

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>

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

11.3 Hashing Functions and Record Distributions	472
11.3.1 Distributing Records among Addresses	472
11.3.2 Some Other Hashing Methods	473
11.3.3 Predicting the Distribution of Records	475
11.3.4 Predicting Collisions for a Full File	479
11.4 How Much Extra Memory Should Be Used?	480
11.4.1 Packing Density	481
11.4.2 Predicting Collisions for Different Packing Densities	481
11.5 Collision Resolution by Progressive Overflow	485
11.5.1 How Progressive Overflow Works	485
11.5.2 Search Length	487
11.6 Storing More Than One Record per Address: Buckets	490
11.6.1 Effects of Buckets on Performance	491
11.6.2 Implementation Issues	496
11.7 Making Deletions	498
11.7.1 Tombstones for Handling Deletions	499
11.7.2 Implications of Tombstones for Insertions	500
11.7.3 Effects of Deletions and Additions on Performance	501
11.8 Other Collision Resolution Techniques	502
11.8.1 Double Hashing	502
11.8.2 Chained Progressive Overflow	502
11.8.3 Chaining with a Separate Overflow Area	505
11.8.4 Scatter Tables: Indexing Revisited	506
11.9 Patterns of Record Access	507
Summary	508
Key Terms	512
Further Readings	514
Exercises	515
Programming Exercises	520

Chapter 12 Extendible Hashing

523

12.1 Introduction	524
12.2 How Extendible Hashing Works	525
12.2.1 Tries	525
12.2.2 Turning the Trie into a Directory	526
12.2.3 Splitting to Handle Overflow	528
12.3 Implementation	530
12.3.1 Creating the Addresses	530
12.3.2 Classes for Representing Bucket and Directory Objects	533
12.3.3 Bucket and Directory Operations	536
12.3.4 Implementation Summary	542
12.4 Deletion	543
12.4.1 Overview of the Deletion Process	543

Hashing

CHAPTER OBJECTIVES

- ❖ Introduce the concept of *hashing*.
- ❖ Examine the problem of choosing a good *hashing algorithm*, present a reasonable one in detail, and describe some others.
- ❖ Explore three approaches for *reducing collisions*: randomization of addresses, use of extra memory, and storage of several records per address.
- ❖ Develop and use mathematical tools for analyzing performance differences resulting from the use of different hashing techniques.
- ❖ Examine problems associated with *file deterioration* and discuss some solutions.
- ❖ Examine effects of *patterns of record access* on performance.

CHAPTER OUTLINE

- 11.1 Introduction**
 - 11.1.1 What Is Hashing?
 - 11.1.2 Collisions
- 11.2 A Simple Hashing Algorithm**
- 11.3 Hashing Functions and Record Distributions**
 - 11.3.1 Distributing Records among Addresses
 - 11.3.2 Some Other Hashing Methods
 - 11.3.3 Predicting the Distribution of Records
 - 11.3.4 Predicting Collisions for a Full File
- 11.4 How Much Extra Memory Should Be Used?**
 - 11.4.1 Packing Density
 - 11.4.2 Predicting Collisions for Different Packing Densities
- 11.5 Collision Resolution by Progressive Overflow**
 - 11.5.1 How Progressive Overflow Works
 - 11.5.2 Search Length
- 11.6 Storing More Than One Record per Address: Buckets**
 - 11.6.1 Effects of Buckets on Performance
 - 11.6.2 Implementation Issues
- 11.7 Making Deletions**
 - 11.7.1 Tombstones for Handling Deletions
 - 11.7.2 Implications of Tombstones for Insertions
 - 11.7.3 Effects of Deletions and Additions on Performance
- 11.8 Other Collision Resolution Techniques**
 - 11.8.1 Double Hashing
 - 11.8.2 Chained Progressive Overflow
 - 11.8.3 Chaining with a Separate Overflow Area
 - 11.8.4 Scatter Tables: Indexing Revisited
- 11.9 Patterns of Record Access**

11.1 Introduction

$O(1)$ access to files means that no matter how big the file grows, access to a record always takes the same, small number of seeks. By contrast, sequential searching gives us $O(N)$ access, wherein the number of seeks grows in proportion to the size of the file. As we saw in the preceding chapters, B-trees improve on this greatly, providing $O(\log_k N)$ access; the number of seeks increases as the logarithm to the base k of the number of records, where k is a measure of the leaf size. $O(\log_k N)$ access can provide

very good retrieval performance, even for very large files, but it is still not $O(1)$ access.

In a sense, $O(1)$ access has been the Holy Grail of file structure design. Everyone agrees that $O(1)$ access is what we want to achieve, but until about ten years ago, it was not clear if one could develop a general class of $O(1)$ access strategies that would work on dynamic files that change greatly in size.

In this chapter we begin with a description of static hashing techniques. They provide us with $O(1)$ access but are not extensible as the file increases in size. Static hashing was the state of the art until about 1980. In the following chapter we show how research and design work during the 1980s found ways to extend hashing, and $O(1)$ access, to files that are dynamic and increase in size over time.

11.1.1 What Is Hashing?

A *hash function* is like a black box that produces an address every time you drop in a key. More formally, it is a function $h(K)$ that transforms a key K into an address. The resulting address is used as the basis for storing and retrieving records. In Fig. 11.1, the key LOWELL is transformed by the hash function to the address 4. That is, $h(\text{LOWELL}) = 4$. Address 4 is said to be the *home address* of LOWELL.

Hashing is like indexing in that it involves associating a key with a relative record address. Hashing differs from indexing in two important ways:

- With hashing, the addresses generated appear to be random—there is no immediately obvious connection between the key and the location of the corresponding record, even though the key is used to determine the location of the record. For this reason, hashing is sometimes referred to as *randomizing*.
- With hashing, two different keys may be transformed to the same address so two records may be sent to the same place in the file. When this occurs, it is called a *collision* and some means must be found to deal with it.

Consider the following simple example. Suppose you want to store seventy-five records in a file in which the key to each record is a person's name. Suppose also that you set aside space for one thousand records. The key can be hashed by taking two numbers from the ASCII representations of the first two characters of the name, multiplying these together, then

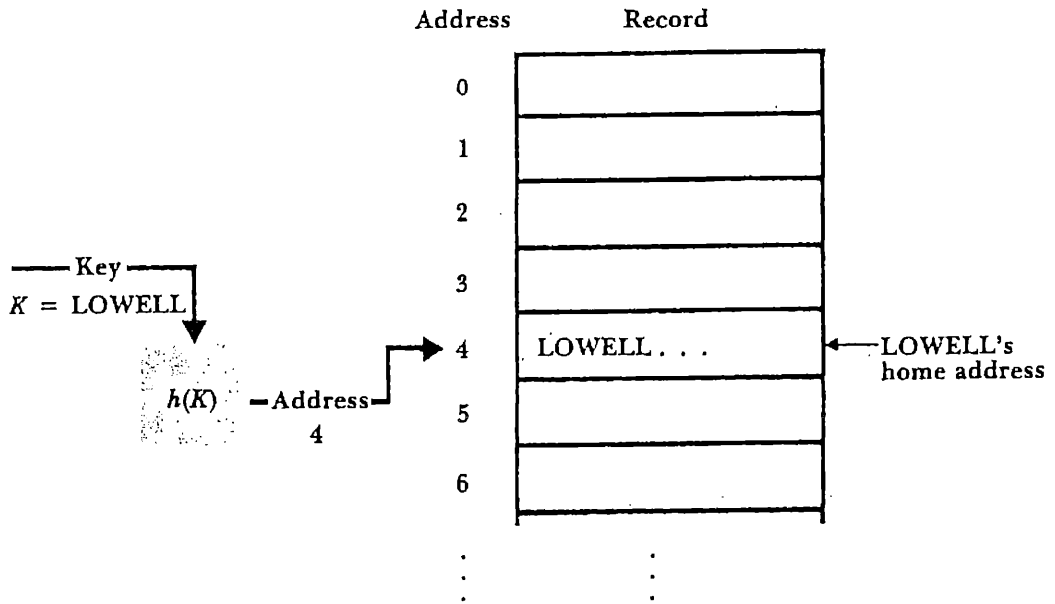


Figure 11.1 Hashing the key LOWELL to address 4.

using the rightmost three digits of the result for the address. Table 11.1 shows how three names would produce three addresses. Note that even though the names are listed in alphabetical order, there is no apparent order to the addresses. They appear to be in *random* order.

11.1.2 Collisions

Now suppose there is a key in the sample file with the name OLIVIER. Because the name OLIVIER starts with the same two letters as the name LOWELL, they produce the same address (004). There is a *collision* between the record for OLIVIER and the record for LOWELL. We refer to keys that hash to the same address as *synonyms*.

Collisions cause problems. We cannot put two records in the same space, so we must resolve collisions. We do this in two ways: by choosing hashing algorithms partly on the basis of how few collisions they are likely to produce and by playing some tricks with the way we store records.

The ideal solution to collisions is to find a transformation algorithm that avoids collisions altogether. Such an algorithm is called a *perfect hashing algorithm*. It turns out to be much more difficult to find a perfect hashing algorithm than one might expect. Suppose, for example, that you want to store 4000 records among 5000 available addresses. It can be shown (Hanson, 1982) that of the huge number of possible hashing algorithms

Table 11.1 A simple hashing scheme

Name	ASCII code for first two letters	Product	Home address
BALL	66 65	$66 \times 65 = 4290$	290
LOWELL	76 79	$76 \times 79 = 6004$	004
TREE	84 82	$84 \times 82 = 6888$	888

for doing this, only one out of $10^{120\,000}$ avoids collisions altogether. Hence, it is usually not worth trying.¹

A more practical solution is to reduce the number of collisions to an acceptable number. For example, if only one out of ten searches for a record results in a collision, then the *average* number of disk accesses required to retrieve a record remains quite low. There are several different ways to reduce the number of collisions, including the following:

- *Spread out the records.* Collisions occur when two or more records compete for the same address. If we could find a hashing algorithm that distributes the records fairly randomly among the available addresses, then we would not have large numbers of records clustering around certain addresses. Our sample hash algorithm, which uses only two letters from the key, is not good on this account because certain combinations of two letters are quite common in starting names, while others are uncommon (e.g., compare the number of names that start with “JO” with the number that start with “XZ”). We need to find a hashing algorithm that distributes records more randomly.
- *Use extra memory.* It is easier to find a hash algorithm that avoids collisions if we have only a few records to distribute among many addresses than if we have about the same number of records as addresses. Our sample hashing algorithm is very good on this account since there are one thousand possible addresses, and only seventy-five addresses (corresponding to the seventy-five records) will be generated. The

1. It is not unreasonable to try to generate perfect hashing functions for small (less than 500), stable sets of keys, such as might be used to look up reserved words in a programming language. But files generally contain more than a few hundred keys, or they contain sets of keys that change frequently, so they are not normally considered candidates for perfect hashing functions. See Knuth (1998), Sager (1985), Chang (1984), and Chichelli (1980) for more on perfect hashing functions.

obvious disadvantage to spreading out the records is that storage space is wasted. (In the example, 7.5 percent of the available record space is used, and the remaining 92.5 percent is wasted.) There is no simple answer to the question of how much empty space should be tolerated to get the best hashing performance, but some techniques are provided later in this chapter for measuring the relative gains in performance for different amounts of free space.

- *Put more than one record at a single address.* Up to now we have assumed tacitly that each physical record location in a file could hold exactly one record, but there is usually no reason we cannot create our file in such a way that every file address is big enough to hold several records. If, for example, each record is 80 bytes long and we create a file with 512-byte physical records, we can store up to six records at each file address. Each address is able to tolerate five synonyms. Addresses that can hold several records in this way are sometimes called *buckets*.

In the following sections we elaborate on these collision-reducing methods, and as we do so we present some programs for managing hashed files.

11.2 A Simple Hashing Algorithm

One goal in choosing any hashing algorithm should be to spread out records as uniformly as possible over the range of addresses available. The use of the term *hash* for this technique suggests what is done to achieve this. Our dictionary reminds us that the verb *to hash* means “to chop into small pieces . . . muddle or confuse.” The algorithm used previously chops off the first two letters and then uses the resulting ASCII codes to produce a number that is in turn chopped to produce the address. It is not very good at avoiding clusters of synonyms because so many names begin with the same two letters.

One problem with the algorithm is that it does not do very much hashing. It uses only two letters of the key and does little with those two letters. Now let us look at a hash function that does much more randomizing, primarily because it uses more of the key. It is a reasonably good basic algorithm and is likely to give good results no matter what kinds of keys are used. It is also an algorithm that is not too difficult to alter in case a specific instance of the algorithm does not work well.

This algorithm has three steps:

1. Represent the key in numerical form.
2. Fold and add.
3. Divide by a prime number and use the remainder as the address.

Step 1. Represent the Key in Numerical Form

If the key is already a number, then this step is already accomplished. If it is a string of characters, we take the ASCII code of each character and use it to form a number. For example,

```

LOWELL =      76 79 87 69 76 76 32 32 32 32 32 32
              L  O  W  E  L  L |←      Blanks      →|

```

In this algorithm we use the entire key rather than just the first two letters. By using more parts of a key, we increase the likelihood that differences among the keys cause differences in addresses produced. The extra processing time required to do this is usually insignificant when compared with the potential improvement in performance.

Step 2. Fold and Add

Folding and adding means chopping off pieces of the number and adding them together. In our algorithm we chop off pieces with two ASCII numbers each:

```
76 79 | 87 69 | 76 76 | 32 32 | 32 32 | 32 32
```

These number pairs can be thought of as integer variables (rather than character variables, which is how they started out) so we can do arithmetic on them. If we can treat them as integer variables, then we can add them. This is easy to do in C because C allows us to do arithmetic on characters. In Pascal, we can use the *ord()* function to obtain the integer position of a character within the computer's character set.

Before we add the numbers, we have to mention a problem caused by the fact that in most cases the sizes of numbers we can add together are limited. On some microcomputers, for example, integer values that exceed 32 767 (15 bits) cause overflow errors or become negative. For example, adding the first five of the foregoing numbers gives

$$7679 + 8769 + 7676 + 3232 + 3232 = 30\ 588$$

Adding in the last 3232 would, unfortunately, push the result over the maximum 32 767 ($30\,588 + 3232 = 33\,820$), causing an overflow error. Consequently, we need to make sure that each successive sum is less than 32 767. We can do this by first identifying the largest single value we will ever add in our summation and then making sure after each step that our intermediate result differs from 32 767 by that amount.

In our case, let us assume that keys consist only of blanks and upper-case alphabetic characters, so the largest addend is 9090, corresponding to ZZ. Suppose we choose 19 937 as our largest allowable intermediate result. This differs from 32 767 by much more than 9090, so we can be confident (in this example) that no new addition will cause overflow. We can ensure in our algorithm that no intermediate sum exceeds 19 937 by using the *mod* operator, which returns the remainder when one integer is divided by another:

7679 + 8769	→ 16 448	→ 16 448 mod 19 937	→ 16448
16 448 + 7676	→ 24 124	→ 24 124 mod 19 937	→ 4187
4187 + 3232	→ 7419	→ mod 19 937	→ 7419
7419 + 3232	→ 10 651	→ mod 19 937	→ 10 651
10 651 + 3232	→ 13 883	→ 13 883 mod 19 937	→ 13 883

The number 13 883 is the result of the fold-and-add operation.

Why did we use 19 937 as our upper bound rather than, say, 20 000? Because the division and subtraction operations associated with the *mod* operator are more than just a way of keeping the number small; they are part of the transformation work of the hash function. As we see in the discussion for the next step, division by a prime number usually produces a more random distribution than does transformation by a nonprime. The number 19 937 is prime.

Step 3. Divide by the Size of the Address Space

The purpose of this step is to cut down to size the number produced in step 2 so it falls within the range of addresses of records in the file. This can be done by dividing that number by a number that is the address size of the file, then taking the remainder. The remainder will be the home address of the record.

We can represent this operation symbolically as follows: if s represents the sum produced in step 2 (13 883 in the example), n represents the divisor (the number of addresses in the file), and a represents the address we are trying to produce, we apply the formula

$$a = s \bmod n$$

The remainder produced by the mod operator will be a number between 0 and $n - 1$.

Suppose, for example, that we decide to use the 100 addresses 0–99 for our file. In terms of the preceding formula,

$$\begin{aligned} a &= 13\,883 \bmod 100 \\ &= 83 \end{aligned}$$

Since the number of addresses allocated for the file does not have to be any specific size (as long as it is big enough to hold all of the records to be stored in the file), we have a great deal of freedom in choosing the divisor n . It is a good thing that we do, because the choice of n can have a major effect on how well the records are spread out.

A *prime* number is usually used for the divisor because primes tend to distribute remainders much more uniformly than do nonprimes. A nonprime can work well in many cases, however, especially if it has no prime divisors less than 20 (Hanson, 1982). Since the remainder is going to be the address of a record, we choose a number as close as possible to the desired size of the address space. This number determines the size of the address space. For a file with 75 records, a good choice might be 101, which would leave the file 74.3 percent full ($74/101 = 0.743$).

If 101 is the size of the address space, the home address of the record in the example becomes

$$\begin{aligned} a &= 13\,883 \bmod 101 \\ &= 46 \end{aligned}$$

Hence, the record whose key is LOWELL is assigned to record number 46 in the file.

This procedure can be carried out with the function *Hash* in Fig. 11.2. Function *Hash* takes two inputs: *key*, which must be an array of ASCII codes for at least twelve characters, and *maxAddress*, which has the maximum address value. The value returned by *Hash* is the address.

```
int Hash (char key[12], int maxAddress)
{
    int sum = 0;
    for (int j = 0; j < 12; j += 2)
        sum = (sum * 100 * key[j] * key[j+1]) % 19937;
    return sum % maxAddress;
}
```

Figure 11.2 Function *Hash* uses folding and prime number division to compute a hash address for a twelve-character string.

11.3 Hashing Functions and Record Distributions

Of the two hash functions we have so far examined, one spreads out records pretty well, and one does not spread them out well at all. In this section we look at ways to describe distributions of records in files. Understanding distributions makes it easier to discuss other hashing methods.

11.3.1 Distributing Records among Addresses

Figure 11.3 illustrates three different distributions of seven records among ten addresses. Ideally, a hash function should distribute records in a file so there are no collisions, as illustrated by distribution (a). Such a distribution is called *uniform* because the records are spread out uniformly among the addresses. We pointed out earlier that completely uniform distributions are so hard to find it is generally not considered worth trying to find them.

Distribution (b) illustrates the worst possible kind of distribution. All records share the same home address, resulting in the maximum number of collisions. The more a distribution looks like this one, the more that collisions will be a problem.

Distribution (c) illustrates a distribution in which the records are somewhat spread out, but with a few collisions. This is the most likely case if we have a function that distributes keys *randomly*. If a hash function is random, then for a given key every address has the same likelihood of being chosen as every other address. The fact that a certain address is chosen for one key neither diminishes nor increases the likelihood that the same address will be chosen for another key.

It should be clear that if a random hash function is used to generate a large number of addresses from a large number of keys, then simply *by chance* some addresses are going to be generated more often than others. If you have, for example, a random hash function that generates addresses between 0 and 99 and you give the function one hundred keys, you would expect some of the one hundred addresses to be chosen more than once and some to be chosen not at all.

Although a random distribution of records among available addresses is not ideal, it is an acceptable alternative given that it is practically impossible to find a function that allows a uniform distribution. Uniform distri-

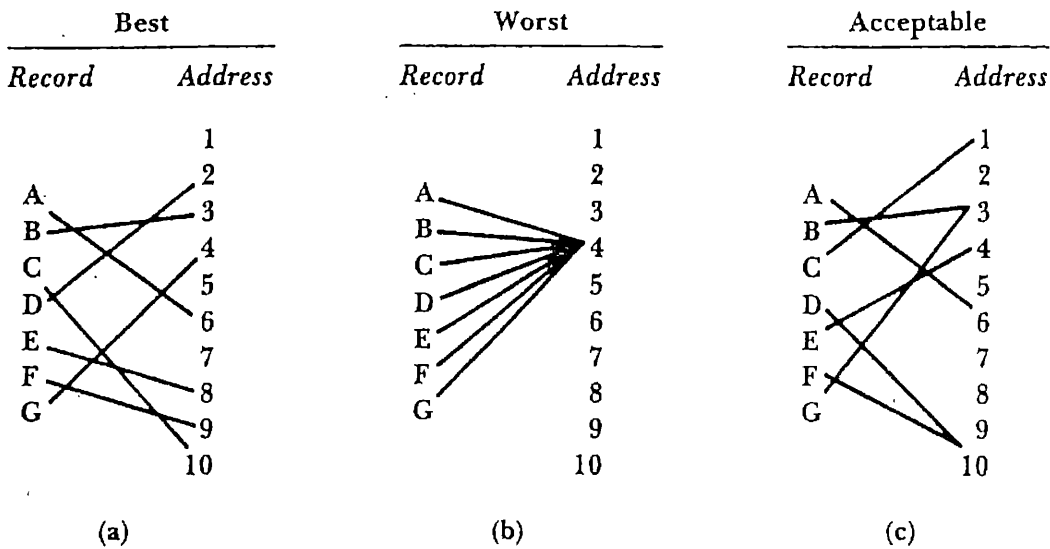


Figure 11.3 Different distributions. (a) No synonyms (uniform). (b) All synonyms (worst case). (c) A few synonyms.

butions may be out of the question, but there are times when we can find distributions that are better than random in the sense that, while they do generate a fair number of synonyms, they spread out records among addresses more uniformly than does a random distribution.

11.3.2 Some Other Hashing Methods

It would be nice if there were a hash function that guaranteed a better-than-random distribution in all cases, but there is not. The distribution generated by a hashing function depends on the set of keys that are actually hashed. Therefore, the choice of a proper hashing function should involve some intelligent consideration of the keys to be hashed, and perhaps some experimentation. The approaches to choosing a reasonable hashing function covered in this section are ones that have been found to work well, given the right circumstances. Further details on these and other methods can be found in Knuth (1998), Maurer (1975), Hanson (1982), and Sorenson et al. (1978).

Here are some methods that are potentially better than random:

- *Examine keys for a pattern.* Sometimes keys fall in patterns that naturally spread themselves out. This is more likely to be true of numeric keys than of alphabetic keys. For example, a set of employee

identification numbers might be ordered according to when the employees entered an organization. This might even lead to *no* synonyms. If some *part* of a key shows a usable underlying pattern, a hash function that extracts that part of the key can also be used.

- *Fold parts of the key.* Folding is one stage in the method discussed earlier. It involves extracting digits from part of a key and adding the extracted parts together. This method destroys the original key patterns but in some circumstances may preserve the separation between certain *subsets* of keys that naturally spread themselves out.
- *Divide the key by a number.* Division by the address size and use of the remainder usually is involved somewhere in a hash function since the purpose of the function is to produce an address within a certain range. Division preserves consecutive key sequences, so you can take advantage of sequences that effectively spread out keys. However, if there are *several* consecutive key sequences, division by a number that has many small factors can result in many collisions. Research has shown that numbers with no divisors less than 19 generally avoid this problem. Division by a *prime* is even more likely than division by a nonprime to generate different results from different consecutive sequences.

The preceding methods are designed to take advantage of natural orderings among the keys. The next two methods should be tried when, for some reason, the better-than-random methods do not work. In these cases, randomization is the goal.

- *Square the key and take the middle.* This popular method (often called the *mid-square* method) involves treating the key as a single large number, squaring the number, and extracting whatever number of digits is needed from the middle of the result. For example, suppose you want to generate addresses between 0 and 99. If the key is the number 453, its square is 205 209. Extracting the middle two digits yields a number between 0 and 99, in this case 52. As long as the keys do not contain many leading or trailing zeros, this method usually produces fairly random results. One unattractive feature of this method is that it often requires multiple precision arithmetic.
- *Radix transformation.* This method involves converting the key to some number base other than the one you are working in, then taking the result modulo the maximum address as the hash address. For

example, suppose you want to generate addresses between 0 and 99. If the key is the decimal number 453, its base 11 equivalent is 382; $382 \bmod 99 = 85$, so 85 is the hash address.

Radix transformation is generally more reliable than the mid-square method for approaching true randomization, though mid-square has been found to give good results when applied to some sets of keys.

11.3.3 Predicting the Distribution of Records

Given that it is nearly impossible to achieve a uniform distribution of records among the available addresses in a file, it is important to be able to predict how records are likely to be distributed. If we know, for example, that a large number of addresses are likely to have far more records assigned to them than they can hold, then we know that there are going to be a lot of collisions.

Although there are no nice mathematical tools available for predicting collisions among distributions that are better than random, there are mathematical tools for understanding just this kind of behavior when records are distributed randomly. If we assume a random distribution (knowing that very likely it will be better than random), we can use these tools to obtain conservative estimates of how our hashing method is likely to behave.

The Poisson Distribution²

We want to predict the number of collisions that are likely to occur in a file that can hold only one record at an address. We begin by concentrating on what happens to a single given address when a hash function is applied to a key. We would like to answer the following questions: When all of the keys in a file are hashed, what is the likelihood that

- None will hash to the given address?
- Exactly one key will hash to the address?
- Exactly two keys will hash to the address (two synonyms)?

2. This section develops a formula for predicting the ways in which records will be distributed among addresses in a file if a random hashing function is used. The discussion assumes knowledge of some elementary concepts of probability and combinatorics. You may want to skip the development and go straight to the formula, which is introduced in the next section.

- Exactly three, four, and so on keys will hash to the address?
- All keys in the file will hash to the same given address?

Which of these outcomes would you expect to be fairly likely, and which quite unlikely? Suppose there are N addresses in a file. When a single key is hashed, there are two possible outcomes with respect to the given address:

A —The address is not chosen; or

B —The address is chosen.

How do we express the probabilities of the two outcomes? If we let both $p(A)$ and a stand for the probability that the address is not chosen, and $p(B)$ and b stand for the probability that the address is chosen, then

$$p(B) = b = \frac{1}{N}$$

since the address has one chance in N of being chosen, and

$$p(A) = a = \frac{N-1}{N} = 1 - \frac{1}{N}$$

since the address has $N-1$ chances in N of not being chosen. If there are 10 addresses ($N = 10$), the probability of our address being chosen is $b = 1/10 = 0.1$, and the probability of the address not being chosen is $a = 1 - 0.1 = 0.9$.

Now suppose *two* keys are hashed. What is the probability that both keys hash to our given address? Since the two applications of the hashing function are independent of one another, the probability that both will produce the given address is a *product*:

$$p(BB) = b \times b = \frac{1}{N} \times \frac{1}{N} \quad \text{for } N = 10 : b \times b = 0.1 \times 0.1 = 0.01$$

Of course, other outcomes are possible when two keys are hashed. For example, the second key could hash to an address other than the given address. The probability of this is the product

$$p(BA) = b \times a = \frac{1}{N} \times \left(1 - \frac{1}{N}\right) \quad \text{for } N = 10 : b \times a = 0.1 \times 0.9 = 0.09$$

In general, when we want to know the probability of a certain sequence of outcomes, such as $BABBA$, we can replace each A and B by a and b , respectively, and compute the indicated product:

$$p(BABBA) = b \times a \times b \times b \times a = a^2 b^3 = (0.9)^2 (0.1)^3.$$

This example shows how to find the probability of three *B*s and two *A*s, where the *B*s and *A*s occur in the order shown. We want to know the probability that there are a certain number of *B*s and *A*s, but *without regard to order*. For example, suppose we are hashing four keys and we want to know how likely it is that exactly two of the keys hash to our given address. This can occur in six ways, all six ways having the same probability:

Outcome	Probability	For $N = 10$
BBAA	$bbaa = b^2 a^2$	$(0.1)^2(0.9)^2 = 0.0036$
BABA	$baba = b^2 a^2$	$(0.1)^2(0.9)^2 = 0.0036$
BAAB	$baab = b^2 a^2$	$(0.1)^2(0.9)^2 = 0.0036$
ABBA	$abba = b^2 a^2$	$(0.1)^2(0.9)^2 = 0.0036$
ABAB	$abab = b^2 a^2$	$(0.1)^2(0.9)^2 = 0.0036$
AABB	$aabb = b^2 a^2$	$(0.1)^2(0.9)^2 = 0.0036$

Since these six sequences are independent of one another, the probability of two *B*s and two *A*s is the sum of the probabilities of the individual outcomes:

$$p(BBAA) + p(BABA) + \dots + p(AABB) = 6b^2a^2 = 6 \times 0.0036 = 0.0216.$$

The 6 in the expression $6b^2a^2$ represents the number of ways two *B*s and two *A*s can be distributed among four places.

In general, the event “ r trials result in $r - x$ *A*s and x *B*s” can happen in as many ways as $r - x$ letters *A* can be distributed among r places. The probability of each such way is

$$a^{r-x} b^x$$

and the number of such ways is given by the formula

$$C = \frac{r!}{(r-x)! x!}$$

This is the well-known formula for the number of ways of selecting x items out of a set of r items. It follows that when r keys are hashed, the probability that an address will be chosen x times and not chosen $r - x$ times can be expressed as

$$p(x) = C a^{r-x} b^x$$

Furthermore, if we know that there are N addresses available, we can be precise about the individual probabilities of A and B , and the formula becomes

$$p(x) = C \left(1 - \frac{1}{N}\right)^{r-x} \left(\frac{1}{N}\right)^x$$

where C has the definition given previously.

What does this *mean*? It means that if, for example, $x = 0$, we can compute the probability that a given address will have 0 records assigned to it by the hashing function using the formula

$$p(0) = C \left(1 - \frac{1}{N}\right)^{r-0} \left(\frac{1}{N}\right)^0$$

If $x = 1$, this formula gives the probability that *one* record will be assigned to a given address:

$$p(1) = C \left(1 - \frac{1}{N}\right)^{r-1} \left(\frac{1}{N}\right)^1$$

This expression has the disadvantage that it is awkward to compute. (Try it for 1000 addresses and 1000 records: $N = r = 1000$.) Fortunately, for large values of N and r , there is a function that is a very good approximation for $p(x)$ and is much easier to compute. It is called the *Poisson function*.

The Poisson Function Applied to Hashing

The Poisson function, which we also denote by $p(x)$, is given by

$$p(x) = \frac{(r/N)^x e^{-(r/N)}}{x!}$$

where N , r , x , and $p(x)$ have exactly the same meaning they have in the previous section. That is, if

- N = the number of available addresses;
- r = the number of records to be stored; and
- x = the number of records assigned to a given address,

then $p(x)$ gives the probability that a given address will have had x records assigned to it after the hashing function has been applied to all n records.

Suppose, for example, that there are 1000 addresses ($N = 1000$) and 1000 records whose keys are to be hashed to the addresses ($r = 1000$). Since

$r/N = 1$, the probability that a given address will have *no* keys hashed to it ($x = 0$) becomes

$$p(0) = \frac{1^0 e^{-1}}{0!} = 0.368$$

The probabilities that a given address will have exactly one, two, or three keys, respectively, hashed to it are

$$p(1) = \frac{1^1 e^{-1}}{1!} = 0.368$$

$$p(2) = \frac{1^2 e^{-1}}{2!} = 0.184$$

$$p(3) = \frac{1^3 e^{-1}}{3!} = 0.061$$

If we can use the Poisson function to estimate the probability that a given address will have a certain number of records, we can also use it to predict the number of addresses that will have a certain number of records assigned.

For example, suppose there are 1000 addresses ($N = 1000$) and 1000 records ($r = 1000$). Multiplying 1000 by the probability that a *given* address will have x records assigned to it gives the expected *total* number of addresses with x records assigned to them. That is, $1000p(x)$ gives the number of addresses with x records assigned to them.

In general, if there are N addresses, then the expected number of addresses with x records assigned to them is

$$Np(x)$$

This suggests another way of thinking about $p(x)$. Rather than thinking about $p(x)$ as a measure of probability, we can think of $p(x)$ as giving the proportion of addresses having x logical records assigned by hashing.

Now that we have a tool for predicting the expected proportion of addresses that will have zero, one, two, etc. records assigned to them by a random hashing function, we can apply this tool to predicting numbers of collisions.

11.3.4 Predicting Collisions for a Full File

Suppose you have a hashing function that you believe will distribute records randomly and you want to store 10 000 records in 10 000 addresses. How many addresses do you expect to have no records assigned to them?

Since $r = 10\,000$ and $N = 10\,000$, $r/N = 1$. Hence the proportion of addresses with 0 records assigned should be

$$p(0) = \frac{1^0 e^{-1}}{0!} \doteq 0.3679$$

The *number* of addresses with no records assigned is

$$10\,000 \times p(0) = 3679$$

How many addresses should have one, two, and three records assigned, respectively?

$$10\,000 \times p(1) = 0.3679 \times 10\,000 = 3679$$

$$10\,000 \times p(2) = 0.1839 \times 10\,000 = 1839$$

$$10\,000 \times p(3) = 0.0613 \times 10\,000 = 613$$

Since the 3679 addresses corresponding to $x = 1$ have exactly one record assigned to them, their records have no synonyms. The 1839 addresses with two records apiece, however, represent potential trouble. If each such address has space only for one record, and two records are assigned to them, there is a collision. This means that 1839 records will fit into the addresses, but another 1839 will not fit. There will be 1839 *overflow* records.

Each of the 613 addresses with three records apiece has an even bigger problem. If each address has space for only one record, there will be two overflow records per address. Corresponding to these addresses will be a total of $2 \times 613 = 1226$ overflow records. This is a bad situation. We have thousands of records that do not fit into the addresses assigned by the hashing function. We need to develop a method for handling these overflow records. But first, let's try to reduce the *number* of overflow records.

11.4 How Much Extra Memory Should Be Used?

We have seen the importance of choosing a good hashing algorithm to reduce collisions. A second way to decrease the number of collisions (and thereby decrease the average search length) is to use extra memory. The tools developed in the previous section can be used to help us determine the effect of the use of extra memory on performance.

11.4.1 Packing Density

The term *packing density* refers to the ratio of the number of records to be stored (r) to the number of available spaces (N):³

$$\frac{\text{Number of records}}{\text{Number of spaces}} = \frac{r}{N} = \text{packing density}$$

For example, if there are 75 records ($n = 75$) and 100 addresses ($N = 100$), the packing density is

$$\frac{75}{100} = 0.75 = 75\%$$

The packing density gives a measure of the amount of space in a file that is used, and it is the only such value needed to assess performance in a hashing environment, assuming that the hash method used gives a reasonably random distribution of records. The raw size of a file and its address space do not matter; what is important is the relative sizes of the two, which are given by the packing density.

Think of packing density in terms of tin cans lined up on a 10-foot length of fence. If there are ten tin cans and you throw a rock, there is a certain likelihood that you will hit a can. If there are twenty cans on the same length of fence, the fence has a higher packing density and your rock is more likely to hit a can. So it is with records in a file. The more records there are packed into a given file space, the more likely it is that a collision will occur when a new record is added.

We need to decide how much space we are willing to waste to reduce the number of collisions. The answer depends in large measure on particular circumstances. We want to have as few collisions as possible, but not, for example, at the expense of requiring the file to use two disks instead of one.

11.4.2 Predicting Collisions for Different Packing Densities

We need a quantitative description of the effects of changing the packing density. In particular, we need to be able to predict the number of collisions that are likely to occur for a given packing density. Fortunately, the Poisson function provides us with just the tool to do this.

3. We assume here that only one record can be stored at each address. In fact, that is not necessarily the case, as we see later.

You may have noted already that the formula for packing density (r/N) occurs twice in the Poisson formula

$$p(x) = \frac{(r/N)^x e^{-(r/N)}}{x!}$$

Indeed, the numbers of records (r) and addresses (N) always occur together as the *ratio* r/N . They never occur independently. An obvious implication of this is that the way records are distributed depends partly on the ratio of the number of records to the number of available addresses, and *not* on the absolute numbers of records or addresses. The same behavior is exhibited by 500 records distributed among 1000 addresses as by 500 000 records distributed among 1 000 000 addresses.

Suppose that 1000 addresses are allocated to hold 500 records in a randomly hashed file, and that each address can hold one record. The packing density for the file is

$$\frac{r}{N} = \frac{500}{1000} = 0.5$$

Let us answer the following questions about the distribution of records among the available addresses in the file:

- How many addresses should have no records assigned to them?
- How many addresses should have exactly one record assigned (no synonyms)?
- How many addresses should have one record *plus* one or more synonyms?
- Assuming that only one record can be assigned to each home address, how many overflow records can be expected?
- What percentage of records should be overflow records?

1. How many addresses should have no records assigned to them? Since $p(0)$ gives the *proportion* of addresses with no records assigned, the number of such addresses is

$$\begin{aligned} Np(0) &= 1000 \times \frac{(0.5)^0 e^{-0.5}}{0!} \\ &= 1000 \times 0.607 \\ &= 607 \end{aligned}$$

2. How many addresses should have exactly one record assigned (no synonyms)?

$$\begin{aligned}
 Np(1) &= 1000 \times \frac{(0.5)^1 e^{-0.5}}{1!} \\
 &= 1000 \times 0.303 \\
 &= 303
 \end{aligned}$$

3. *How many addresses should have one record plus one or more synonyms?* The values of $p(2)$, $p(3)$, $p(4)$, and so on give the proportions of addresses with one, two, three, and so on synonyms assigned to them. Hence the sum

$$p(2) + p(3) + p(4) + \dots$$

gives the proportion of all addresses with at least one synonym. This may appear to require a great deal of computation, but it doesn't since the values of $p(x)$ grow quite small for x larger than 3. This should make intuitive sense. Since the file is only 50 percent loaded, one would not expect very many keys to hash to any one address. Therefore, the number of addresses with more than about three keys hashed to them should be quite small. We need only compute the results up to $p(5)$ before they become insignificantly small:

$$\begin{aligned}
 p(2) + p(3) + p(4) + p(5) &= 0.0758 + 0.0126 + 0.0016 + 0.0002 \\
 &= 0.0902
 \end{aligned}$$

The *number* of addresses with one or more synonyms is just the product of N and this result:

$$\begin{aligned}
 N[p(2) + p(3) + \dots] &= 1000 \times 0.0902 \\
 &= 90
 \end{aligned}$$

4. *Assuming that only one record can be assigned to each home address, how many overflow records could be expected?* For each of the addresses represented by $p(2)$, one record can be stored at the address and one must be an overflow record. For each address represented by $p(3)$, one record can be stored at the address, *two* are overflow records, and so on. Hence, the expected number of overflow records is given by

$$\begin{aligned}
 1 \times N \times p(2) + 2 \times N \times p(3) + 3 \times N \times p(4) + 4 \times N \times p(5) \\
 &= N \times [1 \times p(2) + 2 \times p(3) + 3 \times p(4) + 4 \times p(5)] \\
 &= 1000 \times [1 \times 0.0758 + 2 \times 0.0126 + 3 \times 0.0016 + 4 \times 0.0002] \\
 &= 107
 \end{aligned}$$

5. *What percentage of records should be overflow records?* If there are 107 overflow records and 500 records in all, then the proportion of overflow records is

$$\frac{107}{500} = 0.124 = 21.4\%$$

Conclusion: if the packing density is 50 percent and each address can hold only one record, we can expect about 21 percent of all records to be stored somewhere other than at their home addresses.

Table 11.2 shows the proportion of records that are not stored in their home addresses for several different packing densities. The table shows that if the packing density is 10 percent, then about 5 percent of the time we try to access a record, there is already another record there. If the density is 100 percent, then about 37 percent of all records collide with other records at their home addresses. The 4.8 percent collision rate that results when the packing density is 10 percent looks very good until you realize that for every record in your file there will be nine unused spaces!

The 36.8 percent that results from 100 percent usage looks good when viewed in terms of 0 percent unused space. Unfortunately, 36.8 percent doesn't tell the whole story. If 36.8 percent of the records are not at their

Table 11.2 Effect of packing density on the proportion of records not stored at their home addresses

Packing density (percent)	Synonyms as percent of records
10	4.8
20	9.4
30	13.6
40	17.6
50	21.4
60	24.8
70	28.1
80	31.2
90	34.1
100	36.8

home addresses, then they are somewhere else, probably in many cases using addresses that are home addresses for other records. The more homeless records there are, the more contention there is for space with other homeless records. After a while, clusters of overflow records can form, leading in some cases to extremely long searches for some of the records. Clearly, the placement of records that collide is an important matter. Let us now look at one simple approach to placing overflow records.

11.5 Collision Resolution by Progressive Overflow

Even if a hashing algorithm is very good, it is likely that collisions will occur. Therefore, any hashing program must incorporate some method for dealing with records that cannot fit into their home addresses. There are a number of techniques for handling overflow records, and the search for ever better techniques continues to be a lively area of research. We examine several approaches, but we concentrate on a very simple one that often works well. The technique has various names, including *progressive overflow* and *linear probing*.

11.5.1 How Progressive Overflow Works

An example of a situation in which a collision occurs is shown in Fig. 11.4. In the example, we want to store the record whose key is York in the file. Unfortunately, the name York hashes to the same address as the name Rosen, whose record is already stored there. Since York cannot fit in its home address, it is an overflow record. If progressive overflow is used, the next several addresses are searched in sequence until an empty one is found. The first free address becomes the address of the record. In the example, address 9 is the first record found empty, so the record pertaining to York is stored in address 9.

Eventually we need to find York's record in the file. Since York still hashes to 6, the search for the record begins at address 6. It does not find York's record there, so it proceeds to look at successive records until it gets to address 9, where it finds York.

An interesting problem occurs when there is a search for an open space or for a record at the *end* of the file. This is illustrated in Fig. 11.5, in which it is assumed that the file can hold 100 records in addresses 0–99. Blue is hashed to record number 99, which is already occupied by Jello.

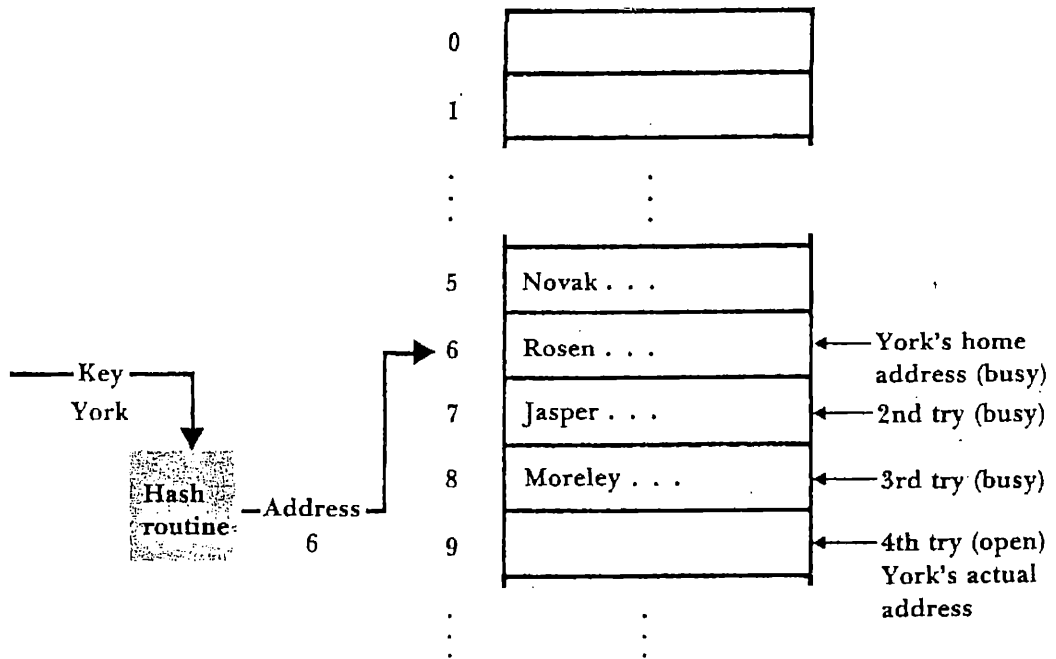


Figure 11.4 Collision resolution with progressive overflow.

Since the file holds only 100 records, it is not possible to use 100 as the next address. The way this is handled in progressive overflow is to wrap around the address space of the file by choosing address 0 as the next address. Since address 0 is not occupied in this case, Blue gets stored in address 0.

What happens if there is a search for a record but the record was never placed in the file? The search begins, as before, at the record's home address, then proceeds to look for it in successive locations. Two things can happen:

- If an open address is encountered, the searching routine might assume this means that the record is not in the file; or
- If the file is full, the search comes back to where it began. Only then is it clear that the record is not in the file. When this occurs, or even when we approach filling our file, searching can become intolerably slow, whether or not the record being sought is in the file.

The greatest strength of progressive overflow is its simplicity. In many cases, it is a perfectly adequate method. There are, however, collision-handling techniques that perform better than progressive overflow, and we examine some of them later in this chapter. Now let us look at the effect of progressive overflow on performance.

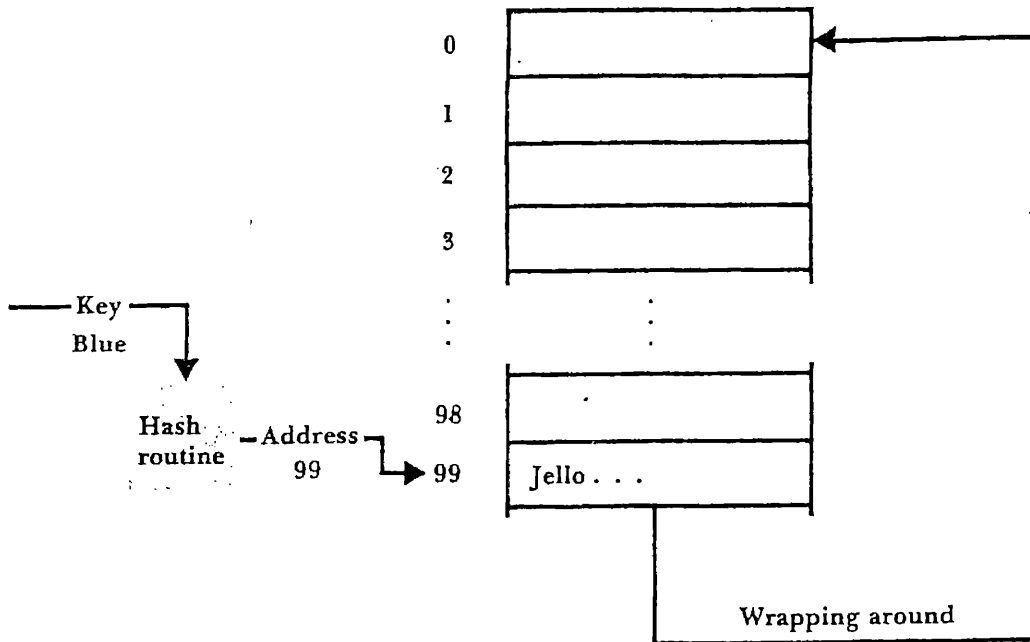


Figure 11.5 Searching for an address beyond the end of a file.

11.5.2 Search Length

The reason to avoid overflow is, of course, that extra searches (hence, extra disk accesses) have to occur when a record is not found in its home address. If there are a lot of collisions, there are going to be a lot of overflow records taking up spaces where they ought not to be. Clusters of records can form, resulting in the placement of records a long way from home, so many disk accesses are required to retrieve them.

Consider the following set of keys and the corresponding addresses produced by some hash function.

Key	Home Address
Adams	20
Bates	21
Cole	21
Dean	22
Evans	20

If these records are loaded into an empty file and progressive overflow is used to resolve collisions, only two of the records will be at their home addresses. All the others require extra accesses to retrieve. Figure 11.6

Actual address		Home address	Number of accesses needed to retrieve
0			
⋮	⋮		
20	Adams . . .	20	1
21	Bates . . .	21	1
22	Cole . . .	21	2
23	Dean . . .	22	2
24	Evans . . .	20	5
25			
⋮	⋮		

Figure 11.6 Illustration of the effects of clustering of records. As keys are clustered, the number of accesses required to access later keys can become large.

shows where each key is stored, together with information on how many accesses are required to retrieve it.

The term *search length* refers to the number of accesses required to retrieve a record from secondary memory. In the context of hashing, the search length for a record increases every time there is a collision. If a record is a long way from its home address, the search length may be unacceptable. A good measure of the extent of the overflow problem is *average search length*. The average search length is the average number of times you can expect to have to access the disk to retrieve a record. A rough estimate of average search length may be computed by finding the *total search length* (the sum of the search lengths of the individual records) and dividing this by the number of records:

$$\text{Average search length} = \frac{\text{total search length}}{\text{total number of records}}$$

In the example, the average search length for the five records is

$$\frac{1 + 1 + 2 + 2 + 5}{5} = 2.2$$

With no collisions at all, the average search length is 1, since only one access is needed to retrieve any record. (We indicated earlier that an algorithm that distributes records so evenly no collisions occur is appropriately called a *perfect* hashing algorithm, and we mentioned that, unfortunately, such an algorithm is almost impossible to construct.) On the other hand, if a large number of the records in a file results in collisions, the average search length becomes quite long. There are ways to estimate the expected average search length, given various file specifications, and we discuss them in a later section.

It turns out that, using progressive overflow, the average search length increases very rapidly as the packing density increases. The curve in Fig. 11.7, adapted from Peterson (1957), illustrates the problem. If the packing density is kept as low as 60 percent, the average record takes fewer than two tries to access, but for a much more desirable packing density of 80 percent or more, it increases very rapidly.

Average search lengths of greater than 2.0 are generally considered unacceptable, so it appears that it is usually necessary to use less than 40 percent of your storage space to get tolerable performance. Fortunately, we can improve on this situation substantially by making one small change to

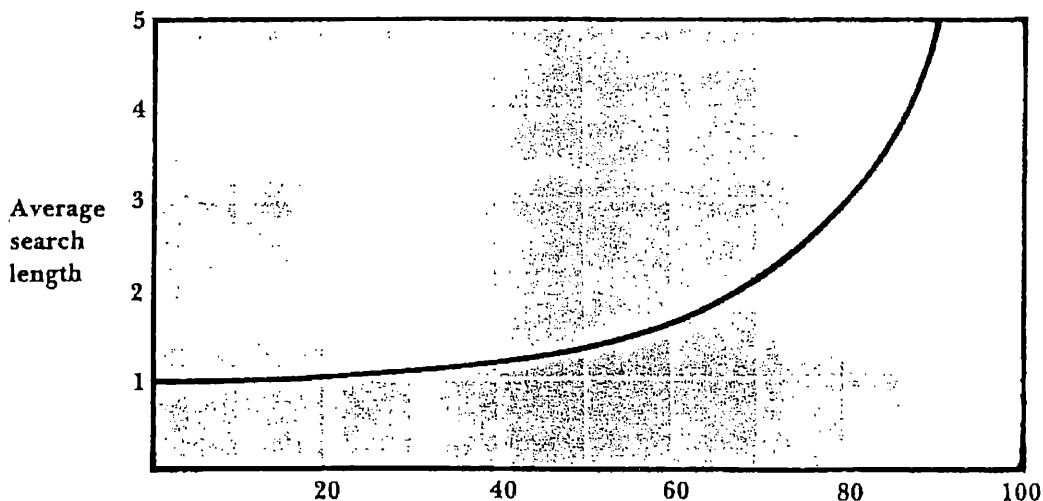


Figure 11.7 Average search length versus packing density in a hashed file in which one record can be stored per address, progressive overflow is used to resolve collisions, and the file has just been loaded.

our hashing program. The change involves putting more than one record at a single address.

11.6 Storing More Than One Record per Address: Buckets

Recall that when a computer receives information from a disk, it is just about as easy for the I/O system to transfer several records as it is to transfer a single record. Recall too that sometimes it might be advantageous to think of records as being grouped together in *blocks* rather than stored individually. Therefore, why not extend the idea of a record address in a file to an address of a *group* of records? The word *bucket* is sometimes used to describe a block of records that is retrieved in one disk access, especially when those records are seen as sharing the same address. On sector-addressing disks, a bucket typically consists of one or more sectors; on block-addressing disks, a bucket might be a block.

Consider the following set of keys, which is to be loaded into a hash file.

Key	Home Address
Green	30
Hall	30
Jenks	32
King	33
Land	33
Marx	33
Nutt	33

Figure 11.8 illustrates part of a file into which the records with these keys are loaded. Each address in the file identifies a bucket capable of holding the records corresponding to three synonyms. Only the record corresponding to Nutt cannot be accommodated in a home address.

When a record is to be stored or retrieved, its home *bucket address* is determined by hashing. The entire bucket is loaded into primary memory. An in-memory search through successive records in the bucket can then be used to find the desired record. When a bucket is filled, we still have to

Bucket contents

⋮

Green . . .	Hall . . .	
Jenks . . .		
King . . .	Land . . .	Marks . . .

⋮

(Nutt . . . is an overflow record)

Figure 11.8 An illustration of buckets. Each bucket can hold up to three records. Only one synonym (Nutt) results in overflow.

worry about the record overflow problem (as in the case of Nutt), but this occurs much less often when buckets are used than when each address can hold only one record.

11.6.1 Effects of Buckets on Performance

When buckets are used, the formula used to compute packing density is changed slightly since each bucket address can hold more than one record. To compute how densely packed a file is, we need to consider both the number of addresses (buckets) and the number of records we can put at each address (bucket size). If N is the number of addresses and b is the number of records that fit in a bucket, then bN is the number of available locations for records. If r is still the number of records in the file, then

$$\text{Packing density} = \frac{r}{bN}$$

Suppose we have a file in which 750 records are to be stored. Consider the following two ways we might organize the file.

- We can store the 750 data records among 1000 locations, where each location can hold one record. The packing density in this case is

$$\frac{750}{1000} = 75\%$$

- We can store the 750 records among 500 locations, where each location has a bucket size of 2. There are still 1000 places (2×500) to store the 750 records, so the packing density is still

$$\frac{r}{bN} = 0.75 = 75\%$$

Since the packing density is not changed, we might at first not expect the use of buckets in this way to improve performance, but in fact it does improve performance dramatically. The key to the improvement is that, although there are fewer addresses, each individual address has more room for variation in the number of records assigned to it.

Let's calculate the difference in performance for these two ways of storing the same number of records in the same amount of space. The starting point for our calculations is the fundamental description of each file structure.

	File without buckets	File with buckets
Number of records	$r = 750$	$r = 750$
Number of addresses	$N = 1000$	$N = 500$
Bucket size	$b = 1$	$b = 2$
Packing density	0.75	0.75
Ratio of records to addresses	$r/N = 0.75$	$r/N = 1.5$

To determine the number of overflow records that are expected in the case of each file, recall that when a random hashing function is used, the Poisson function

$$p(x) = \frac{(r/N)^x e^{-(r/N)}}{x!}$$

gives the expected proportion of addresses assigned x records. Evaluating the function for the two different file organizations, we find that records are assigned to addresses according to the distributions that are shown in Table 11.3.

We see from the table that when buckets are not used, 47.2 percent of the addresses have no records assigned, whereas when two-record buckets are used, only 22.3 percent of the addresses have no records assigned. This should make intuitive sense—since in the two-record case there are only half as many addresses to choose from, it stands to reason that a greater proportion of the addresses are chosen to contain at least one record.

Table 11.3 Poisson distributions for two different file organizations.

$p(x)$	File without buckets ($r/N = 0.75$)	File with buckets ($r/N = 1.5$)
$p(0)$	0.472	0.223
$p(1)$	0.354	0.335
$p(2)$	0.133	0.251
$p(3)$	0.033	0.126
$p(4)$	0.006	0.047
$p(5)$	0.001	0.014
$p(6)$	—	0.004
$p(7)$	—	0.001

Note that the bucket column in Table 11.3 is longer than the nonbucket column. Does this mean that there are more synonyms in the bucket case than in the nonbucket case? Indeed it does, but half of those synonyms do not result in overflow records because each bucket can hold two records. Let us examine this further by computing the exact number of overflow records likely to occur in the two cases.

In the case of the file with bucket size 1, any address that is assigned exactly one record does not have overflow. Any address with more than one record does have overflow. Recall that the expected number of overflow records is given by

$$N \times [1 \times p(2) + 2 \times p(3) + 3 \times p(4) + 4 \times p(5) + \dots]$$

which, for $r/N = 0.75$ and $N = 1000$, is approximately

$$1000 \times [1 \times 0.1328 + 2 \times 0.0332 + 3 \times 0.0062 + 4 \times 0.0009 + 5 \times 0.0001] \\ = 222$$

The 222 overflow records represent 29.6 percent overflow.

In the case of the bucket file, any address that is assigned either one or two records does not have overflow. The value of $p(1)$ (with $r/N = 1.5$) gives the proportion of addresses assigned exactly one record, and $p(2)$ (with $r/N = 1.5$) gives the proportion of addresses assigned exactly two records. It is not until we get to $p(3)$ that we encounter addresses for which

there are overflow records. For each address represented by $p(3)$, two records can be stored at the address, and one must be an overflow record. Similarly, for each address represented by $p(4)$, there are two overflow records, and so forth. Hence, the expected number of overflow records in the bucket file is

$$N \times [1 \times p(3) + 2 \times p(4) + 3 \times p(5) + 4 \times p(6) + \dots]$$

which for $r/N = 1.5$ and $N = 500$ is approximately

$$500 \times [1 \times 0.1255 + 2 \times 0.0471 + 3 \times 0.0141 + 4 \times 0.0035 + 5 \times 0.0008] \\ = 140$$

The 140 overflow records represent 18.7 percent overflow.

We have shown that with one record per address and a packing density of 75 percent, the expected number of overflow records is 29.6 percent. When 500 buckets are used, each capable of holding two records, the packing density remains 75 percent, but the expected number of overflow records drops to 18.7 percent. That is about a 37 percent decrease in the number of times the program has to look elsewhere for a record. As the bucket size gets larger, performance continues to improve.

Table 11.4 shows the proportions of collisions that occur for different packing densities and for different bucket sizes. We see from the table, for example, that if we keep the packing density at 75 percent and increase the bucket size to 10, record accesses result in overflow only 4 percent of the time.

It should be clear that the use of buckets can improve hashing performance substantially. One might ask, "How big should buckets be?" Unfortunately, there is no simple answer to this question because it depends a great deal on a number of different characteristics of the system, including the sizes of buffers the operating system can manage, sector and track capacities on disks, and access times of the hardware (seek, rotation, and data transfer times).

As a rule, it is probably not a good idea to use buckets larger than a track (unless records are very large). Even a track, however, can sometimes be too large when one considers the amount of time it takes to transmit an entire track, as compared with the amount of time it takes to transmit a few sectors. Since hashing almost always involves retrieving only one record per search, any extra transmission time resulting from the use of extra-large buckets is essentially wasted.

In many cases a single cluster is the best bucket size. For example, suppose that a file with 200-byte records is to be stored on a disk system that uses 1024-byte clusters. One could consider each cluster as a bucket,

Table 11.4 Synonyms causing collisions as a percent of records for different packing densities and different bucket sizes

Packing density (%)	Bucket size					
	1	2	5	10	100	
10	4.8	0.6	0.0	0.0	0.0	
20	9.4	2.2	0.1	0.0	0.0	
30	13.6	4.5	0.4	0.0	0.0	
40	17.6	7.3	1.1	0.1	0.0	
50	21.3	11.4	2.5	0.4	0.0	
60	24.8	13.7	4.5	1.3	0.0	
70	28.1	17.0	7.1	2.9	0.0	
75	29.6	18.7	8.6	4.0	0.0	
80	31.2	20.4	11.3	5.2	0.1	
90	34.1	23.8	13.8	8.6	0.8	
100	36.8	27.1	17.6	12.5	4.0	

store five records per cluster, and let the remaining 24 bytes go unused. Since it is no more expensive, in terms of seek time, to access a five-record cluster than it is to access a single record, the only losses from the use of buckets are the extra transmission time and the 24 unused bytes.

The obvious question now is, “How do improvements in the number of collisions affect the average search time?” The answer depends in large measure on characteristics of the drive on which the file is loaded. If there are a large number of tracks in each cylinder, there will be very little seek time because overflow records will be unlikely to spill over from one cylinder to another. If, on the other hand, there is only one track per cylinder, seek time could be a major consumer of search time.

A less exact measure of the amount of time required to retrieve a record is average search length, which we introduced earlier. In the case of buckets, average search length represents the average number of buckets that must be accessed to retrieve a record. Table 11.5 shows the expected average search lengths for files with different packing densities and bucket sizes, given that progressive overflow is used to handle collisions. Clearly, the use of buckets seems to help a great deal in decreasing the average search length. The bigger the bucket, the shorter the search length.

Table 11.5 Average number of accesses required in a successful search by progressive overflow.

Packing density (%)	Bucket size				
	1	2	5	10	100
10	1.06	1.01	1.00	1.00	1.00
30	1.21	1.06	1.00	1.00	1.00
40	1.33	1.10	1.01	1.00	1.00
50	1.50	1.18	1.03	1.00	1.00
60	1.75	1.29	1.07	1.01	1.00
70	2.17	1.49	1.14	1.04	1.00
80	3.00	1.90	1.29	1.11	1.01
90	5.50	3.15	1.78	1.35	1.04
95	10.50	5.60	2.70	1.80	1.10

Adapted from Donald Knuth, *The Art of Computer Programming*, Vol. 3, ©1973, Addison-Wesley, Reading, Mass. Page 536. Reprinted with permission.

11.6.2 Implementation Issues

In the early chapters of this text, we paid quite a bit of attention to issues involved in producing, using, and maintaining random-access files with fixed-length records that are accessed by relative record number (RRN). Since a hashed file is a fixed-length record file whose records are accessed by RRN, you should already know much about implementing hashed files. Hashed files differ from the files we discussed earlier in two important respects, however:

1. Since a hash function depends on there being a fixed number of available addresses, the logical size of a hashed file must be fixed before the file can be populated with records, and it must remain fixed as long as the same hash function is used. (We use the phrase *logical size* to leave open the possibility that physical space be allocated as needed.)
2. Since the home RRN of a record in a hashed file is uniquely related to its key, any procedures that add, delete, or change a record must do so without breaking the bond between a record and its home address. If this bond is broken, the record is no longer accessible by hashing.

We must keep these special needs in mind when we write programs to work with hashed files.

Bucket Structure

The only difference between a file with buckets and one in which each address can hold only one key is that with a bucket file each address has enough space to hold more than one logical record. All records that are housed in the same bucket share the same address. Suppose, for example, that we want to store as many as *five* names in one bucket. Here are three such buckets with different numbers of records.

An empty bucket:	0	/////	/////	/////	/////	/////
Two entries:	2	JONES	ARNSWORTH	/////	/////	/////
A full bucket:	5	JONES	ARNSWORTH	STOCKTON	BRICE	THROOP

Each bucket contains a *counter* that keeps track of how many records it has stored in it. Collisions can occur only when the addition of a new record causes the counter to exceed the number of records a bucket can hold.

The counter tells us how many data records are stored in a bucket, but it does not tell us which slots are used and which are not. We need a way to tell whether a record slot is empty. One simple way to do this is to use a special marker to indicate an empty record, just as we did with deleted records earlier. We use the key value */////* to mark empty records in the preceding illustration.

Initializing a File for Hashing

Since the *logical* size of a hashed file must remain fixed, it makes sense in most cases to allocate physical space for the file before we begin storing data records in it. This is generally done by creating a file of empty spaces for all records, then filling the slots as they are needed with the data records. (It is not necessary to construct a file of empty records before putting data in it, but doing so increases the likelihood that records will be stored close to one another on the disk, avoids the error that occurs when an attempt is made to read a missing record, and makes it easy to process the file sequentially, without having to treat the empty records in any special way.)

Loading a Hash File

A program that loads a hash file is similar in many ways to earlier programs we used for populating fixed-length record files, with two differences. First, the program uses the function `hash` to produce a home address for each key. Second, the program looks for a free space for the record by starting with the bucket stored at its home address and then, if the home bucket is full, continuing to look at successive buckets until one is found that is not full. The new record is inserted in this bucket, and the bucket is rewritten to the file at the location from which it was loaded.

If, as it searches for an empty bucket, a loading program passes the maximum allowable address, it must wrap around to the beginning address. A potential problem occurs in loading a hash file when so many records have been loaded into the file that there are no empty spaces left. A naive search for an open slot can easily result in an infinite loop. Obviously, we want to prevent this from occurring by having the program make sure that there is space available for each new record somewhere in the file.

Another problem that often arises when adding records to files occurs when an attempt is made to add a record that is already stored in the file. If there is a danger of duplicate keys occurring, and duplicate keys are not allowed in the file, some mechanism must be found for dealing with this problem.

11.7 Making Deletions

Deleting a record from a hashed file is more complicated than adding a record for two reasons:

- The slot freed by the deletion must not be allowed to hinder later searches; and
- It should be possible to reuse the freed slot for later additions.

When progressive overflow is used, a search for a record terminates if an open address is encountered. Because of this, we do not want to leave open addresses that break overflow searches improperly. The following example illustrates the problem.

Adams, Jones, Morris, and Smith are stored in a hash file in which each address can hold one record. Adams and Smith both are hashed to address 5, and Jones and Morris are hashed to address 6. If they are loaded

Record	Home address	Actual address		
			:	:
			:	:
			:	:
Adams	5	5	4	
Jones	6	6	5	Adams . . .
Morris	6	7	6	Jones . . .
Smith	5	8	7	Morris . . .
			8	Smith . . .
			:	:
			:	:

Figure 11.9 File organization before deletions.

in alphabetical order using progressive overflow for collisions, they are stored in the locations shown in Fig. 11.9.

A search for Smith starts at address 5 (Smith’s home address), successively looks for Smith at addresses 6, 7, and 8, then finds Smith at 8. Now suppose Morris is deleted, leaving an empty space, as illustrated in Fig. 11.10. A search for Smith again starts at address 5, then looks at addresses 6 and 7. Since address 7 is now empty, it is reasonable for the program to conclude that Smith’s record is not in the file.

11.7.1 Tombstones for Handling Deletions

In Chapter 6 we discussed techniques for dealing with the deletion problem. One simple technique we use for identifying deleted records involves replacing the deleted record (or just its key) with a marker indicating that a record once lived there but no longer does. Such a marker is sometimes referred to as a *tombstone* (Wiederhold, 1983). The nice thing about the use of tombstones is that it solves both of the problems described previously:

- The freed space does not break a sequence of searches for a record; and
- The freed space is obviously available and may be reclaimed for later additions.

Figure 11.11 illustrates how the sample file might look after the tombstone ##### is inserted for the deleted record. Now a search for Smith

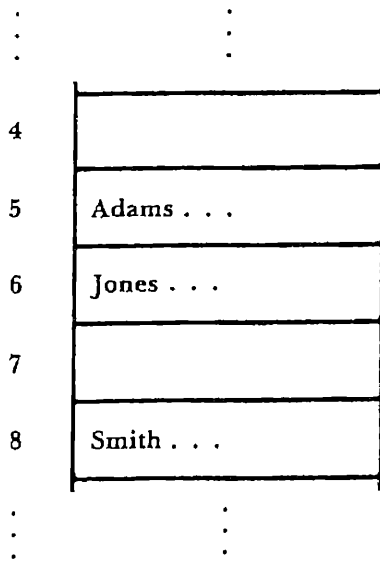


Figure 11.10 The same organization as in Fig. 11.9, with Morris deleted.

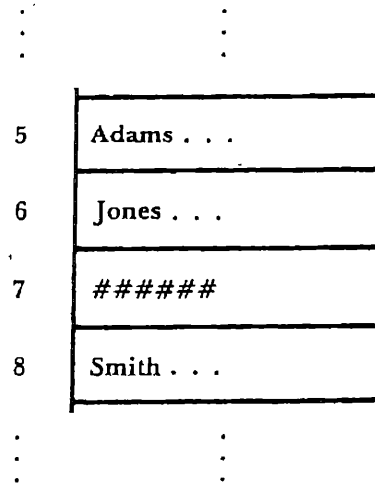


Figure 11.11 The same file as in Fig. 11.9 after the insertion of a tombstone for Morris.

does *not* halt at the empty record number 7. Instead, it uses the ##### as an indication that it should continue the search.

It is not necessary to insert tombstones every time a deletion occurs. For example, suppose in the preceding example that the record for Smith is to be deleted. Since the slot following the Smith record is empty, nothing is lost by marking Smith's slot as empty rather than inserting a tombstone. Indeed, it is unwise to insert a tombstone where it is not needed. (If, after putting an unnecessary tombstone in Smith's slot, a new record is added at address 9, how would a subsequent unsuccessful search for Smith be affected?)

11.7.2 Implications of Tombstones for Insertions

With the introduction of the use of tombstones, the *insertion* of records becomes slightly more difficult than our earlier discussions imply. Whereas programs that perform initial loading simply search for the first occurrence of an empty record slot (signified by the presence of the key /////), it is now permissible to insert a record where either ///// or ##### occurs as the key.

This new feature, which is desirable because it yields a shorter average search length, brings with it a certain danger. Consider, for example, the earlier example in which Morris is deleted, giving the file organization

shown in Fig. 11.11. Now suppose you want a program to insert Smith into the file. If the program simply searches until it encounters a #####, it never notices that Smith is already in the file. We almost certainly don't want to put a second Smith record into the file, since doing so means that later searches would never find the older Smith record. To prevent this from occurring, the program must examine the entire cluster of contiguous keys and tombstones to ensure that no duplicate key exists, then go back and insert the record in the first available tombstone, if there is one.

11.7.3 Effects of Deletions and Additions on Performance

The use of tombstones enables our search algorithms to work and helps in storage recovery, but one can still expect some deterioration in performance after a number of deletions and additions occur within a file.

Consider, for example, our little four-record file of Adams, Jones, Smith, and Morris. After deleting Morris, Smith is one slot further from its home address than it needs to be. If the tombstone is never to be used to store another record, every retrieval of Smith requires one more access than is necessary. More generally, after a large number of additions and deletions, one can expect to find many tombstones occupying places that could be occupied by records whose home records precede them but that are stored after them. In effect, each tombstone represents an unexploited opportunity to reduce by one the number of locations that must be scanned while searching for these records.

Some experimental studies show that after a 50 percent to 150 percent turnover of records, a hashed file reaches a point of equilibrium, so average search length is as likely to get better as it is to get worse (Bradley, 1982; Peterson, 1957). By this time, however, search performance has deteriorated to the point at which the average record is three times as far (in terms of accesses) from its home address as it would be after initial loading. This means, for example, that if after original loading the average search length is 1.2, it will be about 1.6 after the point of equilibrium is reached.

There are three types of solutions to the problem of deteriorating average search lengths. One involves doing a bit of local reorganizing every time a deletion occurs. For example, the deletion algorithm might examine the records that follow a tombstone to see if the search length can be shortened by moving the record backward toward its home address. Another solution involves completely reorganizing the file after the average search length reaches an unacceptable value. A third type of solution involves using an altogether different collision resolution algorithm.

11.8 Other Collision Resolution Techniques

Despite its simplicity, randomized hashing using progressive overflow with reasonably sized buckets generally performs well. If it does not perform well enough, however, there are a number of variations that may perform even better. In this section we discuss some refinements that can often improve hashing performance when using external storage.

11.8.1 Double Hashing

One of the problems with progressive overflow is that if many records hash to buckets in the same vicinity, clusters of records can form. As the packing density approaches one, this clustering tends to lead to extremely long searches for some records. One method for avoiding clustering is to store overflow records a long way from their home addresses by *double hashing*. With double hashing, when a collision occurs, a second hash function is applied to the key to produce a number c that is relatively prime to the number of addresses.⁴ The value c is added to the home address to produce the overflow address. If the overflow address is already occupied, c is added to it to produce another overflow address. This procedure continues until a free overflow address is found.

Double hashing does tend to spread out the records in a file, but it suffers from a potential problem that is encountered in several improved overflow methods: it violates locality by deliberately moving overflow records some distance from their home addresses, increasing the likelihood that the disk will need extra time to get to the new overflow address. If the file covers more than one cylinder, this could require an expensive extra head movement. Double hashing programs can solve this problem if they are able to generate overflow addresses in such a way that overflow records are kept on the same cylinder as home records.

11.8.2 Chained Progressive Overflow

Chained progressive overflow is another technique designed to avoid the problems caused by clustering. It works in the same manner as progressive overflow, except that synonyms are linked together with pointers. That is, each home address contains a number indicating the location of the next

4. If N is the number of addresses, then c and N are relatively prime if they have no common divisors.

Key	Home address	Actual address	Search length
Adams	20	20	1
Bates	21	21	1
Cole	20	22	3
Dean	21	23	3
Evans	24	24	1
Flint	20	25	6

Average search length = $(1 + 1 + 3 + 3 + 1 + 6)/6 = 2.5$

Figure 11.12 Hashing with progressive overflow.

record with the same home address. The next record in turn contains a pointer to the following record with the same home address, and so forth. The net effect of this is that for each set of synonyms there is a linked list connecting their records, and it is this list that is searched when a record is sought.

The advantage of chained progressive overflow over simple progressive overflow is that only records with keys that are synonyms need to be accessed in any given search. Suppose, for example, that the set of keys shown in Fig. 11.12 is to be loaded in the order shown into a hash file with bucket size 1, and progressive overflow is used. A search for Cole involves an access to Adams (a synonym) and Bates (not a synonym). Flint, the worst case, requires six accesses, only two of which involve synonyms.

Since Adams, Cole, and Flint are synonyms, a chaining algorithm forms a linked list connecting these three names, with Adams at the head of the list. Since Bates and Dean are also synonyms, they form a second list. This arrangement is illustrated in Fig. 11.13. The average search length decreases from 2.5 to

$$\frac{1 + 1 + 2 + 2 + 1 + 3}{6} = 1.7$$

The use of chained progressive overflow requires that we attend to some details that are not required for simple progressive overflow. First, a *link field* must be added to each record, requiring the use of a little more storage. Second, a chaining algorithm must guarantee that it is possible to get to any synonym by starting at its home address. This second requirement is not a trivial one, as the following example shows.

Suppose that in the example Dean's home address is 22 instead of 21. Since, by the time Dean is loaded, address 22 is already occupied by Cole,

Home address	Actual address	Data	Address of next synonym	Search length
20	20	Adams . . .	22	1
21	21	Bates . . .	23	1
20	22	Cole . . .	25	2
21	23	Dean . . .	-1	2
24	24	Evans . . .	-1	1
20	25	Flint . . .	-1	3

Figure 11.13 Hashing with chained progressive overflow. Adams, Cole, and Flint are synonyms; Bates and Dean are synonyms.

Dean still ends up at address 23. Does this mean that Cole's pointer should point to 23 (Dean's actual address) or to 25 (the address of Cole's synonym Flint)? If the pointer is 25, the linked list joining Adams, Cole, and Flint is kept intact, but Dean is lost. If the pointer is 23, Flint is lost.

The problem here is that a certain address (22) that *should* be occupied by a home record (Dean) is occupied by a different record. One solution to the problem is to require that every address qualifying as a home address for some record in the file actually hold a home record. The problem can be handled easily when a file is first loaded by using a technique called two-pass loading.

Two-pass loading, as the name implies, involves loading a hash file in two passes. On the first pass, only home records are loaded. All records that are not home records are kept in a separate file. This guarantees that no potential home addresses are occupied by overflow records. On the second pass, each overflow record is loaded and stored in one of the free addresses according to whatever collision resolution technique is being used.

Two-pass loading guarantees that every potential home address actually is a home address, so it solves the problem in the example. It does not guarantee that later deletions and additions will not re-create the same problem, however. As long as the file is used to store both home records and overflow records, there remains the problem of overflow records

displacing new records that hash to an address occupied by an overflow record.

The methods used for handling these problems after initial loading are somewhat complicated and can, in a very volatile file, require many extra disk accesses. (For more information on techniques for maintaining pointers, see Knuth, 1998 and Bradley, 1982.) It would be nice if we could somehow altogether avoid this problem of overflow lists bumping into one another, and that is what the next method does.

11.8.3 Chaining with a Separate Overflow Area

One way to keep overflow records from occupying home addresses where they should not be is to move them all to a separate overflow area. Many hashing schemes are variations of this basic approach. The set of home addresses is called the *prime data area*, and the set of overflow addresses is called the *overflow area*. The advantage of this approach is that it keeps all unused but potential home addresses free for later additions.

In terms of the file we examined in the preceding section, the records for Cole, Dean, and Flint could have been stored in a separate overflow area rather than in potential home addresses for later-arriving records (Fig. 11.14). Now no problem occurs when a new record is added. If its home address has room, it is stored there. If not, it is moved to the overflow file, where it is added to the linked list that starts at the home address.

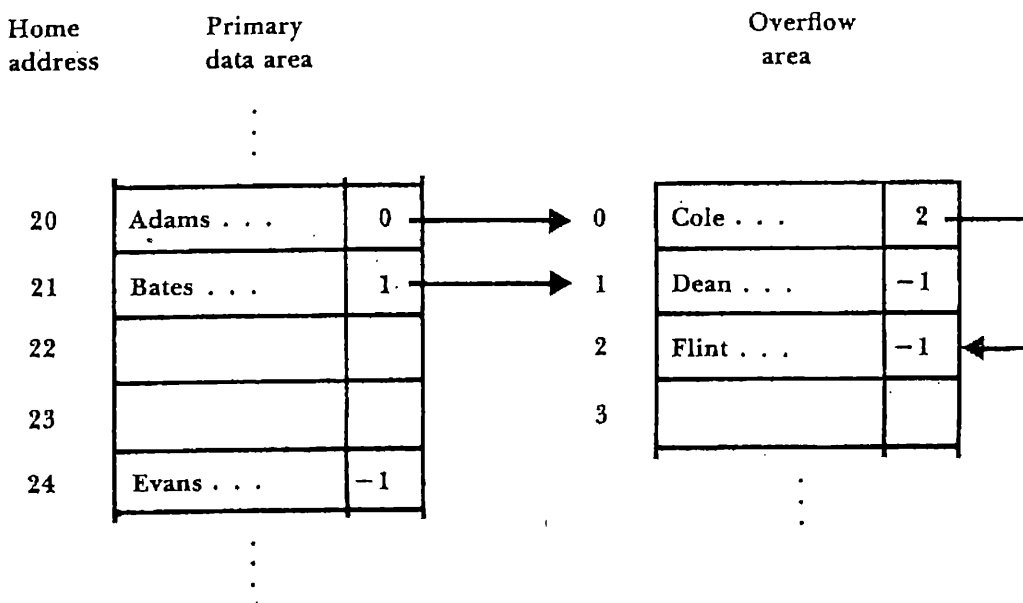


Figure 11.14 Chaining to a separate overflow area. Adams, Cole, and Flint are synonyms; Bates and Dean are synonyms.

If the bucket size for the primary file is large enough to prevent excessive numbers of overflow records, the overflow file can be a simple entry-sequenced file with a bucket size of 1. Space can be allocated for overflow records only when it is needed.

The use of a separate overflow area simplifies processing somewhat and would seem to improve performance, especially when many additions and deletions occur. However, this is not always the case. If the separate overflow area is on a different cylinder than is the home address, every search for an overflow record will involve a very costly head movement. Studies show that access time is generally worse when overflow records are stored in a separate overflow area than when they are stored in the prime overflow area (Lum, 1971).

One situation in which a separate overflow area is *required* occurs when the packing density is greater than one—there are more records than home addresses. If, for example, it is anticipated that a file will grow beyond the capacity of the initial set of home addresses and that rehashing the file with a larger address space is not reasonable, then a separate overflow area must be used.

11.8.4 Scatter Tables: Indexing Revisited

Suppose you have a hash file that contains no records, only pointers to records. The file is obviously just an index that is searched by hashing rather than by some other method. The term *scatter table* (Severance, 1974) is often applied to this approach to file organization. Figure 11.15 illustrates the organization of a file using a scatter table.

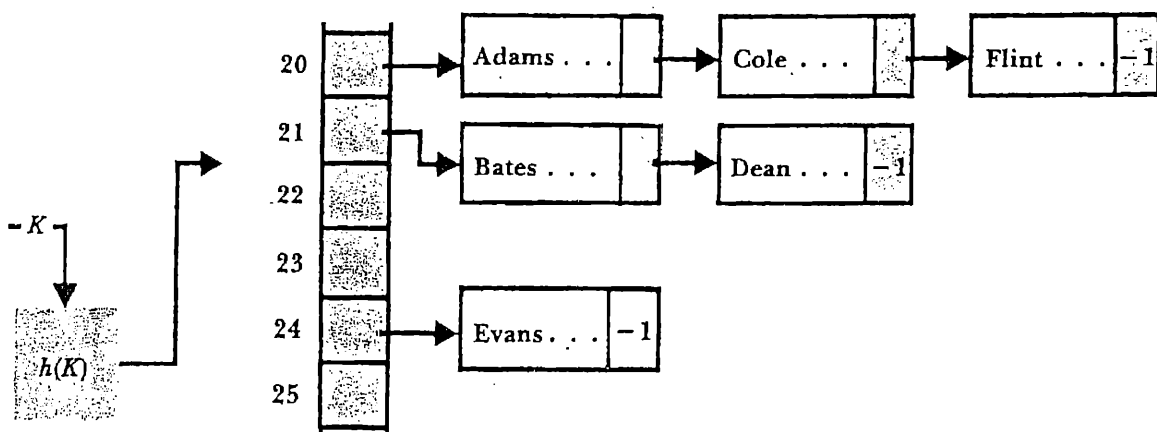


Figure 11.15 Example of a scatter table structure. Because the hashed part is an index, the data file may be organized in any way that is appropriate.

The scatter table organization provides many of the same advantages simple indexing generally provides, with the additional advantage that the search of the index itself requires only one access. (Of course, that one access is one more than other forms of hashing require, unless the scatter table can be kept in primary memory.) The data file can be implemented in many different ways. For example, it can be a set of linked lists of synonyms (as shown in Fig. 11.15), a sorted file, or an entry-sequenced file. Also, scatter table organizations conveniently support the use of variable-length records. For more information on scatter tables, see Severance (1974).

11.9 Patterns of Record Access

Twenty percent of the fishermen catch 80 percent of the fish.
Twenty percent of the burglars steal 80 percent of the loot.

L.M. Boyd

The use of different collision resolution techniques is not the only nor necessarily the best way to improve performance in a hashed file. If we know something about the patterns of record access, for example, then it is often possible to use simple progressive overflow techniques and still achieve very good performance.

Suppose you have a grocery store with 10 000 different categories of grocery items and you have on your computer a hashed inventory file with a record for each of the 10 000 items that your company handles. Every time an item is purchased, the record that corresponds to that item must be accessed. Since the file is hashed, it is reasonable to assume that the 10 000 records are distributed randomly among the available addresses that make up the file. Is it equally reasonable to assume that the distribution of *accesses* to the records in the inventory are randomly distributed? Probably not. Milk, for example, will be retrieved very frequently, brie seldom.

There is a principle used by economists called the Pareto Principle, or The Concept of the Vital Few and the Trivial Many, which in file terms says that a small percentage of the records in a file account for a large percentage of the accesses. A popular version of the Pareto Principle is the 80/20 Rule of Thumb: 80 percent of the accesses are performed on 20 percent of the records. In our groceries file, milk would be among the 20 percent high-activity items, brie among the rest.

We cannot take advantage of the 80/20 principle in a file structure unless we know something about the probable distribution of record accesses. Once we have this information, we need to find a way to place the high-activity items where they can be found with as few accesses as possible. If, when items are loaded into a file, they can be loaded in such a way that the 20 percent (more or less) that are most likely to be accessed are loaded at or near their home addresses, then most of the transactions will access records that have short search lengths, so the *effective* average search length will be shorter than the nominal average search length that we defined earlier.

For example, suppose our grocery store's file handling program keeps track of the number of times each item is accessed during a one-month period. It might do this by storing with each record a counter that starts at zero and is incremented every time the item is accessed. At the end of the month the records for all the items in the inventory are dumped onto a file that is sorted in descending order according to the number of times they have been accessed. When the sorted file is rehashed and reloaded, the first records to be loaded are the ones that, according to the previous month's experience, are most likely to be accessed. Since they are the first ones loaded, they are also the ones most likely to be loaded into their home addresses. If reasonably sized buckets are used, there will be *very few*, if any, high-activity items that are not in their home addresses and therefore retrievable in one access.

S U M M A R Y

There are three major modes for accessing files: *sequentially*, which provides $O(N)$ performance, through *tree structures*, which can produce $O(\log_k N)$ performance, and *directly*. Direct access provides $O(1)$ performance, which means that the number of accesses required to retrieve a record is constant and independent of the size of the file. Hashing is the primary form of organization used to provide direct access.

Hashing can provide faster access than most of the other organizations we study, usually with very little storage overhead, and it is adaptable to most types of primary keys. Ideally, hashing makes it possible to find any record with only one disk access, but this ideal is rarely achieved. The primary disadvantage of hashing is that hashed files may not be sorted by key.

Hashing involves the application of a hash function $h(K)$ to a record key K to produce an address. The address is taken to be the *home address* of the record whose key is K , and it forms the basis for searching for the record. The addresses produced by hash functions generally appear to be random.

When two or more keys hash to the same address, they are called *synonyms*. If an address cannot accommodate all of its synonyms, *collisions* result. When collisions occur, some of the synonyms cannot be stored in the home address and must be stored elsewhere. Since searches for records begin with home addresses, searches for records that are not stored at their home addresses generally involve extra disk accesses. The term *average search length* is used to describe the average number of disk accesses that are required to retrieve a record. An average search length of 1 is ideal.

Much of the study of hashing deals with techniques for decreasing the number and effects of collisions. In this chapter we look at three general approaches to reducing the number of collisions:

- Spreading out the records;
- Using extra memory; and
- Using buckets.

Spreading out the records involves choosing a hashing function that distributes the records at least randomly over the address space. A *uniform distribution* spreads out records evenly, resulting in no collisions. A *random* or nearly random distribution is much easier to achieve and is usually considered acceptable.

In this chapter a simple hashing algorithm is developed to demonstrate the kinds of operations that take place in a hashing algorithm. The three steps in the algorithm are:

1. Represent the key in numerical form;
2. Fold and add; and
3. Divide by the size of the address space, producing a valid address.

When we examine several different types of hashing algorithms, we see that sometimes algorithms can be found that produce *better-than-random* distributions. Failing this, we suggest some algorithms that generally produce distributions that are approximately random.

The *Poisson distribution* provides a mathematical tool for examining in detail the effects of a random distribution. Poisson functions can be used to predict the numbers of addresses likely to be assigned 0, 1, 2, and

so on, records, given the number of records to be hashed and the number of available addresses. This allows us to predict the number of collisions likely to occur when a file is hashed, the number of overflow records likely to occur, and sometimes the average search length.

Using extra memory is another way to avoid collisions. When a fixed number of keys is hashed, the likelihood of synonyms occurring decreases as the number of possible addresses increases. Hence, a file organization that allocates many more addresses than are likely to be used has fewer synonyms than one that allocates few extra addresses. The term *packing density* describes the proportion of available address space that actually holds records. The Poisson function is used to determine how differences in packing density influence the percentage of records that are likely to be synonyms.

Using buckets is the third method for avoiding collisions. File addresses can hold one or more records, depending on how the file is organized by the file designer. The number of records that can be stored at a given address, called *bucket size*, determines the point at which records assigned to the address will overflow. The Poisson function can be used to explore the effects of variations in bucket sizes and packing densities. Large buckets, combined with a low packing density, can result in very small average search lengths.

Although we can reduce the number of collisions, we need some means to deal with collisions when they do occur. We examined one simple collision resolution technique in detail—*progressive overflow*. If an attempt to store a new record results in a collision, progressive overflow involves searching through the addresses that follow the record's home address in order until one is found to hold the new record. If a record is sought and is not found in its home address, successive addresses are searched until either the record is found or an empty address is encountered.

Progressive overflow is simple and sometimes works very well. However, progressive overflow creates long search lengths when the packing density is high and the bucket size is low. It also sometimes produces clusters of records, creating very long search lengths for new records whose home addresses are in the clusters.

Three problems associated with record deletion in hashed files are

1. The possibility that empty slots created by deletions will hinder later searches for overflow records;

2. The need to recover space made available when records are deleted; and
3. The deterioration of average search lengths caused by empty spaces keeping records further from home than they need be.

The first two problems can be solved by using *tombstones* to mark spaces that are empty (and can be reused for new records) but should not halt a search for a record. Solutions to the deterioration problem include local reorganization, complete file reorganization, and the choice of a collision-resolving algorithm that does not cause deterioration to occur.

Because overflow records have a major influence on performance, many different overflow handling techniques have been proposed. Four such techniques that are appropriate for file applications are discussed briefly:

1. *Double hashing* reduces local clustering but may place some overflow records so far from home that they require extra seeks.
2. *Chained progressive overflow* reduces search lengths by requiring that only synonyms be examined when a record is being sought. For chained overflow to work, every address that qualifies as a home record for some record in the file must hold a home record. Mechanisms for making sure that this occurs are discussed.
3. *Chaining with a separate overflow area* simplifies chaining substantially and has the advantage that the overflow area may be organized in ways more appropriate to handling overflow records. A danger of this approach is that it might lose locality.
4. *Scatter tables* combine indexing with hashing. This approach provides much more flexibility in organizing the data file. A disadvantage of using scatter tables is that, unless the index can be held in memory, it requires one extra disk access for every search.

Since in many cases certain records are accessed more frequently than others (the *80/20 rule of thumb*), it is often worthwhile to take access patterns into account. If we can identify those records that are most likely to be accessed, we can take measures to make sure they are stored closer to home than less frequently accessed records, thus decreasing the *effective* average search length. One such measure is to load the most frequently accessed records before the others.

KEY TERMS

Average search length. We define average search length as the *sum of the number of accesses required for each record in the file* divided by the *number of records in the file*. This definition does not take into account the number of accesses required for unsuccessful searches, nor does it account for the fact that some records are likely to be accessed more often than others. See *80/20 rule of thumb*.

Better-than-random. This term is applied to distributions in which the records are spread out more uniformly than they would be if the hash function distributed them randomly. Normally, the distribution produced by a hash function is a little bit better than random.

Bucket. An area of space on the file that is treated as a physical record for storage and retrieval purposes but is capable of storing several *logical* records. By storing and retrieving logical records in buckets rather than individually, access times can, in many cases, be improved substantially.

Collision. Situation in which a record is hashed to an address that does not have sufficient room to store the record. When a collision occurs, some means has to be found to resolve the collision.

Double hashing. A collision resolution scheme in which collisions are handled by applying a second hash function to the key to produce a number c , which is added to the original address (modulo the number of addresses) as many times as necessary until either the desired record is located or an empty space is found. Double hashing helps avoid some of the clustering that occurs with progressive overflow.

The 80/20 rule of thumb. An assumption that a large percentage (e.g., 80 percent) of the accesses are performed on a small percentage (e.g., 20 percent) of the records in a file. When the 80/20 rule applies, the *effective* average search length is determined largely by the search lengths of the more active records, so attempts to make *these* search lengths short can result in substantially improved performance.

Fold and add. A method of hashing in which the encodings of fixed-sized parts of a key are extracted (e.g., every two bytes) and are added. The resulting sum can be used to produce an address.

Hashing. A technique for generating a unique home address for a given key. Hashing is used when rapid access to a key (or its corresponding record) is required. In this chapter applications of hashing involve

direct access to records in a file, but hashing is also often used to access items in arrays in memory. In indexing, for example, an index might be organized for hashing rather than for binary search if extremely fast searching of the index is desired.

Home address. The address generated by a hash function for a given key. If a record is stored at its home address, then the search length for the record is 1 because only one access is required to retrieve the record. A record not at its home address requires more than one access to retrieve or store.

Indexed hash. Instead of using the results of a hash to produce the address of a record, the hash can be used to identify a location in an index that in turn points to the address of the record. Although this approach requires one extra access for every search, it makes it possible to organize the data records in a way that facilitates other types of processing, such as sequential processing.

Mid-square method. A hashing method in which a representation of the key is squared and some digits from the middle of the result are used to produce the address.

Minimum hashing. Hashing scheme in which the number of addresses is exactly equal to the number of records. No storage space is wasted.

Open addressing. See *progressive overflow*.

Overflow. The situation that occurs when a record cannot be stored in its home address.

Packing density. The proportion of allocated file space that actually holds records. (This is sometimes referred to as *load factor*.) If a file is half full, its packing density is 50 percent. The packing density and bucket size are the two most important measures in determining the likelihood of a collision occurring when searching for a record in a file.

Perfect hashing function. A hashing function that distributes records uniformly, minimizing the number of collisions. Perfect hashing functions are very desirable, but they are extremely difficult to find for large sets of keys.

Poisson distribution. Distribution generated by the Poisson function, which can be used to approximate the distribution of records among addresses if the distribution is random. A particular Poisson distribution depends on the ratio of the number of records to the number of available addresses. A particular instance of the Poisson function, $p(x)$,

gives the proportion of addresses that will have x keys assigned to them. See *better-than-random*.

Prime division. Division of a number by a prime number and use of the remainder as an address. If the address size is taken to be a prime number p , a large number can be transformed into a valid address by dividing it by p . In hashing, division by primes is often preferred to division by nonprimes because primes tend to produce more random remainders.

Progressive overflow. An overflow handling technique in which collisions are resolved by storing a record in the next available address after its home address. Progressive overflow is not the most efficient overflow handling technique, but it is one of the simplest and is adequate for many applications.

Randomize. To produce a number (e.g., by hashing) that appears to be random.

Synonyms. Two or more different keys that hash to the same address. When each file address can hold only one record, synonyms always result in collisions. If buckets are used, several records whose keys are synonyms may be stored without collisions.

Tombstone. A special marker placed in the key field of a record to mark it as no longer valid. The use of tombstones solves two problems associated with the deletion of records: the freed space does not break a sequential search for a record, and the freed space is easily recognized as available and may be reclaimed for later additions.

Uniform. A distribution in which records are spread out evenly among addresses. Algorithms that produce uniform distributions are better than randomizing algorithms in that they tend to avoid the numbers of collisions that would occur with a randomizing algorithm.

FURTHER READINGS

There are a number of good surveys of hashing and issues related to hashing generally, including Knuth (1998), Severance (1974), Maurer (1975), and Sorenson, Tremblay, and Deutscher (1978). Textbooks concerned with file design generally contain substantial amounts of material on hashing, and they often provide extensive references for further study. Loomis (1989) also covers hashing generally, with additional emphasis on pro-

gramming for hashed files in COBOL. Cormen, Leiserson and Rivest (1990), Standish (1995), Shaffer (1997), and Wiederhold (1983) will be useful to practitioners interested in analyses of trade-offs among the basic hashing methods.

One of the applications of hashing that has stimulated a great deal of interest recently is the development of *spelling checkers*. Because of special characteristics of spelling checkers, the types of hashing involved are quite different from the approaches we describe in this text. Papers by Bentley (1985) and Dodds (1982) provide entry into the literature on this topic. (See also exercise 14.)

EXERCISES

1. Use the function $\text{hash}(\text{KEY}, \text{MAXAD})$ described in the text to answer the following questions.
 - a. What is the value of $\text{hash}(\text{"Jacobs"}, 101)$?
 - b. Find two different words of more than four characters that are synonyms.
 - c. It is assumed in the text that the function hash does not need to generate an integer greater than 19 937. This could present a problem if we have a file with addresses larger than 19 937. Suggest some ways to get around this problem.
2. In understanding hashing, it is important to understand the relationships between the size of the available memory, the number of keys to be hashed, the range of possible keys, and the nature of the keys. Let us give names to these quantities, as follows:
 - M = the number of memory spaces available (each capable of holding one record);
 - r = the number of records to be stored in the memory spaces; n = the number of unique home addresses produced by hashing the r record keys; and
 - K = a key, which may be any combination of exactly five uppercase characters.

Suppose $h(K)$ is a hash function that generates addresses between 0 and $M - 1$.

- a. How many unique keys are possible? (Hint: If K were one uppercase letter rather than five, there would be 26 possible unique keys.)
 - b. How are n and r related?
 - c. How are r and M related?
 - d. If the function h were a minimum perfect hashing function, how would n , r , and M be related?
3. The following table shows distributions of keys resulting from three different hash functions on a file with 6000 records and 6000 addresses.

	<i>Function A</i>	<i>Function B</i>	<i>Function C</i>
$d(0)$	0.71	0.25	0.40
$d(1)$	0.05	0.50	0.36
$d(2)$	0.05	0.25	0.15
$d(3)$	0.05	0.00	0.05
$d(4)$	0.05	0.00	0.02
$d(5)$	0.04	0.00	0.01
$d(6)$	0.05	0.00	0.01
$d(7)$	0.00	0.00	0.00

- a. Which of the three functions (if any) generates a distribution of records that is approximately random?
 - b. Which generates a distribution that is nearest to uniform?
 - c. Which (if any) generates a distribution that is worse than random?
 - d. Which function should be chosen?
4. There is a surprising mathematical result called *the birthday paradox* that says that if there are more than 23 people in a room, there is a better than 50-50 chance that two of them have the same birthday. How is the birthday paradox illustrative of a major problem associated with hashing?
5. Suppose that 10 000 addresses are allocated to hold 8000 records in a randomly hashed file and that each address can hold one record. Compute the following values:
- a. The packing density for the file;
 - b. The expected number of addresses with no records assigned to them by the hash function;
 - c. The expected number of addresses with one record assigned (no synonyms);

- d. The expected number of addresses with one record *plus* one or more synonyms;
 - e. The expected number of overflow records; and
 - f. The expected percentage of overflow records.
6. Consider the file described in the preceding exercise. What is the expected number of overflow records if the 10 000 locations are reorganized as
 - a. 5000 two-record buckets; and
 - b. 1000 ten-record buckets?
 7. Make a table showing Poisson function values for $r/N = 0.1, 0.5, 0.8, 1, 2, 5,$ and 11 . Examine the table and discuss any features and patterns that provide useful information about hashing.
 8. There is an overflow handling technique called *count-key progressive overflow* (Bradley, 1982) that works on block-addressable disks as follows. Instead of generating a relative record number from a key, the hash function generates an address consisting of three values: a cylinder, a track, and a block number. The corresponding three numbers constitute the home address of the record.

Since block-organized drives (see Chapter 3) can often scan a track to find a record with a given key, there is no need to load a block into memory to find out whether it contains a particular record. The I/O processor can direct the disk drive to search a track for the desired record. It can even direct the disk to search for an empty record slot if a record is not found in its home position, effectively implementing progressive overflow.

- a. What is it about this technique that makes it superior to progressive overflow techniques that might be implemented on sector-organized drives?
 - b. The main disadvantage of this technique is that it can be used only with a bucket size of 1. Why is this the case, and why is it a disadvantage?
9. In discussing implementation issues, we suggest initializing the data file by creating real records that are marked empty before loading the file with data. There are some good reasons for doing this. However, there might be some reasons not to do it this way. For example, suppose you want a hash file with a very low packing density and cannot afford to have the unused space allocated. How might a file management system be designed to work with a very large *logical* file but allocate space only for those blocks in the file that contain data?

10. This exercise (inspired by an example in Wiederhold, 1983, p. 136) concerns the problem of deterioration. A number of additions and deletions are to be made to a file. Tombstones are to be used where necessary to preserve search paths to overflow records.
- a. Show what the file looks like after the following operations, and compute the average search length.

<i>Operation</i>	<i>Home Address</i>
Add Alan	0
Add Bates	2
Add Cole	4
Add Dean	0
Add Evans	1
Del Bates	
Del Cole	
Add Finch	0
Add Gates	2
Del Alan	
Add Hart	3

- How has the use of tombstones caused the file to deteriorate?
 - What would be the effect of reloading the remaining items in the file in the order Dean, Evans, Finch, Gates, Hart?
- b. What would be the effect of reloading the remaining items using two-pass loading?
11. Suppose you have a file in which 20 percent of the records account for 80 percent of the accesses and that you want to store the file with a packing density of 0 and a bucket size of 5. When the file is loaded, you load the active 20 percent of the records first. After the active 20 percent of the records are loaded and before the other records are loaded, what is the packing density of the partially filled file? Using this packing density, compute the percentage of the active 20 percent that would be overflow records. Comment on the results.
12. In our computations of average search lengths, we consider only the times it takes for *successful* searches. If our hashed file were to be used in such a way that searches were often made for items that are not in the file, it would be useful to have statistics on average search length for an *unsuccessful* search. If a large percentage of searches to a hashed file are unsuccessful, how do you expect this to affect overall performance if overflow is handled by

- a. Progressive overflow; or
- b. Chaining to a separate overflow area?

(See Knuth, 1973b, pp. 535–539 for a treatment of these differences.)

13. Although hashed files are not generally designed to support access to records in any sorted order, there may be times when batches of transactions need to be performed on a hashed data file. If the data file is sorted (rather than hashed), these transactions are normally carried out by some sort of cosequential process, which means that the transaction file also has to be sorted. If the data file is hashed, the transaction file might also be presorted, but on the basis of the home addresses of its records rather than some more “natural” criterion.

Suppose you have a file whose records are usually accessed directly but is periodically updated from a transaction file. List the factors you would have to consider in deciding between using an indexed sequential organization and hashing. (See Hanson, 1982, pp. 280–285, for a discussion of these issues.)

14. We assume throughout this chapter that a hashing program should be able to tell correctly whether a given key is located at a certain address. If this were not so, there would be times when we would assume that a record exists when in fact it does not, a seemingly disastrous result. But consider what Doug McIlroy did in 1978 when he was designing a spelling checker program. He found that by letting his program allow one out of every four thousand misspelled words to sneak by as valid (and using a few other tricks), he could fit a 75 000-word spelling dictionary into 64 kilobytes of memory, thereby improving performance enormously.

McIlroy was willing to tolerate one undetected misspelled word out of every four thousand because he observed that drafts of papers rarely contained more than twenty errors, so one could expect at most one out of every two hundred runs of the program to fail to detect a misspelled word. Can you think of some other cases in which it might be reasonable to report that a key exists when in fact it does not?

Jon Bentley (1985) provides an excellent account of McIlroy’s program, plus several insights on the process of solving problems of this nature. D. J. Dodds (1982) discusses this general approach to hashing, called *check-hashing*. Read Bentley’s and Dodds’s articles and report on them to your class. Perhaps they will inspire you to write a spelling checker.

PROGRAMMING EXERCISES

15. Implement and test a version of the function hash.
16. Create a hashed file with one record for every city in California. The key in each record is to be the name of the corresponding city. (For the purposes of this exercise, there need be no fields other than the key field.) Begin by creating a sorted list of the names of all of the cities and towns in California. (If time or space is limited, just make a list of names starting with the letter S.)
 - a. Examine the sorted list. What patterns do you notice that might affect your choice of a hash function?
 - b. Implement the function hash in such a way that you can alter the number of characters that are folded. Assuming a packing density of 1, hash the entire file several times, each time folding a different number of characters and producing the following statistics for each run:
 - The number of collisions; and
 - The number of addresses assigned 0, 1, 2, . . . , 10, and 10-or-more records.Discuss the results of your experiment in terms of the effects of folding different numbers of characters and how they compare with the results you might expect from a random distribution.
 - c. Implement and test one or more of the other hashing methods described in the text, or use a method of your own invention.
17. Using a set of keys, such as the names of California towns, do the following:
 - a. Write and test a program for loading the keys into three different hash files using bucket sizes of 1, 2, and 5, respectively, and a packing density of 0.8. Use progressive overflow for handling collisions.
 - b. Have your program maintain statistics on the average search length, the maximum search length, and the percentage of records that are overflow records.
 - c. Assuming a Poisson distribution, compare your results with the expected values for average search length and the percentage of records that are overflow records.
18. Repeat exercise 17, but use double hashing to handle overflow.

19. Repeat exercise 17, but handle overflow using chained overflow into a separate overflow area. Assume that the packing density is the ratio of number of keys to available *home* addresses.
20. Write a program that can perform insertions and deletions in the file created in the previous problem using a bucket size of 5. Have the program keep running statistics on average search length. (You might also implement a mechanism to indicate when search length has deteriorated to a point where the file should be reorganized.) Discuss in detail the issues you have to confront in deciding how to handle insertions and deletions.

Extendible Hashing

CHAPTER OBJECTIVES

- ❖ Describe the problem solved by extendible hashing and related approaches.
- ❖ Explain how extendible hashing works; show how it combines *tries* with conventional, static hashing.
- ❖ Use the `buffer`, `file`, and `index` classes of previous chapters to implement extendible hashing, including deletion.
- ❖ Review studies of extendible hashing performance.
- ❖ Examine alternative approaches to the same problem, including *dynamic hashing*, *linear hashing*, and hashing schemes that control splitting by allowing for overflow buckets.

CHAPTER OUTLINE

- 12.1 Introduction
- 12.2 How Extendible Hashing Works
 - 12.2.1 Tries
 - 12.2.2 Turning the Trie into a Directory
 - 12.2.3 Splitting to Handle Overflow
- 12.3 Implementation
 - 12.3.1 Creating the Addresses
 - 12.3.2 Classes for Representing Bucket and Directory Objects
 - 12.3.3 Bucket and Directory Operations
 - 12.3.4 Implementation Summary
- 12.4 Deletion
 - 12.4.1 Overview of the Deletion Process
 - 12.4.2 A Procedure for Finding Buddy Buckets
 - 12.4.3 Collapsing the Directory
 - 12.4.4 Implementing the Deletion Operations
 - 12.4.5 Summary of the Deletion Operation
- 12.5 Extendible Hashing Performance
 - 12.5.1 Space Utilization for Buckets
 - 12.5.2 Space Utilization for the Directory
- 12.6 Alternative Approaches
 - 12.6.1 Dynamic Hashing
 - 12.6.2 Linear Hashing
 - 12.6.3 Approaches to Controlling Splitting

12.1 Introduction

In Chapter 9 we began with a historical review of the work that led up to B-trees. B-trees are such an effective solution to the problems that stimulated their development that it is easy to wonder if there is any more important thinking to be done about file structures. Work on extendible hashing during the late 1970s and early 1980s shows that the answer to that question is yes. This chapter tells the story of that work and describes some of the file structures that emerge from it.

B-trees do for secondary storage what AVL trees do for storage in memory: they provide a way of using tree structures that works well with *dynamic* data. By *dynamic* we mean that records are added and deleted from the data set. The key feature of both AVL trees and B-trees is that they are self-adjusting structures that include mechanisms to maintain

themselves. As we add and delete records, the tree structures use limited, local restructuring to ensure that the additions and deletions do not degrade performance beyond some predetermined level.

Robust, self-adjusting data and file structures are critically important to data storage and retrieval. Judging from the historical record, they are also hard to develop. It was not until 1963 that Adel'son-Vel'skii and Landis developed a self-adjusting structure for tree storage in memory, and it took another decade of work before computer scientists found, in B-trees, a dynamic tree structure that works well on secondary storage.

B-trees provide $O(\log_k N)$ access to the keys in a file. Hashing, when there is no overflow, provides access to a record with a single seek. But as a file grows larger, looking for records that overflow their buckets degrades performance. For dynamic files that undergo a lot of growth, the performance of a static hashing system such as we described in Chapter 11 is typically worse than the performance of a B-tree. So, by the late 1970s, after the initial burst of research and design work revolving around B-trees was over, a number of researchers began to work on finding ways to modify hashing so that it, too, could be self-adjusting as files grow and shrink. As often happens when a number of groups are working on the same problem, several different, yet essentially similar, approaches emerged to extend hashing to dynamic files. We begin our discussion of the problem by looking closely at the approach called "extendible hashing" described by Fagin, Nievergelt, Pippenger, and Strong (1979). Later in this chapter we compare this approach with others that emerged more recently.

12.2 How Extendible Hashing Works

12.2.1 Tries

The key idea behind extendible hashing is to combine conventional hashing with another retrieval approach called the *trie*. (The word *trie* is pronounced so that it rhymes with *sky*.) Tries are also sometimes referred to as *radix searching* because the branching factor of the search tree is equal to the number of alternative symbols (the radix of the alphabet) that can occur in each position of the key. A few examples will illustrate how this works.

Suppose we want to build a trie that stores the keys *able*, *abrahms*, *adams*, *anderson*, *andrews*, and *baird*. A schematic form of the trie is shown in Fig. 12.1. As you can see, the searching proceeds letter by letter

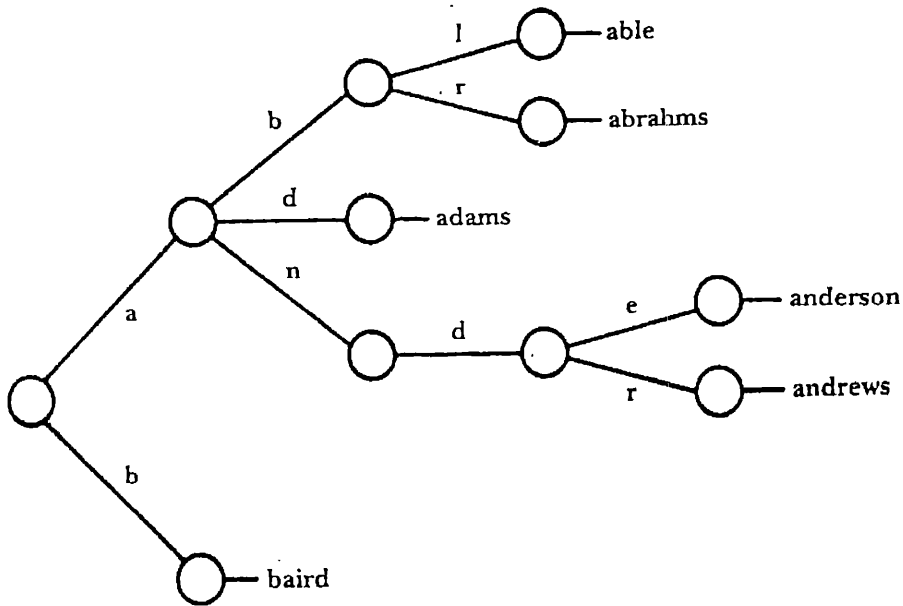


Figure 12.1 Radix 26 trie that indexes names according to the letters of the alphabet.

through the key. Because there are twenty-six symbols in the alphabet, the potential branching factor at every node of the search is twenty-six. If we used the digits 0–9 as our search alphabet rather than the letters *a–z*, the radix of the search would be reduced to 10. A search tree using digits might look like the one shown in Fig. 12.2.

Notice that in searching a trie we sometimes use only a portion of the key. We use more of the key as we need more information to complete the search. This use-more-as-we-need-more capability is fundamental to the structure of extendible hashing.

12.2.2 Turning the Trie into a Directory

We use tries with a radix of 2 in our approach to extendible hashing: search decisions are made on a bit-by-bit basis. Furthermore, since we are retrieving from secondary storage, we will not work in terms of individual keys but in terms of *buckets* containing keys, just as in conventional hashing. Suppose we have bucket *A* containing keys that, when hashed, have hash addresses that begin with the bits *01*. Bucket *B* contains keys with hash addresses beginning with *10*, and bucket *C* contains keys with addresses that start with *11*. Figure 12.3 shows a trie that allows us to retrieve these buckets.

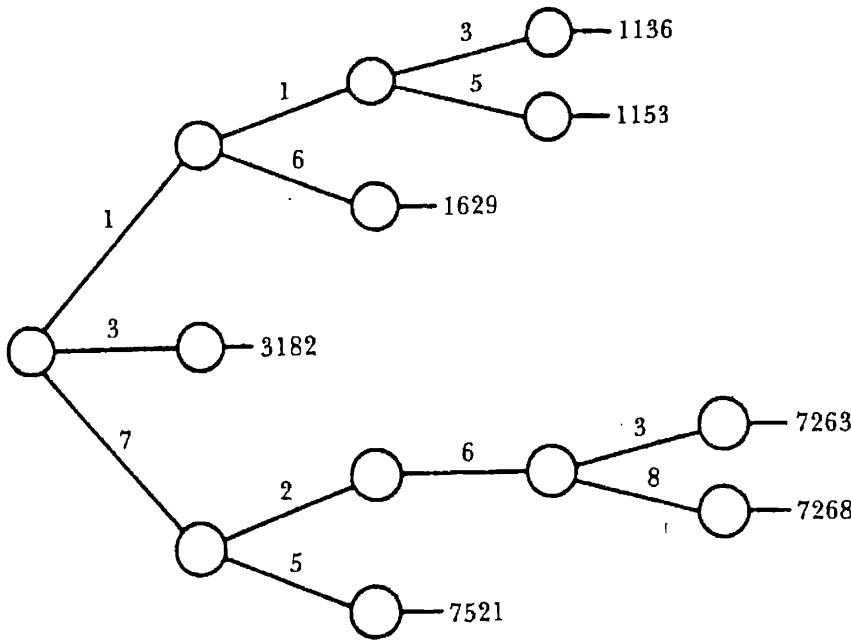


Figure 12.2 Radix 10 trie that indexes numbers according to the digits they contain.

How should we represent the trie? If we represent it as a tree structure, we are forced to do a number of comparisons as we descend the tree. Even worse, if the trie becomes so large that it, too, is stored on disk, we are faced once again with all of the problems associated with storing trees on disk. We might as well go back to B-trees and forget about extendible hashing.

So, rather than representing the trie as a tree, we flatten it into an array of contiguous records, forming a directory of hash addresses and pointers to the corresponding buckets. The first step in turning a tree into an array involves extending it so it is a complete binary tree with all of its leaves at

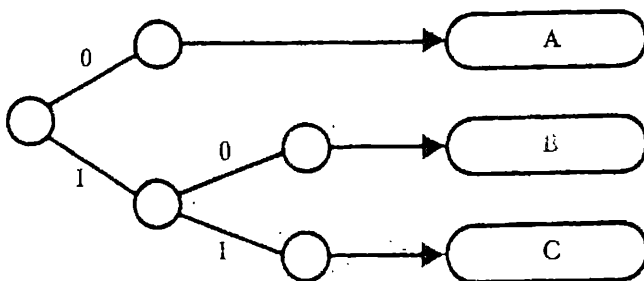


Figure 12.3 Radix 2 trie that provides an index to buckets.

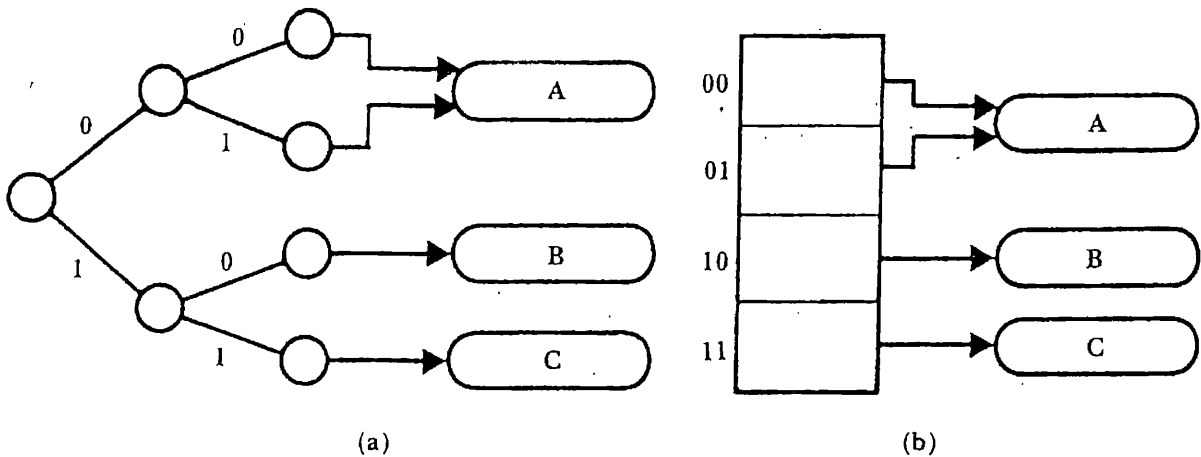


Figure 12.4 The trie from Fig. 12.3 transformed first into a complete binary tree, then flattened into a directory to the buckets.

the same level as shown in Fig. 12.4(a). Even though the initial 0 is enough to select bucket A, the new form of the tree also uses the second address bit so both alternatives lead to the same bucket. Once we have extended the tree this way, we can collapse it into the directory structure shown in Fig. 12.4(b). Now we have a structure that provides the kind of direct access associated with hashing: given an address beginning with the bits 10, the 10_2^{th} directory entry gives us a pointer to the associated bucket.

12.2.3 Splitting to Handle Overflow

A key issue in any hashing system is what happens when a bucket overflows. The goal in an *extendible* hashing system is to find a way to increase the address space in response to overflow rather than respond by creating long sequences of overflow records and buckets that have to be searched linearly.

Suppose we insert records that cause bucket A in Fig. 12.4(b) to overflow. In this case the solution is simple: since addresses beginning with 00 and 01 are mixed together in bucket A, we can split bucket A by putting all the 01 addresses in a new bucket D, while keeping only the 00 addresses in A. Put another way, we already have 2 bits of address information but are throwing 1 away as we access bucket A. So, now that bucket A is overflowing, we must use the full 2 bits to divide the addresses between two buckets. We do not need to extend the address space; we simply make full use of the address information that we already have. Figure 12.5 shows the directory and buckets after the split.

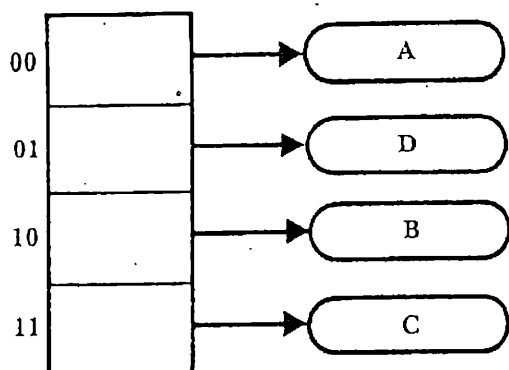


Figure 12.5 The directory from Fig. 12.4(b) after bucket A overflows.

Let's consider a more complex case. Starting once again with the directory and buckets in Fig. 12.4(b), suppose that bucket *B* overflows. How do we split bucket *B* and where do we attach the new bucket after the split? Unlike our previous example, we do not have additional, unused bits of address space that we can press into duty as we split the bucket. We now need to use 3 bits of the hash address in order to divide up the records that hash to bucket *B*. The trie illustrated in Fig. 12.6(a) makes the distinctions required to complete the split. Figure 12.6(b) shows what this trie looks like once it is extended into a completely full binary tree with all leaves at the same level, and Fig. 12.6(c) shows the collapsed, directory form of the trie.

By building on the trie's ability to extend the amount of information used in a search, we have doubled the size of our address space (and, therefore, of our directory), extending it from 2^2 to 2^3 cells. This ability to grow (or shrink) the address space gracefully is what extendible hashing is all about.

We have been concentrating on the contribution that tries make to extendible hashing; one might well ask where the *hashing* comes into play. Why not just use the tries on the bits in the key, splitting buckets and extending the address space as necessary? The answer to this question grows out of hashing's most fundamental characteristic: a good hash function produces a nearly uniform distribution of keys across an address space. Notice that the trie shown in Fig. 12.6 is poorly balanced, resulting in a directory that is twice as big as it needs to be. If we had an uneven distribution of addresses that placed even more records in buckets *B* and *D* without using other parts of the address space, the situation would get even worse. By using a good hash function to create addresses with a nearly uniform distribution, we avoid this problem.

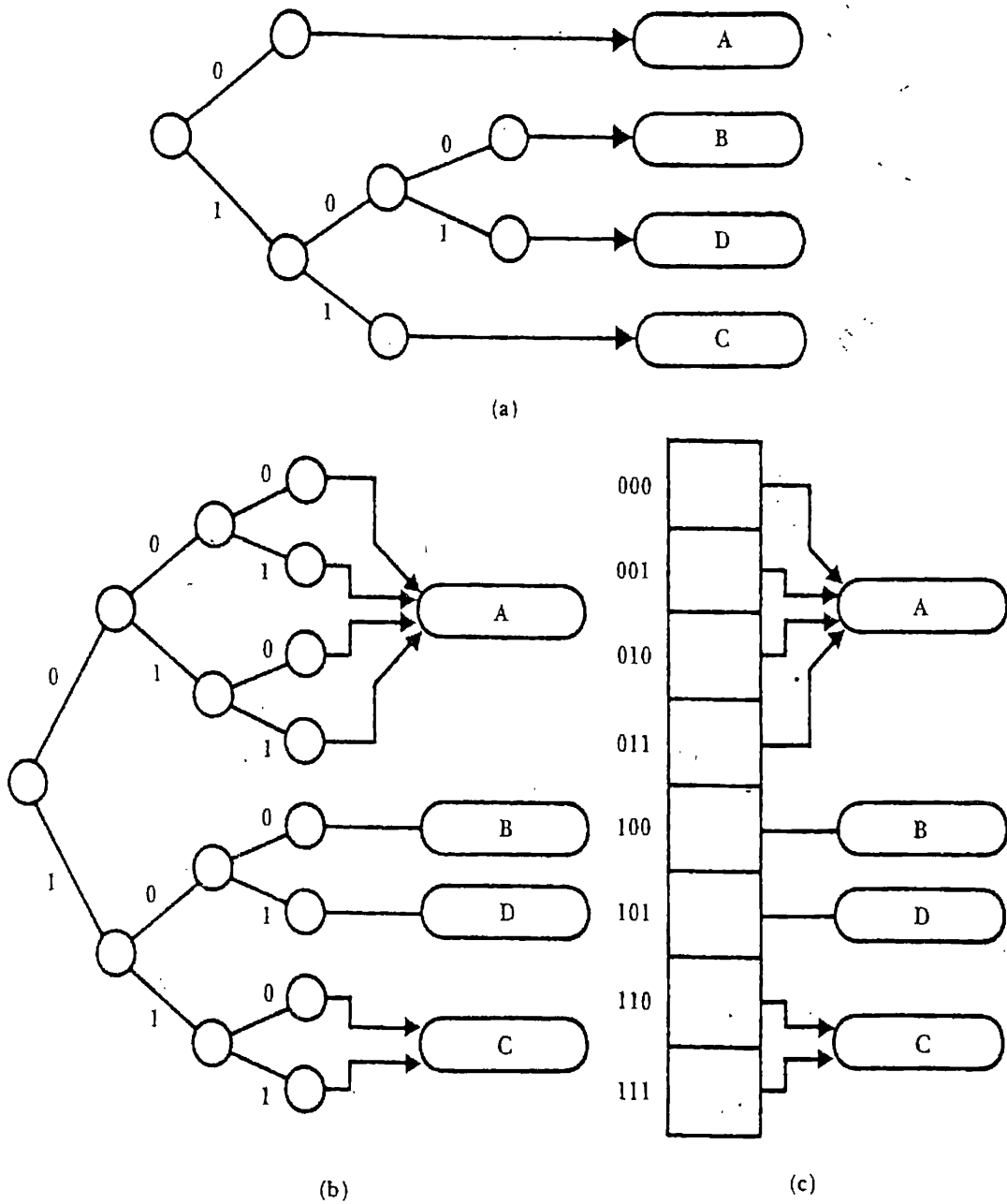


Figure 12.6 The results of an overflow of bucket B in Fig. 12.4(b), represented first as a trie, then as a complete binary tree, and finally as a directory.

12.3 Implementation

12.3.1 Creating the Addresses

Now that we have a high-level overview of how extendible hashing works, let's look at an object-oriented implementation. Appendix J contains the

```
int Hash (char* key)
{
    int sum = 0;
    int len = strlen(key);
    if (len % 2 == 1) len++; // make len even
    // for an odd length, use the trailing '\0' as part of key
    for (int j = 0; j < len; j+=2)
        sum = (sum + 100 * key[j] + key[j+1]) % 19937;
    return sum;
}
```

Figure 12.7 Function Hash (key) returns an integer hash value for key for a 15-bit

full class definitions and method bodies for extendible hashing. The place to start our discussion of the implementation is with the functions that create the addresses, since the notion of an extendible address underlies all other extendible hashing operations.

The function Hash given in Fig. 12.7, and file `hash.cpp` of Appendix J, is a simple variation on the fold-and-add hashing algorithm we used in Chapter 11. The only difference is that we do not conclude the operation by returning the remainder of the folded address divided by the address space. We don't need to do that, since in extendible hashing we don't have a fixed address space, instead we use as much of the address as we need. The division that we perform in this function, when we take the sum of the folded character values modulo 19 937, is to make sure that the character summation stays within the range of a signed 16-bit integer. For machines that use 32-bit integers, we could divide by a larger number and create an even larger initial address.

Because extendible hashing uses more bits of the hashed address as they are needed to distinguish between buckets, we need a function `MakeAddress` that extracts just a portion of the full hashed address. We also use `MakeAddress` to reverse the order of the bits in the hashed address, making the lowest-order bit of the hash address the highest-order bit of the value used in extendible hashing. To see why this reversal of bit order is desirable, look at Fig. 12.8, which is a set of keys and binary hash addresses produced by our hash function. Even a quick scan of these addresses reveals that the distribution of the least significant bits of these integer values tends to have more variation than the high-order bits. This is because many of the addresses do not make use of the upper reaches of our address space; the high-order bits often turn out to be 0.

bill	0000 0011 0110 1100
lee	0000 0100 0010 1000
pauline	0000 1111 0110 0101
alan	0100 1100 1010 0010
julie	0010 1110 0000 1001
mike	0000 0111 0100 1101
elizabeth	0010 1100 0110 1010
mark	0000 1010 0000 0111

Figure 12.8 Output from the hash function for a number of keys.

By reversing the bit order, working from right to left, we take advantage of the greater variability of low-order bit values. For example, given a 4-bit address space, we want to avoid having the addresses of *bill*, *lee*, and *pauline* turn out to be *0000*, *0000*, and *0000*. If we work from right to left, starting with the low-order bit in each address, we get *0011* for *bill*, *0001* for *lee*, and *1010* for *pauline*, which is a much more useful result.

Function `MakeAddress`, given in Fig. 12.9 and file `hash.cpp` of Appendix J, accomplishes this bit extraction and reversal. The `depth` argument tells the function the number of address bits to return.

```
int MakeAddress (char * key, int depth)
{
    int retval = 0;
    int hashVal = Hash(key);
    // reverse the bits
    for (int j = 0; j < depth; j++)
    {
        retval = retval << 1;
        int lowbit = hashVal & 1;
        retval = retval | lowbit;
        hashVal = hashVal >> 1;
    }
    return retval;
}
```

Figure 12.9 Function `MakeAddress` (`key`, `depth`) gets a hashed address, reverses the order of the bits, and returns an address of `depth` bits.

12.3.2 Classes for Representing Bucket and Directory Objects

Our extendible hashing scheme consists of a set of buckets stored in a file and a directory that references them. Each bucket is a record that contains a particular set of keys and information associated with the keys. A directory is primarily an array containing the record addresses of the buckets.

We have chosen to use a data record reference for the information associated with the keys in this implementation. That is, the set of buckets forms an index of the file of actual data records. This is certainly not the only way to configure the bucket file. We have left it as an exercise to extend the bucket class so that the associated information could be the rest of the data record. In this case, the bucket file is the data file. For the rest of the discussion of the implementation of extended hashing, we will treat the buckets as sets of key-reference pairs.

The basic operations on buckets are exactly the same as those of index records: add a key-reference pair to a bucket, search for a key and return its reference, and remove a key. Hence, we have chosen to make class `Bucket` a derived class of the class `TextIndex` from Chapter 5 and Appendix F. The definition of class `Bucket` is given in Fig. 12.10 and file `bucket.h` in Appendix J. These bucket records are stored in a file; we retrieve them as necessary. Each `Bucket` is connected to a directory and can be accessed only in that context. This access restriction is enforced by making the members of the class protected so that no outside access is allowed, then granting special access rights to class `Directory` by including the friend class `Directory` statement. Making class `Directory` a friend of `Bucket` allows methods of class `Directory` to access all of the private and protected members of class `Bucket`. The included methods of class `Bucket` will be explained below.

Class `Directory` is given in Fig. 12.11 and in file `direct.h` of Appendix J. Each cell in the directory consists of the file address of a `Bucket` record. Because we use *direct access* to find directory cells, we implement the directory as an array of these cells in memory. The address values returned by `MakeAddress` are treated as subscripts for this array, ranging from 0 to one less than the number of cells in the directory. Additional members are included to support the file operations required to store the directory and the related buckets. You may notice a striking similarity with classes `IndexedFile` (Chapter 5) and `BTree` (Chapter 9). Each of these classes supports open, create, and close operations as well as insert, search, and remove. Objects of class `BufferFile` are used to provide the I/O operations.

```

class Bucket: protected TextIndex
{protected:
    // there are no public members,
    // access to Bucket members is only through class Directory
    Bucket (Directory & dir, int maxKeys = defaultMaxKeys);
    int Insert (char * key, int recAddr);
    int Remove (char * key);
    Bucket * Split (); // split the bucket and redistribute the keys
    int NewRange (int & newStart, int & newEnd);
        // calculate the range of a new (split) bucket
    int Redistribute (Bucket & newBucket); // redistribute keys
    int FindBuddy (); // find the bucket that is the buddy of this
    int TryCombine (); // attempt to combine buckets
    int Combine (Bucket * buddy, int buddyIndex); //combine buckets
    int Depth; //number of bits used 'in common' by keys in bucket
    Directory & Dir; // directory that contains the bucket
    int BucketAddr; // address in file
    friend class Directory;
    friend class BucketBuffer;
};

```

Figure 12.10 Main members of class Bucket.

In order to use a `Directory` object, it must be constructed and then attached to a file for the directory and one for the buckets. Fig 12.12 (page 536) is a sample test program, `tsthash.cpp`. The two steps for initialization are the declaration of the `Directory` object, and the call to method `Create`, that creates the two files and the initial empty bucket. The program proceeds to insert a set of key-reference pairs. Notice that the reference values have no particular significance in this program.

The constructor and method `Create` are given in Fig. 12.13 (page 537). The constructor creates all of the objects that support the I/O operations: a buffer and a file for the directory and a buffer and a file for the buckets. The directory is stored in memory while the directory is open. The `Open` method reads the directory from the file and the `Close` writes it back into the file.

```

class Directory
{public:
    Directory (int maxBucketKeys = -1);
    ~Directory ();
    int Open (char * name);
    int Create (char * name);
    int Close ();
    int Insert (char * key, int recAddr);
    int Delete (char * key, int recAddr = -1);
    int Search (char * key); // return RecAddr for key
    ostream & Print (ostream & stream);
protected:
    int Depth; // depth of directory
    int NumCells; // number of cells, = 2**Depth
    int * BucketAddr; // array of bucket addresses

    // protected methods
    int DoubleSize (); // double the size of the directory
    int Collapse (); // collapse, halve the size
    int InsertBucket (int bucketAddr, int first, int last);
    int Find (char * key); // return BucketAddr for key
    int StoreBucket (Bucket * bucket);
        // update or append bucket in file
    int LoadBucket (Bucket * bucket, int bucketAddr);
        // load bucket from file
    // members to support directory and bucket files
    int MaxBucketKeys;
    BufferFile * DirectoryFile;
    LengthFieldBuffer * DirectoryBuffer;
    Bucket * CurrentBucket; // object to hold one bucket
    BucketBuffer * theBucketBuffer; // buffer for buckets
    BufferFile * BucketFile;
    int Pack () const;
    int Unpack ();
    Bucket * PrintBucket; // object to hold one bucket for printing
    friend class Bucket;
};

```

Figure 12.11 Definition of class Directory.

```

main ()
{
    int result;
    Directory Dir (4);
    result = Dir . Create ("hashfile");
    if (result == 0) {return 0;} // unable to create files
    char * keys[]={ "bill", "lee", "pauline", "alan", "julie",
        "mike", "elizabeth", "mark", "ann", "peter",
        "christina", "john", "charles", "mary", "emily"};
    const int numkeys = 15;
    for (int i = 0; i<numkeys; i++)
    {
        result = Dir . Insert (keys[i], 100 + i);
        if (result == 0)
            cout << "insert for "<<keys[i]<<" failed"<<endl;
        Dir . Print (cout);
    }
    return 1;
}

```

Figure 12.12 Test program `tsthash.cpp` inserts a sequence of key-reference pairs into a directory.

Note that member `Depth` is directly related to the size of the directory, since

$$2^{\text{Depth}} = \text{the number of cells in the directory.}$$

If we are starting a new hash directory, the directory depth is 0, which means that we are using *no* bits to distinguish between addresses; all the keys go into the same bucket, no matter what their address. We get the address of the initial, everything-goes-here bucket and assign it to the single directory cell in this line from `Directory::Create`:

```
BucketAddr[0] = StoreBucket (CurrentBucket);
```

The method `StoreBucket` appends the new bucket to the bucket file and returns its address.

12.3.3 Directory and Bucket Operations

Now that we have a way to open and close the file, we are ready to add records to the directory. The `Insert`, `Search`, and `Find` methods are

```

Directory::Directory (int maxBucketKeys)
{
    Depth = 0; // depth of directory
    NumCells = 1; // number of entries, = 2**Depth
    BucketAddr = new int [NumCells]; // array of bucket addresses
    // create I/O support objects
    MaxBucketKeys = maxBucketKeys;
    DirectoryBuffer = new LengthFieldBuffer; // default size
    DirectoryFile = new BufferFile(*DirectoryBuffer);
    CurrentBucket = new Bucket (*this, MaxBucketKeys);
    theBucketBuffer = new BucketBuffer (MaxKeySize, MaxBucketKeys);
    BucketFile = new BufferFile (*theBucketBuffer);
    PrintBucket = new Bucket (*this, MaxBucketKeys);
}

int Directory::Create (char * name)
{ // create the two files, create a single bucket
  // and add it to the directory and the bucket file
  int result;
  char * directoryName, * bucketName;
  makeNames(name, directoryName, bucketName); // create file names
  result = DirectoryFile->Create(directoryName, ios::in|ios::out);
  if (!result) return 0;
  result = BucketFile->Create(bucketName, ios::in|ios::out);
  if (!result) return 0;
  // store the empty bucket in the BucketFile; add to Directory
  BucketAddr[0] = StoreBucket (CurrentBucket);
  return result;
}

```

Figure 12.13 Constructor and method Create of class Directory.

shown in Fig. 12.14. The Insert method first searches for the key. Search arranges for the CurrentBucket member to contain the proper bucket for the key. If the key is not already in the bucket, then the Bucket::Insert method is called to perform the insertion. In method Directory::Search, as in most search functions we have seen, the Find method determines where the key would be if it were in the structure. In this case, Find determines which bucket is associated with the key. As noted previously, MakeAddress finds the array index of the directory cell that contains the file address of the appropriate bucket.

```

int Directory::Insert (char * key, int recAddr)
{
    int found = Search (key);
    if (found != -1) return 0; // key already in directory
    return CurrentBucket->Insert(key, recAddr);
}
int Directory::Search (char * key)
// return RecAddr for key, also put current bucket into variable
{
    int bucketAddr = Find(key);
    LoadBucket (CurrentBucket, bucketAddr);
    return CurrentBucket->Search(key);
}
int Directory::Find (char * key)
// find BucketAddr associated with key
{ return BucketAddr[MakeAddress (key, Depth)]; }

```

Figure 12.14 Methods Insert, Search, and Find of class Directory.

```

int Bucket::Insert (char * key, int recAddr)
{
    if (NumKeys < MaxKeys)
    {
        int result = TextIndex::Insert (key, recAddr);
        Dir.StoreBucket (this);
        return result;
    }
    else // bucket is full
    {
        Split ();
        return Dir.Insert (key, recAddr);
    }
}

```

Figure 12.15 Method Insert of class Bucket adds the key to the existing bucket if there is room. If the bucket is full, it splits it and then adds the key.

Method `Bucket::Insert`, given in Fig. 12.15 and in file `buffer.cpp` of Appendix J, is called with a key-reference pair. If the bucket is not full, `Insert` simply calls `TextIndex::Insert` to add

the key-reference pair to the bucket and stores the bucket in the file. A full bucket, however, requires a split, which is where things start to get interesting. After the split is done, the `Directory::Insert` is called (recursively) to try again to insert the key-reference pair.

What we do when we split a bucket depends on the relationship between the number of address bits used in the bucket and the number used in the directory as a whole. The two numbers are often not the same. To see this, look at Fig. 12.6(a). The directory uses 3 bits to define its address space (8 cells). The keys in bucket *A* are distinguished from keys in other buckets by having an initial 0 bit. All the other bits in the hashed key values in bucket *A* can be any value; it is only the first bit that matters. Bucket *A* is using only 1 bit and has depth 1.

The keys in bucket *C* all share a common first 2 bits; they all begin with *11*. The keys in buckets *B* and *D* use 3 bits to establish their identities and, therefore, their bucket locations. If you look at Fig. 12.6(c), you can see how using more or fewer address bits changes the relationship between the directory and the bucket. Buckets that do not use as many address bits as the directory have more than one directory cell pointing to them.

If we split one of the buckets that is using fewer address bits than the directory, and therefore is referenced from more than one directory cell, we can use half of the directory cells to point to the new bucket after the split. Suppose, for example, that we split bucket *A* in Fig. 12.6(c). Before the split only 1 bit, the initial 0, is used to identify keys that belong in bucket *A*. After the split, we use 2 bits. Keys starting with *00* (directory cells *000* and *001*) go in bucket *A*; keys starting with *01* (directory cells *010* and *011*) go in the new bucket. We do not have to expand the directory because the directory already has the capacity to keep track of the additional address information required for the split.

If, on the other hand, we split a bucket that has the same address depth as the directory, such as buckets *B* or *D* in Fig. 12.6(c), then there are no additional directory cells that we can use to reference the new bucket. Before we can split the bucket, we have to double the size of the directory, creating a new directory entry for every one that is currently there so we can accommodate the new address information.

Figure 12.16 gives an implementation of method `Split`. First we compare the number of bits used for the directory with the number used for the bucket to determine whether we need to double the directory. If the depths are the same, we double the directory before proceeding.

```

Bucket * Bucket::Split ()
{
    // split this into two buckets, store the new bucket, and
    // return (memory) address of new bucket
    int newStart, newEnd;
    if (Depth == Dir.Depth) // no room to split this bucket
        Dir.DoubleSize(); // increase depth of directory
    Bucket * newBucket = new Bucket (Dir, MaxKeys);
    Dir.StoreBucket (newBucket); // append to file
    NewRange (newStart, newEnd); // determine directory addresses
    Dir.InsertBucket(newBucket->BucketAddr, newStart, newEnd);
    Depth ++; // increment depth of this
    newBucket->Depth = Depth;
    Redistribute (*newBucket); // move some keys into new bucket
    Dir.StoreBucket (this);
    Dir.StoreBucket (newBucket);
    return newBucket;
}

```

Figure 12.16 Method `Split` of class `Bucket` divides keys between an existing bucket and a new bucket. If necessary, it doubles the size of the directory to accommodate the new bucket.

Next we create the new bucket that we need for the split. Then we find the range of directory addresses that we will use for the new bucket. For instance, when we split bucket *A* in Fig. 12.6(c), the range of directory addresses for the new bucket is from *010* to *011*. We attach the new bucket to the directory over this range, adjust the bucket address depth information in both buckets to reflect the use of an additional address bit, then redistribute the keys from the original bucket across the two buckets.

The most complicated operation supporting the `Split` method is `NewRange`, which finds the range of directory cells that should point to the new bucket instead of the old one after the split. It is given in Fig. 12.17. To see how it works, return, once again, to Fig. 12.6(c). Assume that we need to split bucket *A*, putting some of the keys into a new bucket *E*. Before the split, any address beginning with a *0* leads to *A*. In other words, the *shared address* of the keys in bucket *A* is *0*.

When we split bucket *A* we add another address bit to the path leading to the keys; addresses leading to bucket *A* now share an initial *00* while those leading to *E* share an *01*. So, the range of addresses for the new bucket is all directory addresses beginning with *01*. Since the directory address-

```

int Bucket::NewRange (int & newStart, int & newEnd)
{
    // make a range for the new split bucket
    int sharedAddr = MakeAddress(Keys[0], Depth);
    int bitsToFill = Dir.Depth - (Depth + 1);
    newStart = (sharedAddr << 1) | 1;
    newEnd = newStart;
    for (int j = 0; j < bitsToFill; j++)
    {
        newStart = newStart << 1;
        newEnd = (newEnd << 1) | 1;
    }
    return 1;
}

```

Figure 12.17 Method `NewRange` of class `Bucket` finds the start and end directory addresses for the new bucket by using information from the old bucket.

es use 3 bits, the new bucket is attached to the directory cells starting with *010* and ending with *011*.

Suppose that the directory used a 5-bit address instead of a 3-bit address. Then the range for the new bucket would start with *01000* and end with *01111*. This range covers all 5-bit addresses that share *01* as the first 2 bits. The logic for finding the range of directory addresses for the new bucket, then, starts by finding *shared address* bits for the new bucket. It then fills the address out with 0s until we have the number of bits used in the directory. This is the start of the range. Filling the address out with 1s produces the end of the range.

The directory operations required to support `Split` are easy to implement. They are given in Fig. 12.18. The first, `Directory::DoubleSize`, simply calculates the new directory size, allocates the required memory, and writes the information from each old directory cell into two successive cells in the new directory. It finishes by freeing the old space associated with member `BufferAddrs`, renaming the new space as the `BufferAddrs`, and increasing the `Depth` to reflect the fact that the directory is now using an additional address bit.

Method `InsertBucket`, used to attach a bucket address across a range of directory cells, is simply a loop that works through the cells to make the change.

```

int Directory::DoubleSize ()
// double the size of the directory
{
    int newSize = 2 * NumCells;
    int * newBucketAddr = new int[newSize];
    for (int i = 0; i < NumCells; i++)
        (// double the coverage of each bucket
         newBucketAddr[2*i] = BucketAddr[i];
         newBucketAddr[2*i+1] = BucketAddr[i];
        )
    delete BucketAddr; // delete old space for cells
    BucketAddr = newBucketAddr;
    Depth ++;
    NumCells = newSize;
    return 1;
}

int Directory::InsertBucket (int bucketAddr, int first, int last)
{
    for (int i = first; i <= last; i++)
        BucketAddr[i] = bucketAddr;
    return 1;
}

```

Figure 12.18 Methods DoubleSize and Directory InsertBucket of class Directory.

12.3.4 Implementation Summary

Now that we have assembled all of the pieces necessary to add records to an extendible hashing system, let's see how the pieces work together.

The `Insert` method manages record addition. If the key already exists, `Insert` returns immediately. If the key does not exist, `Insert` calls `Bucket::Insert`, for the bucket into which the key is to be added. If `Bucket::Insert` finds that there is still room in the bucket, it adds the key and the operation is complete. If the bucket is full, `Bucket::Insert` calls `Split` to handle the task of splitting the bucket.

The `Split` method starts by determining whether the directory is large enough to accommodate the new bucket. If the directory needs to be larger, `Split` calls method `Directory::DoubleSize` to double the directory size. `Split` then allocates a new bucket, attaches it to the appropriate directory cells, and divides the keys between the two buckets.

When `Bucket::Insert` regains control after `Split` has allocated a new bucket, it calls `Directory::Insert` to try to place the key into the new, revised directory structure. The `Directory::Insert` function, of course, calls `Bucket::Insert` again, recursively. This cycle continues until there is a bucket that can accommodate the new key. A problem can occur if there are many keys that have exactly the same hash address. The process of double, split, and insert will never make room for the new key.

12.4 Deletion

12.4.1 Overview of the Deletion Process

If extendible hashing is to be a truly *dynamic* system, like B-trees or AVL trees, it must be able to *shrink* files gracefully as well as grow them. When we delete a key, we need a way to see if we can decrease the size of the file system by combining buckets and, if possible, decreasing the size of the directory.

As with any dynamic system, the important question during deletion concerns the definition of the triggering condition: When do we combine buckets? This question, in turn, leads us to ask, Which buckets can be combined? For B-trees the answer involves determining whether nodes are *siblings*. In extendible hashing we use a similar concept: buckets that are *buddy* buckets.

Look again at the trie in Fig. 12.6(b). Which buckets could be combined? Trying to combine anything with bucket *A* would mean collapsing everything else in the trie first. Similarly, there is no single bucket that could be combined with bucket *C*. But buckets *B* and *D* are in the same configuration as buckets that have just split. They are ready to be combined: they are *buddy* buckets. We will take a closer look at finding *buddy* buckets when we consider implementation of the deletion procedure; for now let's assume that we combine buckets *B* and *D*.

After combining buckets, we examine the directory to see if we can make changes there. Looking at the directory form of the trie in Fig. 12.6(c), we see that once we combine buckets *B* and *D*, directory cells *100* and *101* both point to the same bucket. In fact, each of the buckets has at least a pair of directory cells pointing to it. In other words, none of the buckets requires the depth of address information that is currently available in the directory. That means that we can shrink the directory and reduce the address space to half its size.

Reducing the size of the address space restores the directory and bucket structure to the arrangement shown in Fig. 12.4, before the additions and splits that produced the structure in Fig. 12.6(c). Reduction consists of collapsing each adjacent pair of directory cells into a single cell. This is easy, because both cells in each pair point to the same bucket. Note that this is nothing more than a reversal of the directory splitting procedure that we use when we need to add new directory cells.

12.4.2 A Procedure for Finding Buddy Buckets

Given this overview of how deletion works, we begin by focusing on buddy buckets. Given a bucket, how do we find its buddy? Figure 12.19 contains the code for method `Bucket::FindBuddy`. The method works by checking to see whether it is possible for there to be a buddy bucket. Clearly, if the directory depth is 0, meaning that there is only a single bucket, there cannot be a buddy.

The next test compares the number of bits used by the bucket with the number of bits used in the directory address space. A pair of buddy buckets is a set of buckets that are immediate descendants of the same node in the trie. They are, in fact, pairwise siblings resulting from a split. Going back to Fig. 12.6(b), we see that asking whether the bucket uses all the address bits in the directory is another way of asking whether the bucket is at the lowest level of the trie. It is only when a bucket is at the outer edge of the trie that it can have a single parent and a single buddy.

Once we determine that there is a buddy bucket, we need to find its address. First we find the address used to find the bucket we have at hand;

```
int Bucket::FindBuddy ()
{
    // find the bucket that is paired with this
    if (Dir.Depth == 0) return -1; // no buddy, empty directory

    // unless bucket depth == directory depth, there is no single
    // bucket to pair with
    if (Depth < Dir.Depth) return -1;
    int sharedAddress = MakeAddress(Keys[0], Depth);
    // address of any key
    return sharedAddress ^ 1; // exclusive or with low bit
}
```

Figure 12.19 Method `FindBuddy` of class `Bucket` returns a buddy bucket or `-1` if none is found.

this is the shared address of the keys in the bucket. Since we know that the buddy bucket is the other bucket that was formed from a split, we know that the buddy has the same address in all regards except for the last bit. Once again, this relationship is illustrated by buckets *B* and *D* in Fig. 12.6(b). So, to get the buddy address, we flip the last bit with an exclusive or. We return directory address of the buddy bucket.

12.4.3 Collapsing the Directory

The other important support function used to implement deletion is the function that handles collapsing the directory. Downsizing the directory is one of the principal potential benefits of deleting records. In our implementation we use one function to see whether downsizing is possible and, if it is, to collapse the directory.

Method `Directory::Collapse`, given in Fig. 12.20, begins by making sure that we are not at the lower limit of directory size. By treating the special case of a directory with a single cell here, at the start of the function, we simplify subsequent processing: with the exception of this case, all directory sizes are evenly divisible by 2.

The test to see if the directory can be collapsed consists of examining each pair of directory cells to see if they point to different buckets. As soon

```
int Directory::Collapse ()
{
    // if collapse is possible, reduce size by half
    if (Depth == 0) return 0; // only 1 bucket
    // look for buddies that are different, if found return
    for (int i = 0; i < NumCells; i += 2)
        if (BucketAddr[i] != BucketAddr[i+1]) return 0;
    int newSize = NumCells / 2;
    int * newAddrs = new int [newSize];
    for (int j = 0; j < newSize; j++)
        newAddrs[j] = BucketAddr[j*2];
    delete BucketAddr;
    BucketAddr = newAddrs;
    Depth --;
    NumCells = newSize;
    return 1;
}
```

Figure 12.20 Method `Collapse` of class `Directory` reduces the size of the directory, if possible.

as we find such a pair, we know that we *cannot* collapse the directory and the method returns. If we get all the way through the directory without encountering such a pair, then we can collapse the directory.

The collapsing operation consists of allocating space for a new array of bucket addresses that is half the size of the original and then copying the bucket references shared by each cell pair to a single cell in the new directory.

12.4.4 Implementing the Deletion Operations

Now that we have an approach to the two critical support operations for deletion, finding buddy buckets and collapsing the directory, we are ready to construct the higher levels of the deletion operation.

The highest-level deletion operation, `Directory::Remove`, is very simple. We first try to find the key to be deleted. If we cannot find it, we return failure; if we find it, we call `Bucket::Remove` to remove the key from the bucket. We return the value reported back from that method. Figure 12.21 gives the implementation of these two methods.

Method `Bucket::Remove` does its work in two steps. The first step, removing the key from the bucket, is accomplished through the call to `TextIndex::Remove`, the base class `Remove` method. The second

```
int Directory::Remove (char * key)
{
    // remove the key and return its RecAddr
    int bucketAddr = Find(key);
    LoadBucket (CurrentBucket, bucketAddr);
    return CurrentBucket -> Remove (key);
}

int Bucket::Remove (char * key)
{
    // remove the key, return its RecAddr
    int result = TextIndex::Remove (key);
    if (!result) return 0; // key not in bucket
    TryCombine (); // attempt to combine with buddy
    // make the changes permanent
    Dir.StoreBucket(this);
    return 1;
}
```

Figure 12.21 Remove methods of classes `Directory` and `Bucket`.

step, which takes place only if a key is removed, consists of calling TryCombine to see if deleting the key has decreased the size of the bucket enough to allow us to combine it with its buddy.

Figure 12.22 shows the implementation of TryCombine and Combine. Note that when we combine buckets, we reduce the address depth associated with the bucket: combining buckets means that we use 1 less address bit to differentiate keys.

```

int Bucket::TryCombine ()
{
    // called after insert to combine buddies, if possible
    int result;
    int buddyIndex = FindBuddy ();
    if (buddyIndex == -1) return 0; // no combination possible
    // load buddy bucket into memory
    int buddyAddr = Dir.BucketAddr[buddyIndex];
    Bucket * buddyBucket = new Bucket (Dir, MaxKeys);
    Dir . LoadBucket (buddyBucket, buddyAddr);
    // if the sum of the sizes of the buckets is too big, return
    if (NumKeys + buddyBucket->NumKeys > MaxKeys) return 0;
    Combine (buddyBucket, buddyIndex);
    result = Dir.Collapse (); // collapse the 2 buckets
    if (result) TryCombine(); //if collapse, may be able to combine
    return 1;
}

int Bucket::Combine (Bucket * buddy, int buddyIndex)
{
    // combine this and buddy to make a single bucket
    int result;
    // move keys from buddy to this
    for (int i = 0; i < buddy->NumKeys; i++)
    {
        // insert the key of the buddy into this
        result = Insert (buddy->Keys[i], buddy->RecAddrs[i]);
        if (!result) return 0; // this should not happen
    }
    Depth --; // reduce the depth of the bucket
    Dir . RemoveBucket (buddyIndex, Depth); // delete buddy bucket
    return 1;
}

```

Figure 12.22 Methods TryCombine and Combine of class Bucket. TryCombine tests to see whether a bucket can be combined with its buddy. If the test succeeds, TryCombine calls Combine to do the combination.

After combining the buckets, we call `Directory::Collapse()` to see if the decrease in the number of buckets enables us to decrease the size of the directory. If we do, in fact, collapse the directory, `TryCombine` calls itself recursively. Collapsing the directory may have created a new buddy for the bucket; it may be possible to do even more combination and collapsing. Typically, this recursive combining and collapsing happens only when the directory has a number of empty buckets that are awaiting changes in the directory structure that finally produce a buddy to combine with.

12.4.5 Summary of the Deletion Operation

Deletion begins with a call to `Directory::Remove` that passes the key that is to be deleted. If the key cannot be found, there is nothing to delete. If the key is found, the bucket containing the key is passed to `Bucket::Remove`.

The `Bucket::Remove` method deletes the key, then passes the bucket on to `Directory::TryCombine` to see if the smaller size of the bucket will now permit combination with a buddy bucket. `TryCombine` first checks to see if there is a buddy bucket. If not, we are done. If there is a buddy, and if the sum of the keys in the bucket and its buddy is less than or equal to the size of a single bucket, we combine the buckets.

The elimination of a bucket through combination might cause the directory to collapse to half its size. We investigate this possibility by calling `Directory::Collapse`. If collapsing succeeds, we may have a new buddy bucket, so `TryCombine` calls itself again, recursively.

File `testdel.cpp` in Appendix J opens the directory created by `testhash.cpp` and proceeds to delete each element of the directory. Using a debugger to step through this program may help in understanding the deletion process.

12.5 Extendible Hashing Performance

Extendible hashing is an elegant solution to the problem of extending and contracting the address space for a hash file as the file grows and shrinks. How well does it work? As always, the answer to this question must consider the trade-off between time and space.

The time dimension is easy to handle: if the directory for extendible hashing can be kept in memory, a single access is all that is ever required to retrieve a record. If the directory is so large that it must be paged in and out of memory, two accesses may be necessary. The important point is that extendible hashing provides $O(1)$ performance: since there is no overflow, these access time values are truly independent of the size of the file.

Questions about space utilization for extendible hashing are more complicated than questions about access time. We need to be concerned about two uses of space: the space for the buckets and the space for the directory.

12.5.1 Space Utilization for Buckets

In their original paper describing extendible hashing, Fagin, Nievergelt, Pippenger, and Strong include analysis and simulation of extendible hashing performance. Both the analysis and simulation show that the space utilization is strongly periodic, fluctuating between values of 0.53 and 0.94. The analysis portion of their paper suggests that for a given number of records r and a block size of b , the average number of blocks N is approximated by the formula

$$N \approx \frac{r}{b \ln 2} N$$

Space utilization, or packing density, is defined as the ratio of the actual number of records to the total number of records that could be stored in the allocated space:

$$\text{Utilization} = \frac{r}{bN}$$

Substituting the approximation for N gives us:

$$\text{Utilization} \approx \ln 2 = 0.69$$

So, we expect *average* utilization of 69 percent. In Chapter 9, where we looked at space utilization for B-trees, we found that simple B-trees tend to have a utilization of about 67 percent, but this can be increased to more than 85 percent by redistributing keys during insertion rather than just splitting when a page is full. So, B-trees tend to use less space than simple extendible hashing, typically at a cost of requiring a few extra seeks.

The average space utilization for extendible hashing is only part of the story; the other part relates to the periodic nature of the variations in

space utilization. It turns out that if we have keys with randomly distributed addresses, the buckets in the extendible hashing table tend to fill up at about the same time and therefore tend to split at the same time. This explains the large fluctuations in space utilization. As the buckets fill up, space utilization can reach past 90 percent. This is followed by a concentrated series of splits that reduce the utilization to below 50 percent. As these now nearly half-full buckets fill up again, the cycle repeats itself.

12.5.2 Space Utilization for the Directory

The directory used in extendible hashing grows by doubling its size. A prudent designer setting out to implement an extendible hashing system will want assurance that this doubling levels off for reasonable bucket sizes, even when the number of keys is quite large. Just how large a directory should we expect to have, given an expected number of keys?

Flajolet (1983) addressed this question in a lengthy, carefully developed paper that produces a number of different ways to estimate the directory size. Table 12.1, which is taken from Flajolet's paper, shows the expected value for the directory size for different numbers of keys and different bucket sizes.

Flajolet also provides the following formula for making rough estimates of the directory size for values that are not in this table. He notes that this formula tends to overestimate directory size by a factor of 2 to 4.

$$\text{Estimated directory size} = \frac{3.92}{b} r^{(1+1/b)}$$

Table 12.1 Expected directory size for a given bucket size b and total number of records r .

b	5	10	20	50	100	200
r						
10^3	1.50 K	0.30 K	0.10 K	0.00 K	0.00 K	0.00 K
10^4	25.60 K	4.80 K	1.70 K	0.50 K	0.20 K	0.00 K
10^5	424.10 K	68.20 K	16.80 K	4.10 K	2.00 K	1.00 K
10^6	6.90 M	1.02 M	0.26 M	62.50 K	16.80 K	8.10 K
10^7	112.11 M	12.64 M	2.25 M	0.52 M	0.26 M	0.13 M

1 K = 10^3 , 1 M = 10^6 .

From Flajolet, 1983.

12.6 Alternative Approaches

12.6.1 Dynamic Hashing

In 1978, before Fagin, Nievergelt, Pippenger, and Strong produced their paper on extendible hashing, Larson published a paper describing a scheme called *dynamic hashing*. Functionally, dynamic hashing and extendible hashing are very similar. Both use a directory to track the addresses of the buckets, and both extend the directory through the use of tries.

The key difference between the approaches is that dynamic hashing, like conventional, static hashing, starts with a hash function that covers an address space of a fixed size. As buckets within that fixed address space overflow, they split, forming the leaves of a trie that grows down from the original address node. Eventually, after enough additions and splitting, the buckets are addressed through a forest of tries that have been seeded out of the original static address space.

Let's look at an example. Figure 12.23(a) shows an initial address space of four and four buckets descending from the four addresses in the directory. In Fig. 12.23(b) we have split the bucket at address 4. We address the two buckets resulting from the split as 40 and 41. We change the shape of the directory node at address 4 from a square to a circle because it has changed from an external node, referencing a bucket, to an internal node that points to two child nodes.

In Fig. 12.23(c) we split the bucket addressed by node 2, creating the new external nodes 20 and 21. We also split the bucket addressed by 41, extending the trie downward to include 410 and 411. Because the directory node 41 is now an internal node rather than an external one, it changes from a square to a circle. As we continue to add keys and split buckets, these directory tries continue to grow.

Finding a key in a dynamic hashing scheme can involve the use of two hash functions rather than just one. First, there is the hash function that covers the original address space. If you find that the directory node is an external node and therefore points to a bucket, the search is complete. However, if the directory node is an internal node, then you need additional address information to guide you through the 1s and 0s that form the trie. Larson suggests using a second hash function on the key and using the result of this hashing as the seed for a random-number generator that produces a sequence of 1s and 0s for the key. This sequence describes the path through the trie.

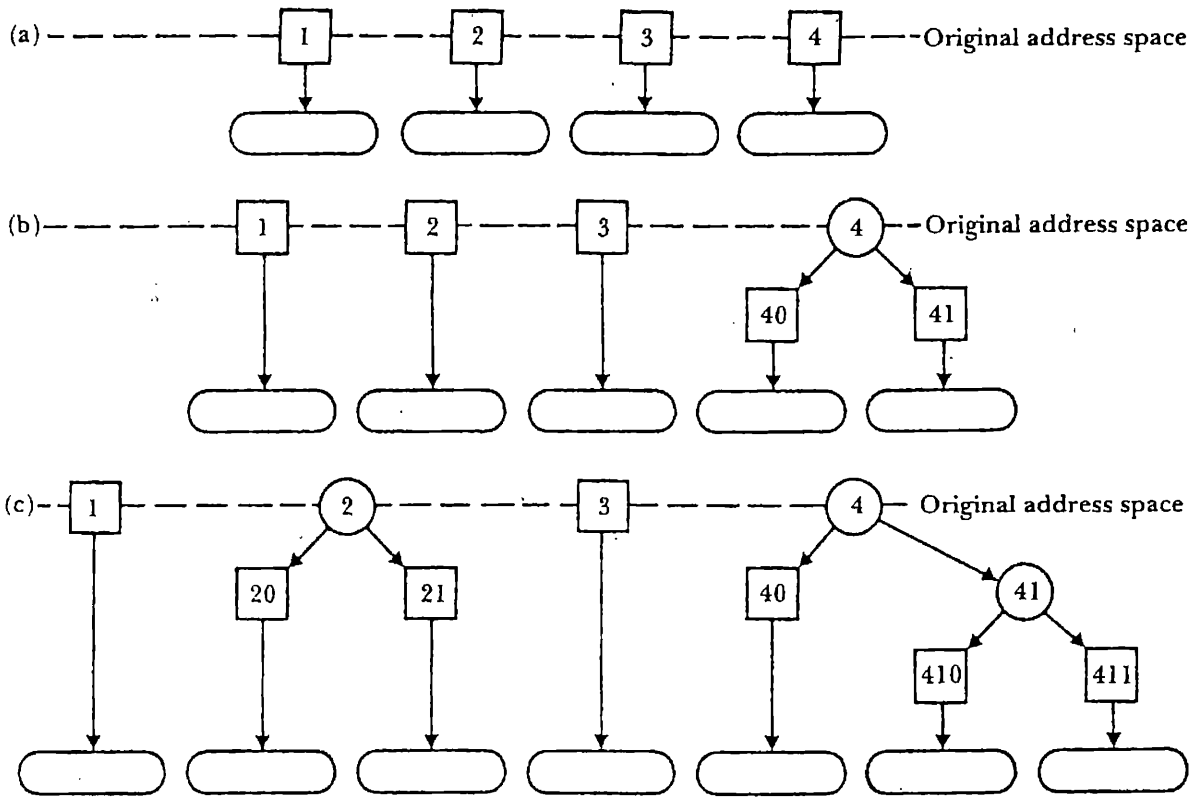


Figure 12.23 The growth of index in dynamic hashing.

It is interesting to compare dynamic hashing and extendible hashing. A brief, but illuminating, characterization of similarities and differences is that while both schemes extend the hash function locally, as a binary search trie, in order to handle overflow, dynamic hashing expresses the extended directory as a linked structure while extendible hashing expresses it as a perfect tree, which is in turn expressible as an array.

Because of this fundamental similarity, it is not surprising that the space utilization within the buckets is the same (69 percent) for both approaches. Moreover, since the directories are essentially equivalent, just expressed differently, it follows that the estimates of directory depth developed by Flajolet (1983) apply equally well to dynamic hashing and extendible hashing. (In section 12.5.2 we talk about estimates for the directory size for extendible hashing, but we know that in extendible hashing $directory\ depth = \log_2\ directory\ size$.)

The primary difference between the two approaches is that dynamic hashing allows for slower, more gradual growth of the directory, whereas extendible hashing extends the directory by doubling it. However, because the directory nodes in dynamic hashing must be capable of holding point-

ers to children, the size of a node in dynamic hashing is larger than a directory cell in extendible hashing, probably by at least a factor of 2. So, the directory for dynamic hashing will usually require more space in memory. Moreover, if the directory becomes so large that it requires use of virtual memory, extendible hashing offers the advantage of being able to access the directory with no more than a single page fault. Since dynamic hashing uses a linked structure for the directory, it may be necessary to incur more than one page fault to move through the directory.

12.6.2 Linear Hashing

The key feature of both extendible hashing and dynamic hashing is that they use a directory to access the buckets containing the key records. This directory makes it possible to expand and modify the hashed address space without expanding the number of buckets: after expanding the directory, more than one directory node can point to the same bucket. However, the directory adds an additional layer of indirection which, if the directory must be stored on disk, can result in an additional seek.

Linear hashing, introduced by Litwin in 1980, does away with the directory. An example, developed in Fig. 12.24, shows how linear hashing works. This example is adapted from a description of linear hashing by Enbody and Du (1988).

Linear hashing, like extendible hashing, uses more bits of hashed value as the address space grows. The example begins (Fig. 12.24[a]) with an address space of four, which means that we are using an address function that produces addresses with two bits of depth. In terms of the operations that we developed earlier in this chapter, we are calling `MakeAddress` with a key and a second argument of 2. For this example we will refer to this as the $h_2(k)$ address function. Note that the address space consists of four *buckets* rather than four directory nodes that can point to buckets.

As we add records, bucket *b* overflows. The overflow forces a split. However, as Fig. 12.24(b) shows, it is not bucket *b* that splits, but bucket *a*. The reason for this is that we are extending the address space *linearly*, and bucket *a* is the next bucket that must split to create the next linear extension, which we call bucket *A*. A 3-bit hash function, $h_3(k)$, is applied to buckets *a* and *A* to divide the records between them. Since bucket *b* was not the bucket that we split, the overflowing record is placed into an overflow bucket *w*.

We add more records, and bucket *d* overflows. Bucket *b* is the next one to split and extend the address space, so we use the $h_3(k)$ address function

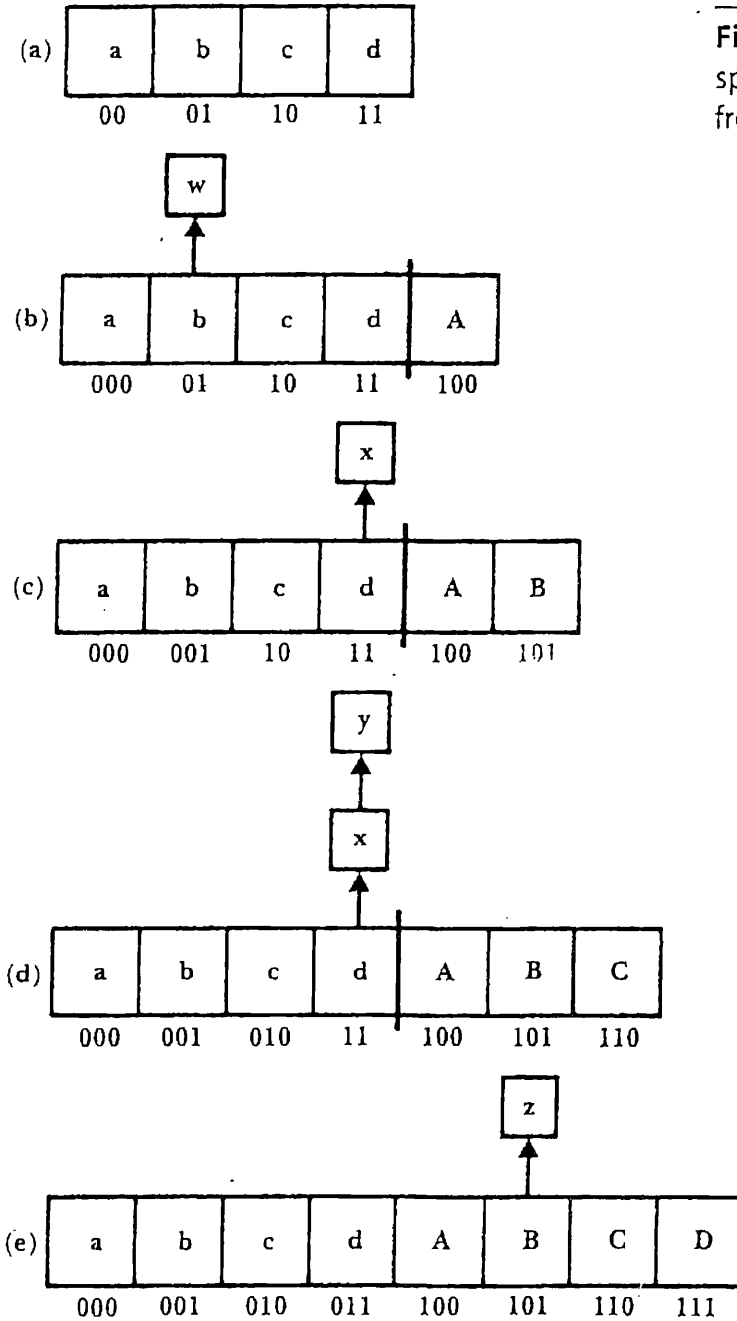


Figure 12.24 The growth of address space in linear hashing. Adapted from Enbody and Du (1988).

to divide the records from bucket b and its overflow bucket w between b and the new bucket B . The record overflowing bucket d is placed in an overflow bucket x . The resulting arrangement is illustrated in Fig. 12.24(c).

Figure 12.24(d) shows what happens when, as we add more records, bucket d overflows beyond the capacity of the overflow bucket w . Bucket c is the next in the extension sequence, so we use the $h_3(k)$ address function to divide the records between c and C .

Finally, assume that bucket B overflows. The overflow record is placed in the overflow bucket z . The overflow also triggers the extension to bucket D , dividing the contents of d , x , and y between buckets d and D . At this point all of the buckets use the $h_3(k)$ address function, and we have finished the expansion cycle. The pointer for the next bucket to be split returns to bucket a to get ready for a new cycle that will use an $h_4(k)$ address function to reach new buckets.

Because linear hashing uses two hash functions to reach the buckets during an expansion cycle, an $h_d(k)$ function for the buckets at the current address depth and an $h_{d+1}(k)$ function for the expansion buckets, finding a record requires knowing which function to use. If p is the pointer to the address of the next bucket to be split and extended, then the procedure for finding the address of the bucket containing a key k is as follows:

```
if ( $h_d(k) \leq p$ )
    address =  $h_d(k)$ ;
else
    address =  $h_{d+1}(k)$ ;
```

Litwin (1980) shows that the access time performance of linear hashing is quite good. There is no directory to access or maintain, and since we extend the address space through splitting every time there is overflow, the overflow chains do not become very large. Given a bucket size of 50, the average number of disk accesses per search approaches very close to one. Space utilization, on the other hand, is lower than it is for extendible hashing or dynamic hashing, averaging around only 60 percent.

12.6.3 Approaches to Controlling Splitting

We know from Chapter 9 that we can increase the storage capacity of B-trees by implementing measures that tend to postpone splitting, redistributing keys between pages rather than splitting pages. We can apply similar logic to the hashing schemes introduced in this chapter, placing records in chains of overflow buckets to postpone splitting.

Since linear hashing has the lowest storage utilization of the schemes introduced here, and since it already includes logic to handle overflow buckets, it is an attractive candidate for use of controlled splitting logic. In its uncontrolled-splitting form, linear hashing splits a bucket and extends the address space every time any bucket overflows. This choice of a triggering event for splitting is arbitrary, particularly when we consider that the bucket that splits is typically not the bucket that overflows. Litwin

(1980) suggests using the overall load factor of the file as an alternative triggering event. Suppose we let the buckets overflow until the space utilization reaches some desired figure, such as 75 percent. Every time the utilization exceeds that figure, we split a bucket and extend the address space. Litwin simulated this kind of system and found that for load factors of 75 percent and even 85 percent, the average number of accesses for successful and unsuccessful searches still stays below 2.

We can also use overflow buckets to defer splitting and increase space utilization for dynamic hashing and extendible hashing. For these methods, which use directories to the buckets, deferring splitting has the additional attraction of keeping the directory size down. For extendible hashing it is particularly advantageous to chain to an overflow bucket and therefore avoid a split when the split would cause the directory to double in size. Consider the example that we used early in this chapter, where we split the bucket *B* in Fig. 12.4(b), producing the expanded directory and bucket structure shown in Fig. 12.6(c). If we had allowed bucket *B* to overflow instead, we could have retained the smaller directory. Depending on how much space we allocated for the overflow buckets, we might also have improved space utilization among the buckets. The cost of these improvements, of course, is a potentially greater search length due to the overflow chains.

Studies of the effects of different overflow bucket sizes and chaining mechanisms supported a small industry of academic research during the early and mid-1980s. Larson (1978) suggested the use of deferred splitting in his original paper on dynamic hashing but found the results of some preliminary simulations of the idea to be disappointing. Scholl (1981) developed a refinement of this idea in which overflow buckets are shared. Master's thesis research by Chang (1985) tested Scholl's suggestions empirically and found that it was possible to achieve storage utilization of about 81 percent while maintaining search performance in the range of 1.1 seeks per search. Veklerov (1985) suggested using buddy buckets for overflow rather than allocating chains of new buckets. This is an attractive suggestion, since splitting buckets without buddies can never cause a doubling of the directory in extendible hashing. Veklerov obtained storage utilization of about 76 percent with a bucket size of 8.

S U M M A R Y

Conventional, static hashing does not adapt well to file structures that are *dynamic*, that grow and shrink over time. Extendible hashing is one of several hashing systems that allow the address space for hashing to grow and shrink along with the file. Because the size of the address space can grow as the file grows, it is possible for extendible hashing to provide hashed access without the need for overflow handling, even as files grow many times beyond their original expected size.

The key to extendible hashing is using more bits of the hashed value as we need to cover more address space. The model for extending the use of the hashed value is the *trie*: every time we use another bit of the hashed value, we have added another level to the depth of a trie with a radix of 2.

In extendible hashing we fill out all the leaves of the trie until we have a perfect tree, then we collapse that tree into a one-dimensional array. The array forms a directory to the buckets, kept on disk, that hold the keys and records. The directory is managed in memory, if possible.

If we add a record and there is no room for it in a bucket, we split the bucket. We use 1 additional bit from the hash values for the keys in the bucket to divide the keys between the old bucket and the new one. If the address space represented in the directory can cover the use of this new bit, no more changes are necessary. If, however, the address space is using fewer bits than are needed by our splitting buckets, then we double the address space to accommodate the use of the new bit.

Deletion reverses the addition process, recognizing that it is possible to combine the records for two buckets only if they are *buddy* buckets, which is to say that they are the pair of buckets that resulted from a split.

Access performance for extendible hashing is a single seek if the directory can be kept in memory. If the directory must be paged off to disk, worst-case performance is two seeks. Space utilization for the buckets is approximately 69 percent. Tables and an approximation formula developed by Flajolet (1983) permit estimation of the probable directory size, given a bucket size and total number of records.

There are a number of other approaches to the problem solved by extendible hashing. *Dynamic hashing* uses a very similar approach but expresses the directory as a linked structure rather than as an array. The linked structure is more cumbersome but grows more smoothly. Space utilization and seek performance for dynamic hashing are the same as for extendible hashing.

Linear hashing does away with the directory entirely, extending the address space by adding new buckets in a linear sequence. Although the overflow of a bucket can be used to trigger extension of the address space in linear hashing, typically the bucket that overflows is not the one that is split and extended. Consequently, linear hashing implies maintaining overflow chains and a consequent degradation in seek performance. The degradation is slight, since the chains typically do not grow to be very long before they are pulled into a new bucket. Space utilization is about 60 percent.

Space utilization for extendible, dynamic, and linear hashing can be improved by postponing the splitting of buckets. This is easy to implement for linear hashing, since there are already overflow buckets. Using deferred splitting, it is possible to increase space utilization for any of the hashing schemes described here to 80 percent or better while still maintaining search performance averaging less than two seeks. Overflow handling for these approaches can use the sharing of overflow buckets.

KEY TERMS

Buddy bucket. Given a bucket with an address $uvwxy$, where u , v , w , x , and y have values of either 0 or 1, the buddy bucket, if it exists, has the value $uvwxz$, such that

$$z = y \text{ XOR } 1$$

Buddy buckets are important in deletion operations for extendible hashing because, if enough keys are deleted, the contents of buddy buckets can be combined into a single bucket.

Deferred splitting. It is possible to improve space utilization for *dynamic hashing*, *extendible hashing*, and *linear hashing* by postponing, or deferring, the splitting of buckets, placing records into overflow buckets instead. This is a classic space/time trade-off in which we accept diminished performance in return for more compact storage.

Directory. Conventional, static hashing schemes transform a key into a bucket address. Both *extendible hashing* and *dynamic hashing* introduce an additional layer of indirection, in which the key is hashed to a *directory address*. The directory, in turn, contains information about the location of the bucket. This additional indirection makes it possible to extend the address space by extending the directory rather than having to work with an address space made up of buckets.

Dynamic hashing. Used in a generic sense, *dynamic hashing* can refer to any hashing system that provides for expansion and contraction of the address space for dynamic files where the number of records changes over time. In this chapter we use the term in a more specific sense to refer to a system initially described by Larson (1978). The system uses a directory to provide access to the buckets that contain the records. Cells in the directory can be used as root nodes of *trie* structures that accommodate greater numbers of buckets as buckets split.

Extendible hashing. Like *dynamic hashing*, *extendible hashing* is sometimes used to refer to any hashing scheme that allows the address space to grow and shrink so it can be used in dynamic file systems. Used more precisely, as it is used in this chapter, *extendible hashing* refers to an approach to hashed retrieval for dynamic files that was first proposed by Fagin, Nievergelt, Pippenger, and Strong (1979). Their proposal is for a system that uses a directory to represent the address space. Access to buckets containing the records is through the directory. The directory is handled as an array; the size of the array can be doubled or halved as the number of buckets changes.

Linear hashing. An approach to hashing for dynamic files that was first proposed by Litwin (1980). Unlike *extendible hashing* and *dynamic hashing*, linear hashing does not use a directory. Instead, the address space is extended one bucket at a time as buckets overflow. Because the extension of the address space does not necessarily correspond to the bucket that is overflowing, linear hashing necessarily involves the use of overflow buckets, even as the address space expands.

Splitting. The hashing schemes described in this chapter make room for new records by splitting buckets to form new buckets, then extending the address space to cover these buckets. Conventional, static hashing schemes rely strictly on overflow buckets without extending the address space.

Trie. A search tree structure in which each successive character of the key is used to determine the direction of the search at each successive level of the tree. The branching factor (the *radix* of the trie) at any level is potentially equal to the number of values that the character can take.

FURTHER READINGS

For information about hashing for dynamic files that goes beyond what we present here, you must turn to journal articles. The best summary of the different approaches is Enbody and Du's *Computing Surveys* article titled "Dynamic Hashing Schemes," which appeared in 1988.

The original paper on extendible hashing is "Extendible Hashing—A Fast Access Method for Dynamic Files" by Fagin, Nievergelt, Pippenger, and Strong (1979). Larson (1978) introduces dynamic hashing in an article titled "Dynamic Hashing." Litwin's initial paper on linear hashing is titled "Linear Hashing: A New Tool for File and Table Addressing" (1980). All three of these introductory articles are quite readable; Larson's paper and Fagin, Nievergelt, Pippenger, and Strong are especially recommended.

Michel Scholl's 1981 paper titled "New File Organizations Based on Dynamic Hashing" provides another readable introduction to dynamic hashing. It also investigates implementations that defer splitting by allowing buckets to overflow.

Papers analyzing the performance of dynamic or extendible hashing often derive results that apply to either of the two methods. Flajolet (1983) presents a careful analysis of directory depth and size. Mendelson (1982) arrives at similar results and goes on to discuss the costs of retrieval and deletion as different design parameters are changed. Veklerov (1985) analyzes the performance of dynamic hashing when splitting is deferred by allowing records to overflow into a buddy bucket. His results can be applied to extendible hashing as well.

After introducing dynamic hashing, Larson wrote a number of papers building on the ideas associated with linear hashing. His 1980 paper titled "Linear Hashing with Partial Expansions" introduces an approach to linear hashing that can avoid the uneven distribution of the lengths of overflow chains across the cells in the address space. He followed up with a performance analysis in a 1982 paper titled "Performance Analysis of Linear Hashing with Partial Expansions." A subsequent, 1985 paper titled "Linear Hashing with Overflow—Handling by Linear Probing" introduces a method of handling overflow that does not involve chaining.

EXERCISES

1. Briefly describe the differences between extendible hashing, dynamic hashing, and linear hashing. What are the strengths and weaknesses of each approach?
2. The tries that are the basis for the extendible hashing procedure described in this chapter have a radix of 2. How does performance change if we use a larger radix?
3. In the `MakeAddress` function, what would happen if we did not reverse the order of the bits but just extracted the required number of low-order bits in the same left-to-right order that they occur in the address? Think about the way the directory location would change as we extend the implicit trie structure to use yet another bit.
4. If the language that you are using to implement the `MakeAddress` function does not support bit shifting and masking operations, how could you achieve the same ends, even if less elegantly and clearly?
5. In the method `Bucket::Split`, we redistribute keys between the original bucket and a new one. How do you decide whether a key belongs in the new bucket or the original bucket?
6. Suppose the redistribution of keys in `Bucket::Split` does not result in moving any keys into the new bucket. Under what conditions could such an event happen? How do the methods of classes `Bucket` and `Directory` handle this?
7. The `Bucket::TryCombine` function is potentially recursive. In section 12.4.4 we described a situation in which there are empty buckets that can be combined with other buckets through a series of recursive calls to `TryCombine`. Describe two situations that could produce empty buckets in the hash structure.
8. Deletion occasionally results in collapsing the directory. Describe the conditions that must be met before the directory can collapse. What methods in classes `Bucket` and `Directory` detect these conditions?
9. Deletion depends on finding buddy buckets. Why does the address depth for a bucket have to be the same as the address depth for the directory in order for a bucket to have a buddy?

10. In the extendible hashing procedure described in this chapter, the directory can occasionally point to empty buckets. Describe two situations that can produce empty buckets. How could we modify the methods to avoid empty buckets?
11. If buckets are large, a bucket containing only a few records is not much less wasteful than an empty bucket. How could we minimize *nearly empty* buckets?
12. Linear hashing makes use of overflow records. Assuming an uncontrolled splitting implementation in which we split and extend the address space as soon as we have an overflow, what is the effect of using different bucket sizes for the overflow buckets? For example, consider overflow buckets that are as large as the original buckets. Now consider overflow buckets that can hold only one record. How does this choice affect performance in terms of space utilization and access time?
13. In section 12.6.3 we described an approach to linear hashing that controls splitting. For a load factor of 85 percent, the average number of accesses for a successful search is 1.20 (Litwin, 1980). Unsuccessful searches require an average of 1.78 accesses. Why is the average search length greater for unsuccessful searches?
14. Because linear hashing splits one bucket at a time, in order, until it has reached the end of the sequence, the overflow chains for the last buckets in the sequence can become much longer than those for the earlier buckets. Read about Larson's approach to solving this problem through the use of "partial expansions," originally described in Larson (1980) and subsequently summarized in Enbody and Du (1988). Write a pseudocode description of linear hashing with partial expansions, paying particular attention to how addressing is handled.
15. In section 12.6.3 we discussed different mechanisms for deferring the splitting of buckets in extendible hashing in order to increase storage utilization. What is the effect of using smaller overflow buckets rather than larger ones? How does using smaller overflow buckets compare with sharing overflow buckets?

PROGRAMMING EXERCISES

16. Write a version of the `MakeAddress` function that prints out the input key, the hash value, and the extracted, reversed address. Build a driver that allows you to enter keys interactively for this function and see the results. Study the operation of the function on different keys.
18. Implement method `Directory::Delete`. Write a driver program to verify that your implementation is correct. Experiment with the program to see how deletion works. Try deleting all the keys. Try to create situations in which the directory will recursively collapse over more than one level.
19. Design and implement a class `HashedFile` patterned after class `TextIndexedFile` of Chapter 7 and Appendix G. A `HashedFile` object is a data file and an extendible hash directory. The class should have methods `Create`, `Open`, `Close`, `Read` (read record that matches key), `Append`, and `Update`.

PROGRAMMING PROJECT

This is the last part of the programming project. We create a hashed index of the student record files and the course registration files from the programming project of Chapter 4. This project depends on the successful completion of exercise 19.

20. Use class `HashedFile` to create a hashed index of a student record file with student identifier as key. Note that the student identifier field is not unique in a student registration file. Write a driver program to create a hashed file from an existing student record file.
21. Use class `HashedFile` to create a hashed index of a course registration record file with student identifier as key. Write a driver program to create a hashed file from an existing course registration record file.
22. Write a program that opens a hashed student file and a hashed course registration file and retrieves information on demand. Prompt a user for a student identifier and print all objects that match it.

