page of the tree has only the minimum number of descendants. In such a case the keys are spread over a *maximal height* for the tree and a *minimal breadth*.

For a B-tree of order m , the minimum number of descendants from the root page is 2, so the second level of the tree contains only 2 pages. Each of these pages, in turn, has at least $\lceil m/2 \rceil$ descendants. The third level, then, contains

$$2 \times \lceil m/2 \rceil$$

pages. Since each of these pages, once again, has a minimum of $\lceil m/2 \rceil$ descendants, the general pattern of the relation between depth and the minimum number of descendants takes the following form:

Level	Minimum number of descendants
1 (root)	2
2	$2 \times \lceil m/2 \rceil$
3	$2 \times \lceil m/2 \rceil \times \lceil m/2 \rceil$ or $2 \times \lceil m/2 \rceil^2$
4	$2 \times \lceil m/2 \rceil^3$
...	...
d	$2 \times \lceil m/2 \rceil^{d-1}$

So, in general, for any level d of a B-tree, the *minimum* number of descendants extending from that level is

$$2 \times \lceil m/2 \rceil^{d-1}$$

For a tree with N keys in its leaves, we can express the relationship between keys and the minimum height d as

$$N \geq 2 \times \lceil m/2 \rceil^{d-1}$$

Solving for d , we arrive at the following expression:

$$d \leq 1 + \log_{\lceil m/2 \rceil} (N/2).$$

This expression gives us an *upper bound* for the depth of a B-tree with N keys. Let's find the upper bound for the hypothetical tree that we describe at the start of this section: a tree of order 512 that contains 1 000 000 keys. Substituting these specific numbers into the expression, we find that

$$d \leq 1 + \log_{256} 500\,000$$

or

$$d \leq 3.37$$

So we can say that given 1 000 000 keys, a B-tree of order 512 has a depth of no more than three levels.

9.12 Deletion, Merging, and Redistribution

Indexing 1 000 000 keys in no more than three levels of a tree is precisely the kind of performance we are looking for. As we have just seen, this performance is predicated on the B-tree properties we described earlier. In particular, the ability to guarantee that B-trees are broad and shallow rather than narrow and deep is coupled with the rules that state the following:

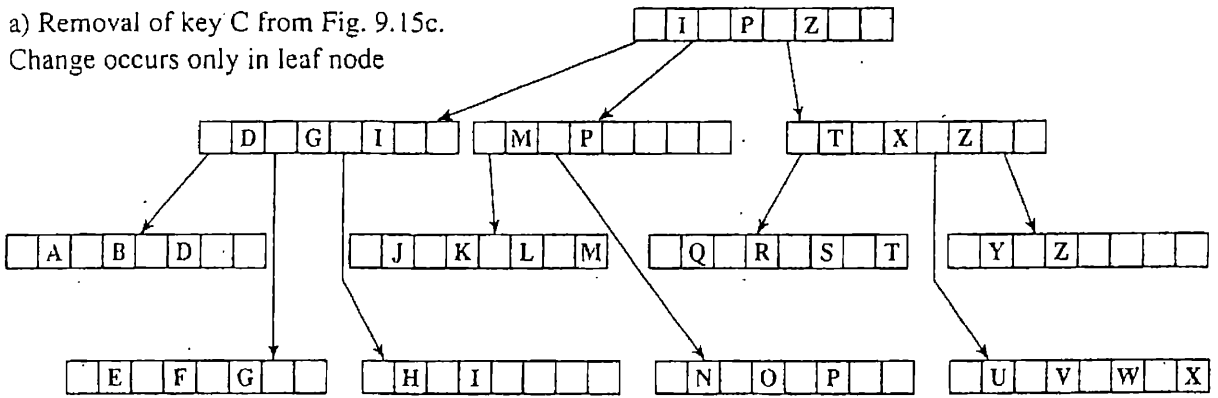
- Every page except for the root and the leaves has at least $\lceil m/2 \rceil$ descendants.
- A page contains at least $\lceil m/2 \rceil$ keys and no more than m keys.

We have already seen that the process of page splitting guarantees that these properties are maintained when new keys are inserted into the tree. We need to develop some kind of equally reliable guarantee that these properties are maintained when keys are *deleted* from the tree.

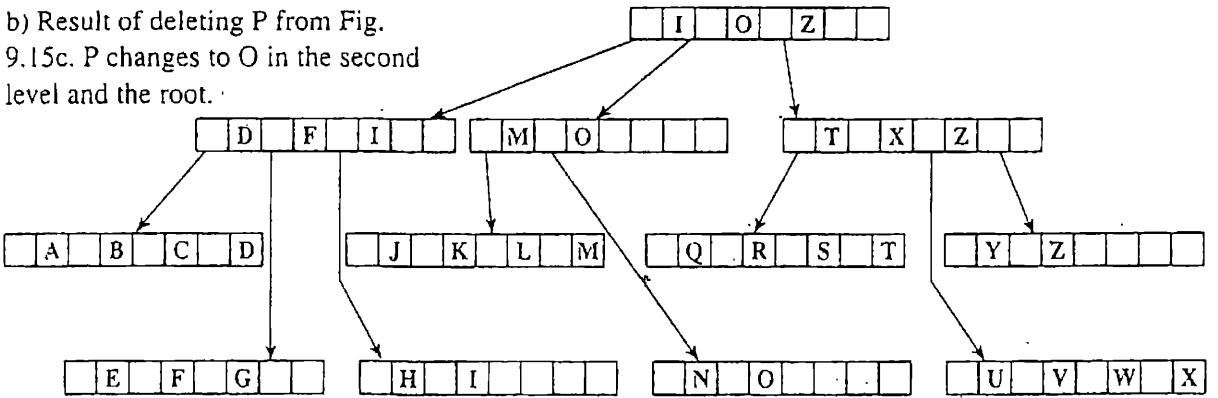
Working through some simple deletion situations by hand helps us demonstrate that the deletion of a key can result in several different situations. We start with the B-tree of Fig. 9.15(c) that contains all the letters of the alphabet. Consider what happens when we try to delete some of its keys.

The simplest situation is illustrated in the result of deleting key C in Fig. 9.21(a). Deleting the key from the first leaf node does not cause an

a) Removal of key C from Fig. 9.15c.
Change occurs only in leaf node



b) Result of deleting P from Fig. 9.15c. P changes to O in the second level and the root.



c) Result of deleting H from Fig. 9.15c. Removal of H caused an underflow, and two leaf nodes were merged.

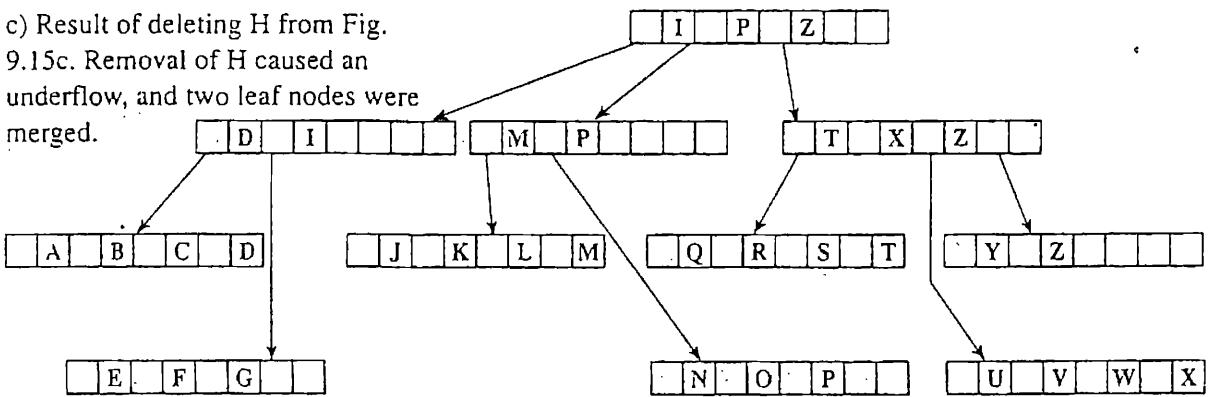


Figure 9.21 Three situations that can occur during deletions.

underflow in the node and does not change its largest value. Consequently, deletion involves nothing more than removing the key from the node.

Deleting the P in Fig. 9.21(b) is more complicated. Removal of P from the second leaf node does not cause underflow, but it does change the largest key in the node. Hence, the second-level node must be modified to reflect this change. The key to the second leaf node becomes O , and the second-level node must be modified so that it contains O instead of P . Since P was the largest key in the second node in the second level, the root node must also have key P replaced by O .

Deleting the H in Fig 9.21(c) causes an underflow in the third leaf node. After H is deleted, the last remaining key in the node, I , is inserted into the neighbor node, and the third leaf node is deleted. Since the second leaf node has only three keys, there is room for the key I in that node. This illustrates a more general *merge* operation. After the merge, the second-level node is modified to reflect the current status of the leaf nodes.

Merging and other modifications can propagate to the root of the B-tree. If the root ends up with only one key and one child, it can be eliminated. Its sole child node becomes the new root of the tree and the tree gets shorter by one level.

The rules for deleting a key k from a node n in a B-tree are as follows:

1. If n has more than the minimum number of keys and the k is not the largest in n , simply delete k from n .
2. If n has more than the minimum number of keys and the k is the largest in n , delete k and modify the higher level indexes to reflect the new largest key in n .
3. If n has exactly the minimum number of keys and one of the siblings of n has few enough keys, merge n with its sibling and delete a key from the parent node.
4. If n has exactly the minimum number of keys and one of the siblings of n has extra keys, redistribute by moving some keys from a sibling to n , and modify the higher level indexes to reflect the new largest keys in the affected nodes.

Rules 3 and 4 include references to “few enough keys” to allow merging and “extra keys” to allow redistribution. These are not exclusive rules, and the implementation of delete is allowed to choose which rule to use when they are both applicable. Look at the example of an order five tree in Fig. 9.22, and consider deleting keys C , M , and W . Since three is the minimum number of keys, deleting any of these keys requires some adjustment of the leaf nodes. In the case of deleting C , the only sibling node has three

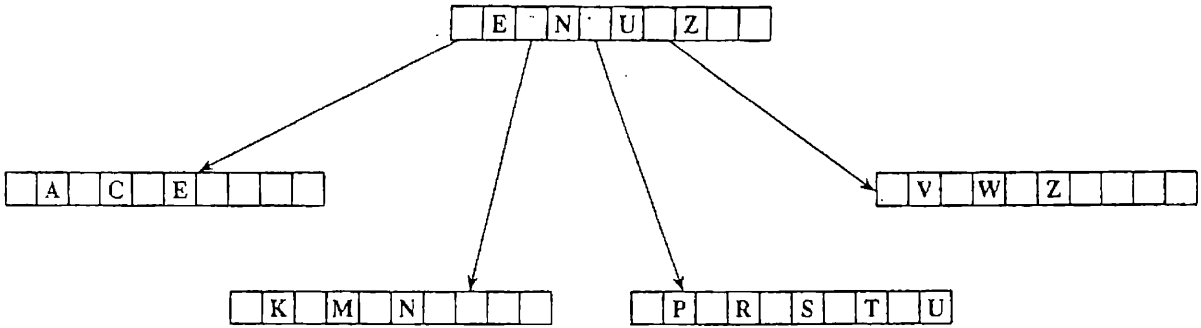


Figure 9.22 Example of order five B-tree. Consider delete of keys C, M, and W.

(b)

keys. After deleting C, there are five keys in the two sibling nodes, so a merge is allowed. No redistribution is possible because the sibling node has the minimum number of keys. In the case of deleting W, the only sibling has five keys, so one or two of the keys can be moved to the underfull node. No merge is possible here, since there are seven keys remaining in the two sibling nodes—too many for a single node. In the case of deleting M, there are two options: merge with the left sibling or redistribute keys in the right sibling.

9.12.1 Redistribution

Unlike merge, which is a kind of reverse split, redistribution is a new idea. Our insertion algorithm does not require operations analogous to redistribution.

Redistribution differs from both splitting and merging in that it never causes the collection of nodes in the tree to change. It is guaranteed to have strictly local effects. Note that the term *sibling* implies that the pages have the same parent page. If there are two nodes at the leaf level that are logically adjacent but do not have the same parent—for example, HI and JKLM in the tree of Fig. 9.22(a)—these nodes are not siblings. Redistribution algorithms are generally written so they do not consider moving keys between nodes that are not siblings, even when they are logically adjacent. Can you see the reasoning behind this restriction?

Another difference between redistribution on the one hand and merging and splitting on the other is that there is no necessary, fixed prescription for how the keys should be rearranged. A single deletion in a properly formed B-tree cannot cause an underflow of more than one key. Therefore, redistribution can restore the B-tree properties by moving only one key from a sibling into the page that has underflowed, even if the

distribution of the keys between the pages is very uneven. Suppose, for example, that we are managing a B-tree of order 101. The minimum number of keys that can be in a page is 50; the maximum is 100. Suppose we have one page that contains the minimum and a sibling that contains the maximum. If a key is deleted from the page containing 50 keys, an underflow condition occurs. We can correct the condition through redistribution by moving one key, 50 keys, or any number of keys between 1 and 50. The usual strategy is to divide the keys as evenly as possible between the pages. In this instance that means moving 25 keys.

9.13 Redistribution During Insertion: A Way to Improve Storage Utilization

As you may recall, B-tree insertion does not require an operation analogous to redistribution; splitting is able to account for all instances of overflow. This does not mean, however, that it is not *desirable* to use redistribution during insertion as an option, particularly since a set of B-tree maintenance algorithms must already include a redistribution procedure to support deletion. Given that a redistribution procedure is already present, what advantage might we gain by using it as an alternative to node splitting?

Redistribution during insertion is a way of avoiding, or at least postponing, the creation of new pages. Rather than splitting a full page and creating two approximately half-full pages, redistribution lets us place some of the overflowing keys into another page. The use of redistribution in place of splitting should therefore tend to make a B-tree more efficient in its utilization of space.

It is possible to quantify this efficiency of space usage by viewing the amount of space used to store information as a percentage of the total amount of space required to hold the B-tree. After a node splits, each of the two resulting pages is about half full. So, in the worst case, space utilization in a B-tree using two-way splitting is around 50 percent. Of course, the actual degree of space utilization is better than this worst-case figure. Yao (1978) has shown that, for large trees of relatively large order, space utilization approaches a theoretical average of about 69 percent if insertion is handled through two-way splitting.

The idea of using redistribution as an alternative to splitting when possible, splitting a page only when both of its siblings are full, is

introduced in Bayer and McCreight's original paper (1972): The paper includes some experimental results that show that two-way splitting results in a space utilization of 67 percent for a tree of order 121 after five thousand random insertions. When the experiment was repeated, using redistribution when possible, space utilization increased to over 86 percent. Subsequent empirical testing by students at Oklahoma State University using B-trees of order 49 and 303 also resulted in space utilization exceeding 85 percent when redistribution was used. These findings and others suggest that any serious application of B-trees to even moderately large files should implement insertion procedures that handle overflow through redistribution when possible.

9.14 B* Trees

In his review and amplification of work on B-trees in 1973, Knuth (1998) extends the notion of redistribution during insertion to include new rules for splitting. He calls the resulting variation on the fundamental B-tree form a B^* tree.

Consider a system in which we are postponing splitting through redistribution, as outlined in the preceding section. If we are considering any page other than the root, we know that when it is finally time to split, the page has at least one sibling that is also full. This opens up the possibility of a two-to-three split rather than the usual one-to-two or two-way split.

The important aspect of this two-to-three split is that it results in pages that are each about two-thirds full rather than just half full. This makes it possible to define a new kind of B-tree, called a B^* tree, which has the following properties:

1. Every page has a maximum of m descendants.
2. Every page except for the root has at least $\lceil (2m - 1)/3 \rceil$ descendants.
3. The root has at least two descendants (unless it is a leaf).
4. All the leaves appear on the same level.

The critical changes between this set of properties and the set we define for a conventional B-tree are in rule 2: a B^* tree has pages that contain a minimum $\lceil (2m - 1)/3 \rceil$ keys. This new property, of course, affects procedures for deletion and redistribution.

To implement B^* tree procedures, one must also deal with the question of splitting the root, which, by definition, never has a sibling. If there

is no sibling, no two-to-three split is possible. Knuth suggests allowing the root to grow to a size larger than the other pages so, when it does split, it can produce two pages that are each about two-thirds full. This has the advantage of ensuring that all pages below the root level adhere to B* tree characteristics. However, it has the disadvantage of requiring that the procedures be able to handle a page that is larger than all the others. Another solution is to handle the splitting of the root as a conventional one-to-two split. This second solution avoids any special page-handling logic. On the other hand, it complicates deletion, redistribution, and other procedures that must be sensitive to the minimum number of keys allowed in a page. Such procedures would have to be able to recognize that pages descending from the root might legally be only half full.

9.15 Buffering of Pages: Virtual B-Trees

We have seen that the B-tree can be a very efficient, flexible storage structure that maintains its balanced properties after repeated deletions and insertions and that provides access to any key with just a few disk accesses. However, focusing on just the structural aspects, as we have so far, can cause us inadvertently to overlook ways of using this structure to full advantage. For example, the fact that a B-tree has a depth of three levels does not at all mean that we need to do three disk accesses to retrieve keys from pages at the leaf level. We can do much better than that.

Obtaining better performance from B-trees involves looking in a precise way at our original problem. We needed to find a way to make efficient use of indexes that are too large to be held *entirely* in memory. Up to this point we have approached this problem in an all-or-nothing way: an index has been held entirely in memory, organized as a list or binary tree, or accessed entirely on secondary store, using a B-tree structure. But, stating that we cannot hold *all* of an index in memory does not imply that we cannot hold *some* of it there. In fact, our implementation of class BTree is already keeping the root in memory at all times and keeping a full branch in memory during insertion and deletion.

For example, assume that we have an index containing 1 megabyte of records and that we cannot reasonably use more than 256 kilobytes of memory for index storage at any given time. Given a page size of 4 kilobytes, holding around 64 keys per page, our B-tree can be contained in three levels. We can reach any one of our keys in no more than two disk

accesses. That is certainly acceptable, but why should we settle for this kind of performance? Why not try to find a way to bring the average number of disk accesses per search down to one disk access or less?

If we're thinking of the problem strictly in terms of physical storage structures, retrieval averaging one disk access or less sounds impossible. But remember, our objective was to find a way to manage our megabyte of index within 256 kilobytes of memory, not within the 4 kilobytes required to hold a single page of our tree.

The simple, keep-the-root strategy we have been using suggests an important, more general approach: rather than just holding the root page in memory, we can create a *page buffer* to hold some number of B-tree pages, perhaps five, ten, or more. As we read pages in from the disk in response to user requests, we fill up the buffer. Then, when a page is requested, we access it from memory if we can, thereby avoiding a disk access. If the page is not in memory, then we read it into the buffer from secondary storage, replacing one of the pages that was previously there. A B-tree that uses a memory buffer in this way is sometimes referred to as a *virtual B-tree*.

For our implementation, we can use the `Nodes` member and the `Fetch` and `Store` methods to manage this page buffer. `Fetch` and `Store` can keep track of which nodes are in memory and avoid the disk read or write whenever possible. This modification is included as an exercise.

9.15.1 LRU Replacement

Clearly, such a buffering scheme works only if we are more likely to request a page that is in the buffer than one that is not. The process of accessing the disk to bring in a page that is *not* already in the buffer is called a *page fault*. There are two causes of page faults:

1. We have never used the page.
2. It was once in the buffer but has since been replaced with a new page.

The first cause of page faults is unavoidable: if we have not yet read in and used a page, there is no way it can already be in the buffer. But the second cause is one we can try to minimize through buffer management. The critical management decision arises when we need to read a new page into a buffer that is already full: which page do we decide to replace?

One common approach is to replace the page that was least recently used; this is called *LRU* replacement. Note that this is different from

replacing the page that was *read into* the buffer least recently. Instead, the LRU method keeps track of the *requests* for pages. The page to be replaced is the one that has gone the longest time without a request for use.

Some research by Webster (1980) shows the effect of increasing the number of pages that can be held in the buffer area under an LRU replacement strategy. Table 9.1 summarizes a small but representative portion of Webster's results. It lists the average number of disk accesses per search given different numbers of page buffers. These results are obtained using a simple LRU replacement strategy without accounting for page height. Keeping less than 15 percent of the tree in memory (20 pages out of the total 140) reduces the average number of accesses per search to less than one.

Note that the decision to use LRU replacement is based on the assumption that we are more likely to need a page that we have used recently than we are to need a page that we have never used or one that we used some time ago. If this assumption is not valid, then there is absolutely no reason to retain preferentially pages that were used recently. The term for this kind of assumption is *temporal locality*. We are assuming that there is a kind of *clustering* of the use of certain pages over time. The hierarchical nature of a B-tree makes this kind of assumption reasonable.

For example, during redistribution after overflow or underflow, we access a page and then access its sibling. Because B-trees are hierarchical, accessing a set of sibling pages involves repeated access to the parent page in rapid succession. This is an instance of temporal locality; it is easy to see how it is related to the tree's hierarchy.

9.15.2 Replacement Based on Page Height

There is another, more direct way to use the hierarchical nature of the B-tree to guide decisions about page replacement in the buffers. Our simple, keep-the-root strategy exemplifies this alternative: always retain the pages that occur at the highest levels of the tree. Given a larger amount of buffer

Table 9.1 Effect of using more buffers with a simple LRU replacement strategy.

Buffer Count	1	5	10	20
Average Accesses per Search	3.00	1.71	1.42	0.97
Number of keys = 2400				
Total pages = 140				
Tree height = 3 levels				

space, it might be possible to retain not only the root, but also all of the pages at the second level of a tree.

Let's explore this notion by returning to a previous example in which we have access to 256 kilobytes of memory and a 1-megabyte index. Since our page size is 4 kilobytes, we could build a buffer area that holds 64 pages within the memory area. Assume that our 1 megabyte worth of index requires around 1.2 megabytes of storage on disk (storage utilization = 83 percent). Given the 4-kilobyte page size, this 1.2 megabytes requires slightly more than 300 pages. We assume that, on the average, each of our pages has around 30 descendants. It follows that our three-level tree has, of course, a single page at the root level, followed by 9 or 10 pages at the second level, with all the remaining pages at the leaf level. Using a page replacement strategy that always retains the higher-level pages, it is clear that our 64-page buffer eventually contains the root page and all the pages at the second level. The approximately 50 remaining buffer slots are used to hold leaf-level pages. Decisions about which of these pages to replace can be handled through an LRU strategy. It is easy to see how, given a sizable buffer, it is possible to bring the average number of disk accesses per search down to a number that is less than one.

Webster's research (1980) also investigates the effect of taking page height into account, giving preference to pages that are higher in the tree when it comes time to decide which pages to keep in the buffers. Augmenting the LRU strategy with a weighting factor that accounts for page height reduces the average number of accesses, given a 10-page buffer, from 1.42 accesses per search down to 1.12 accesses per search.

9.15.3 Importance of Virtual B-Trees

It is difficult to overemphasize the importance of including a page buffering scheme with any implementation of a B-tree index structure. Because the B-tree structure is so interesting and powerful, it is easy to fall into the trap of thinking that the B-tree organization is itself a sufficient solution to the problem of accessing large indexes that must be maintained on secondary storage. As we have emphasized, to fall into that trap is to lose sight of the original problem: to find a way to *reduce* the amount of memory required to handle large indexes. We did not, however, need to reduce the amount of memory to the amount required for a single index page. It is usually possible to find enough memory to hold a number of pages. Doing so can dramatically increase system performance.

9.16 Variable-Length Records and Keys

In many applications the information associated with a key varies in length. Secondary indexes that reference inverted lists are an excellent example of this. One way to handle this variability is to place the associated information in a separate, variable-length record file; the B-tree would contain a reference to the information in this other file. Another approach is to allow a variable number of keys and records in a B-tree page.

Up to this point we have regarded B-trees as being of some order m . Each page has a fixed maximum and minimum number of keys that it can legally hold. The notion of a variable-length record and, therefore, a variable number of keys per page is a significant departure from the point of view we have developed so far. A B-tree with a variable number of keys per page clearly has no single, fixed order.

The variability in length can also extend to the keys as well as to entire records. For example, in a file in which people's names are the keys, we might choose to use only as much space as required for a name rather than allocate a fixed-size field for each key. As we saw in earlier chapters, implementing a structure with variable-length fields can allow us to put many more names in a given amount of space since it eliminates internal fragmentation. If we can put more keys in a page, then we have a larger number of descendants from a page and very probably a tree with fewer levels.

Accommodating this variability in length means using a different kind of page structure. We look at page structures appropriate for use with variable-length keys in detail in the next chapter. We also need a different criterion for deciding when a page is full and when it is in an underflow condition. Rather than use a maximum and minimum number of keys per page, we need to use a maximum and minimum number of bytes.

Once the fundamental mechanisms for handling variable-length keys or records are in place, interesting new possibilities emerge. For example, we might consider the notion of biasing the splitting and redistribution methods so that the shortest variable-length keys are promoted upward in preference to longer keys. The idea is that we want to have pages with the largest numbers of descendants up high in the tree, rather than at the leaf level. Branching out as broadly as possible as high as possible in the tree tends to reduce the overall height of the tree. McCreight (1977) explores this notion in the article, "Pagination of B* Trees with Variable-Length Records."

The principal point we want to make with these examples of variations on B-tree structures is that this chapter introduces only the most basic forms of this very useful, flexible file structure. Implementations of B-trees do not slavishly follow the textbook form of B-trees. Instead, they use many of the other organizational techniques we study in this book, such as variable-length record structures in combination with the fundamental B-tree organization to make new, special-purpose file structures uniquely suited to the problems at hand.

S U M M A R Y

We begin this chapter by picking up the problem we left unsolved at the end of Chapter 7: simple, linear indexes work well if they are held in memory, but they are expensive to maintain and search if they are so big that they must be held on secondary storage. The expense of using secondary storage is most evident in two areas:

- Sorting of the index; and
- Searching, since even binary searching requires more than two or three disk accesses.

We first address the question of structuring an index so it can be kept in order without sorting. We use tree structures to do this, discovering that we need a *balanced* tree to ensure that the tree does not become overly deep after repeated random insertions. We see that AVL trees provide a way of balancing a binary tree with only a small amount of overhead.

Next we turn to the problem of reducing the number of disk accesses required to search a tree. The solution to this problem involves dividing the tree into pages so a substantial portion of the tree can be retrieved with a single disk access. Paged indexes let us search through very large numbers of keys with only a few disk accesses.

Unfortunately, we find that it is difficult to combine the idea of *paging* of tree structures with the *balancing* of these trees by AVL methods. The most obvious evidence of this difficulty is associated with the problem of selecting the members of the root page of a tree or subtree when the tree is built in the conventional top-down manner. This sets the stage for introducing Bayer and McCreight's work on B-trees, which solves the paging and balancing dilemma by starting from the leaf level, promoting keys upward as the tree grows.

Our discussion of B-trees begins by emphasizing the multilevel index approach. We include a full implementation of insertion and searching and examples of searching, insertion, overflow detection, and splitting to show how B-trees grow while maintaining balance in a paged structure. Next we formalize our description of B-trees. This formal definition permits us to develop a formula for estimating worst-case B-tree depth. The formal description also motivates our work on developing deletion procedures that maintain the B-tree properties when keys are removed from a tree.

Once the fundamental structure and procedures for B-trees are in place, we begin refining and improving on these ideas. The first set of improvements involves increasing the storage utilization within B-trees. Of course, increasing storage utilization can also result in a decrease in the height of the tree and therefore in improvements in performance. We sometimes find that by redistributing keys during insertion rather than splitting pages, we can improve storage utilization in B-trees so it averages around 85 percent. Carrying our search for increased storage efficiency even further, we find that we can combine redistribution during insertion with a different kind of splitting to ensure that the pages are about two-thirds full rather than only half full after the split. Trees using this combination of redistribution and two-to-three splitting are called B^* trees.

Next we turn to the matter of buffering pages, creating a *virtual B-tree*. We note that the use of memory is not an all-or-nothing choice: indexes that are too large to fit into memory do not have to be accessed *entirely* from secondary storage. If we hold pages that are likely to be reused in memory, then we can save the expense of reading these pages in from the disk again. We develop two methods of guessing which pages are to be reused. One method uses the height of the page in the tree to decide which pages to keep. Keeping the root has the highest priority, the root's descendants have the next priority, and so on. The second method for selecting pages to keep in memory is based on recentness of use: we always replace the least recently used (LRU) page and retain the pages used most recently. We see that it is possible to combine these methods and that doing so can result in the ability to find keys while using an average of less than one disk access per search.

We close the chapter with a brief look at the use of variable-length records within the pages of a B-tree, noting that significant savings in space and consequent reduction in the height of the tree can result from the use of variable-length records. The modification of the basic textbook B-tree definition to include the use of variable-length records is just one

example of the many variations on B-trees that are used in real-world implementations.

KEY TERMS

AVL tree. A height-balanced (HB(1)) binary tree in which insertions and deletions can be performed with minimal accesses to local nodes. AVL trees are interesting because they keep branches from getting overly long after many random insertions.

B-tree of order m . A multilevel index tree with these properties:

- 0 Every node has a maximum of m descendants.
- 1 Every node except the root has at least $\lceil m/2 \rceil$ descendants.
- 2 The root has at least two descendants (unless it is a leaf).
- 3 All of the leaves appear on the same level.

B-trees are built upward from the leaf level, so creation of new pages always starts at the leaf level.

The power of B-trees lies in the facts that they are balanced (no overly long branches); they are shallow (requiring few seeks); they accommodate random deletions and insertions at a relatively low cost while remaining in balance; and they guarantee at least 50 percent storage utilization.

B* tree. A special B-tree in which each node is at least two-thirds full. B* trees generally provide better storage utilization than B-trees.

Height-balanced tree. A tree structure with a special property: for each node there is a limit to the amount of difference that is allowed among the heights of any of the node's subtrees. An $HB(k)$ tree allows subtrees to be k levels out of balance. (See *AVL tree*.)

Leaf of a B-tree. A page at the lowest level in a B-tree. All leaves in a B-tree occur at the same level.

Merging. When a B-tree node underflows (becomes less than 50 percent full), it sometimes becomes necessary to combine the node with an adjacent node, thus decreasing the total number of nodes in the tree. Since merging involves a change in the number of nodes in the tree, its effects can require reorganization at many levels of the tree.

Order of a B-tree. The maximum number of descendants that a node in the B-tree can have.

Paged index. An index that is divided into blocks, or pages, each of which can hold many keys. The use of paged indexes allows us to search through very large numbers of keys with only a few disk accesses.

Redistribution. When a B-tree node underflows (becomes less than 50 percent full), it may be possible to move keys into the node from an adjacent node with the same parent. This helps ensure that the 50 percent-full property is maintained. When keys are redistributed, it becomes necessary to alter the contents of the parent as well. Redistribution, as opposed to *merging*, does not involve creation or deletion of nodes—its effects are entirely local. Often redistribution can also be used as an alternative to splitting.

Splitting. Creation of two nodes out of one when the original node becomes overfull. Splitting results in the need to promote a key to a higher-level node to provide an index separating the two new nodes.

Virtual B-tree. A B-tree index in which several pages are kept in memory in anticipation of the possibility that one or more of them will be needed by a later access. Many different strategies can be applied to replacing pages in memory when virtual B-trees are used, including the least-recently-used strategy and height-weighted strategies.

FURTHER READINGS

Currently available textbooks on file and data structures contain surprisingly brief discussions on B-trees. These discussions do not, in general, add substantially to the information presented in this chapter and the following chapter. Consequently, readers interested in more information about B-trees must turn to the articles that have appeared in journals over the past 15 years.

The article that introduced B-trees to the world is Bayer and McCreight's "Organization and Maintenance of Large Ordered Indexes" (1972). It describes the theoretical properties of B-trees and includes empirical results concerning, among other things, the effect of using redistribution in addition to splitting during insertion. Readers should be aware that the notation and terminology used in this article differ from those used in this text in a number of important respects.

Comer's (1979) survey article, "The Ubiquitous B-tree," provides an excellent overview of some important variations on the basic B-tree form. Knuth's (1998) discussion of B-trees, although brief, is an important

resource in part because many of the variant forms such as B* trees were first collected together in Knuth's discussion. McCreight (1977) looks specifically at operations on trees that use variable-length records and that are therefore of variable order. Although this article speaks specifically about B* trees, the consideration of variable-length records can be applied to many other B-tree forms. In "Time and Space Optimality on B-trees," Rosenberg and Snyder (1981) analyze the effects of initializing B-trees with the minimum number of nodes. In "Analysis of Design Alternatives for Virtual Memory Indexes," Murayama and Smith (1977) look at three factors that affect the cost of retrieval: choice of search strategy, whether pages in the index are structured, and whether keys are compressed. Gray and Reuter (1993) provide an analysis of issues in B-tree implementation. Zoellick (1986) discusses the use of B-tree—like structures on optical discs.

Since B-trees in various forms have become a standard file organization for databases, a good deal of interesting material on applications of B-trees can be found in the database literature. Held and Stonebraker (1978), Snyder (1978), Kroenke (1998), and Elmasri and Navathe (1994) discuss the use of B-trees in database systems generally. Ullman (1986) covers the problem of dealing with applications in which several programs have access to the same database concurrently and identifies literature concerned with concurrent access to B-tree.

Uses of B-trees for secondary key access are covered in many of the previously cited references. There is also a growing literature on multidimensional dynamic indexes, including variants of the B-tree, *k-d* B-tree and R trees. *K-d* B-trees are described in papers by Ouskel and Scheuermann (1981) and Robinson (1981). R trees support multidimensional queries, so-called *range queries*, and were first described in Guttman (1984) and further extended in Sellis et al (1987), Beckmann et al (1990), and Kamel and Floutsos (1992). Shaffer (1997) and Standish (1995) include extensive coverage of a variety of tree structures. Other approaches to secondary indexing include the use of *tries* and *grid files*. Tries are covered in many texts on files and data structures, including Knuth (1998) and Loomis (1989). Grid files are covered thoroughly in Nievergelt et al. (1984).

An interesting early paper on the use of dynamic tree structures for processing files is "The Use of Tree Structures for Processing Files," by Sussenguth (1963). Wagner (1973) and Keehn and Lacy (1974) examine the index design considerations that led to the development of VSAM. VSAM uses an index structure very similar to a B-tree but appears to have

been developed independently of Bayer and McCreight's work. Readers interested in learning more about AVL trees should read Knuth (1998), who takes a more rigorous, mathematical look at AVL tree operations and properties.

EXERCISES

1. Balanced binary trees can be effective index structures for memory-based indexing, but they have several drawbacks when they become so large that part or all of them must be kept on secondary storage. The following questions should help bring these drawbacks into focus and thus reinforce the need for an alternative structure such as the B-tree.
 - a. There are two major problems with using binary search to search a simple sorted index on secondary storage: the number of disk accesses is larger than we would like, and the time it takes to keep the index sorted is substantial. Which of the problems does a binary search tree alleviate?
 - b. Why is it important to keep search trees balanced?
 - c. In what way is an AVL tree better than a simple binary search tree?
 - d. Suppose you have a file with 1 000 000 keys stored on disk in a completely full, balanced binary search tree. If the tree is not paged, what is the maximum number of accesses required to find a key? If the tree is paged in the manner illustrated in Fig. 9.12, but with each page able to hold 15 keys and to branch to 16 new pages, what is the maximum number of accesses required to find a key? If the page size is increased to hold 511 keys with branches to 512 nodes, how does the maximum number of accesses change?
 - e. Consider the problem of balancing the three-key-per-page tree in Fig. 9.13 by rearranging the pages. Why is it difficult to create a tree-balancing algorithm that has only local effects? When the page size increases to a more likely size (such as 512 keys), why does it become difficult to guarantee that each of the pages contains at least some minimum number of keys?
 - f. Explain the following statement: B-trees are built upward from the bottom, whereas binary trees are built downward from the top.
 - g. Although B-trees are generally considered superior to binary search trees for external searching, binary trees are still commonly used for internal searching. Why is this so?

2. Show the B-trees of order four that result from loading the following sets of keys in order:
 - a. C G J X
 - b. C G J X N S U O A E B H I
 - c. C G J X N S U O A E B H I F
 - d. C G J X N S U O A E B H I F K L Q R T V U W Z
3. Given a B-tree of order 256,
 - a. What is the maximum number of descendants from a page?
 - b. What is the minimum number of descendants from a page (excluding the root and leaves)?
 - c. What is the minimum number of descendants from the root?
 - d. What is the maximum depth of the tree if it contains 100 000 keys?
4. Using a method similar to that used to derive the formula for worst-case depth, derive a formula for best-case, or minimum, depth for an order m B-tree with N keys. What is the minimum depth of the tree described in the preceding question?
5. Suppose you have a B-tree index for an unsorted file containing N data records, where each key has stored with it the RRN of the corresponding record. The depth of the B-tree is d . What are the maximum and minimum numbers of disk accesses required to
 - a. Retrieve a record?
 - b. Add a record?
 - c. Delete a record?
 - d. Retrieve all records from the file in sorted order?

Assume that page buffering is *not* used. In each case, indicate how you arrived at your answer.
6. Show the trees that result after each of the keys N, P, Q, and Y is deleted from the B-tree of Figure 9.15(c).
7. A common belief about B-trees is that a B-tree cannot grow deeper unless it is 100 percent full. Discuss this.
8. Suppose you want to delete a key from a node in a B-tree. You look at the right sibling and find that redistribution does not work; merging would be necessary. You look to the left and see that redistribution is an option here. Do you choose to merge or redistribute?

9. What is the difference between a B* tree and a B-tree? What improvement does a B* tree offer over a B-tree, and what complications does it introduce? How does the minimum depth of an order m B* tree compare with that of an order m B-tree?
10. What is a virtual B-tree? How can it be possible to average fewer than one access per key when retrieving keys from a three-level virtual B-tree? Write a description for an LRU replacement scheme for a ten-page buffer used in implementing a virtual B-tree.
11. Discuss the trade-offs between storing the information indexed by the keys in a B-tree with the key and storing the information in a separate file.
12. We noted that, given variable-length keys, it is possible to optimize a tree by building in a bias toward promoting shorter keys. With fixed-order trees we promote the middle key. In a variable-order, variable-length key tree, what is the meaning of “middle key”? What are the trade-offs associated with building in a bias toward shorter keys in this selection of a key for promotion? Outline an implementation for this selection and promotion process.

PROGRAMMING EXERCISES

13. Implement the Delete method of class BTree.
14. Modify classes BTreeNode and BTree to have one more reference than key in each interior node.
15. Write an interactive program that allows a user to find, insert, and delete keys from a B-tree.
16. Write a B-tree program that uses keys that are strings rather than single characters.
17. Write a program that builds a B-tree index for a data file in which records contain more information than just a key. Use the Person, Recording, Ledger, or Transaction files from previous chapters.
18. Implement B* trees by modifying class BTree.

PROGRAMMING PROJECT

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

19. Use class `BTree` to create a B-tree index of a student record file with the student identifier as key. Write a driver program to create a B-tree file from an existing student record file.
20. Use class `BTree` to create a B-tree index of a course registration record file with the student identifier as key. Write a driver program to create a B-tree file from an existing course registration record file.
21. Write a program that opens a B-tree indexed student file and a B-tree indexed course registration file and retrieves information on demand. Prompt a user for a student identifier, and print all objects that match it.

The next part of the programming project is in Chapter 10.