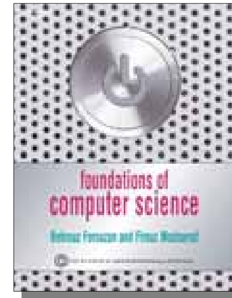
13
File Structures



Source: Foundations of Computer Science © Cengage Learning

Objectives

After studying this chapter, the student should be able to:

- ❑ Define two categories of access methods: sequential access and random access.
- ❑ Understand the structure of sequential files and how they are updated.
- ❑ Understand the structure of indexed files and the relation between the index and the data file.
- ❑ Understand the idea behind hashed files and describe some hashing methods.
- ❑ Describe address collisions and how they can be resolved.
- ❑ Define directories and how they can be used to organize files.
- ❑ Distinguish between text and binary files.

13-1 ACCESS METHODS

When we design a file, the important issue is how we will retrieve information (a specific record) from the file. Sometimes we need to process records one after another, whereas sometimes we need to access a specific record quickly without retrieving the preceding records. The access method determines how records can be retrieved: **sequentially** or **randomly**.

Sequential access

If we need to access a file sequentially—that is, one record after another, from beginning to end —we use a **sequential file structure**.

Random access

If we need to access a specific record without having to retrieve all records before it, we use a file structure that allows random access. Two file structures allow this: **indexed files** and **hashed files**. This taxonomy of file structures is shown in Figure 13.1.

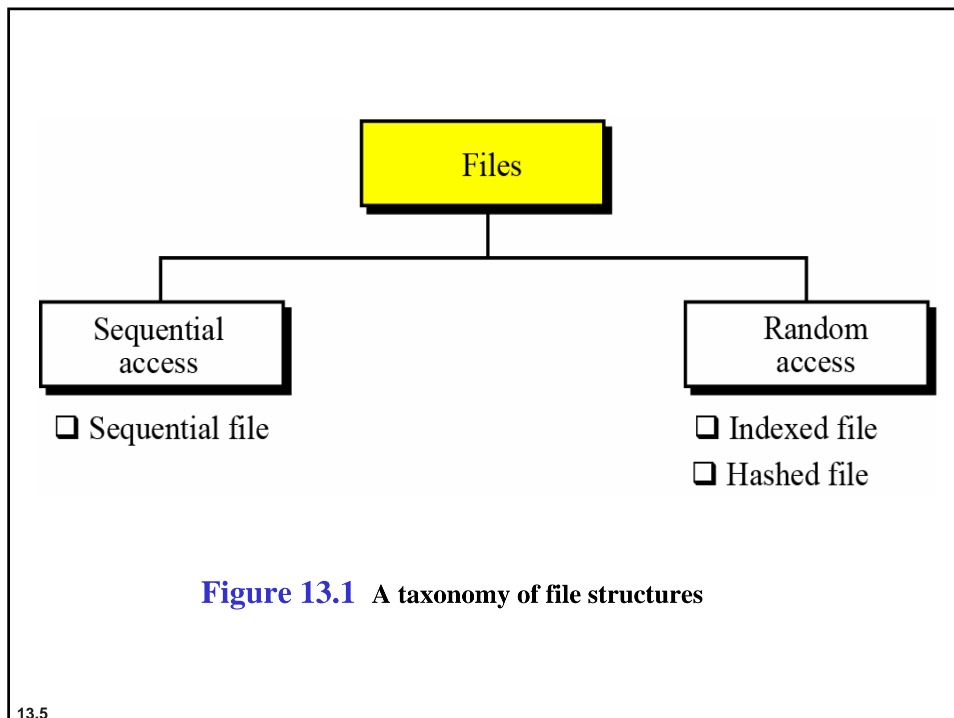


Figure 13.1 **A taxonomy of file structures**

13-2 SEQUENTIAL FILES

A sequential file is one in which records can only be accessed one after another from beginning to end. Figure 13.2 shows the layout of a sequential file. Records are stored one after another in auxiliary storage, such as tape or disk, and there is an EOF (end-of-file) marker after the last record. The operating system has no information about the record addresses, it only knows where the whole file is stored. The only thing known to the operating system is that the records are sequential.

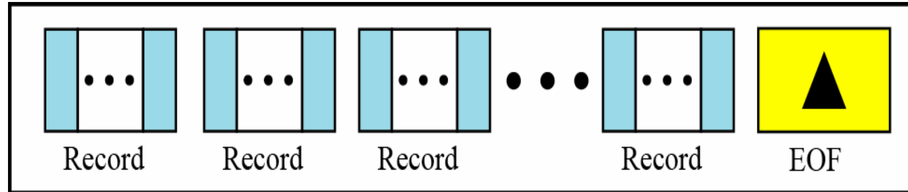


Figure 13.2 A sequential file

Algorithm 13.1 shows how records in a sequential file are processed.

Algorithm 13.1 Pseudocode for processing records in a sequential file

```
Algorithm: SequentialFileProcessing (file)
Purpose: Process all records in a sequential file
Pre: Given the beginning address of the file on the auxiliary storage
Post: None
Return: None
{
    while (Not EOF)
    {
        Read the next record from the auxiliary storage into memory
        Process the record
    }
}
```

Updating sequential files

Sequential files must be updated periodically to reflect changes in information. The updating process is very involved because all the records need to be checked and updated (if necessary) sequentially.

Files involved in updating

There are four files associated with an update program: the **new master file**, the **old master file**, the **transaction file** and the **error report file**. All these files are sorted based on key values. Figure 13.3 is a pictorial representation of a sequential file update.

13.9

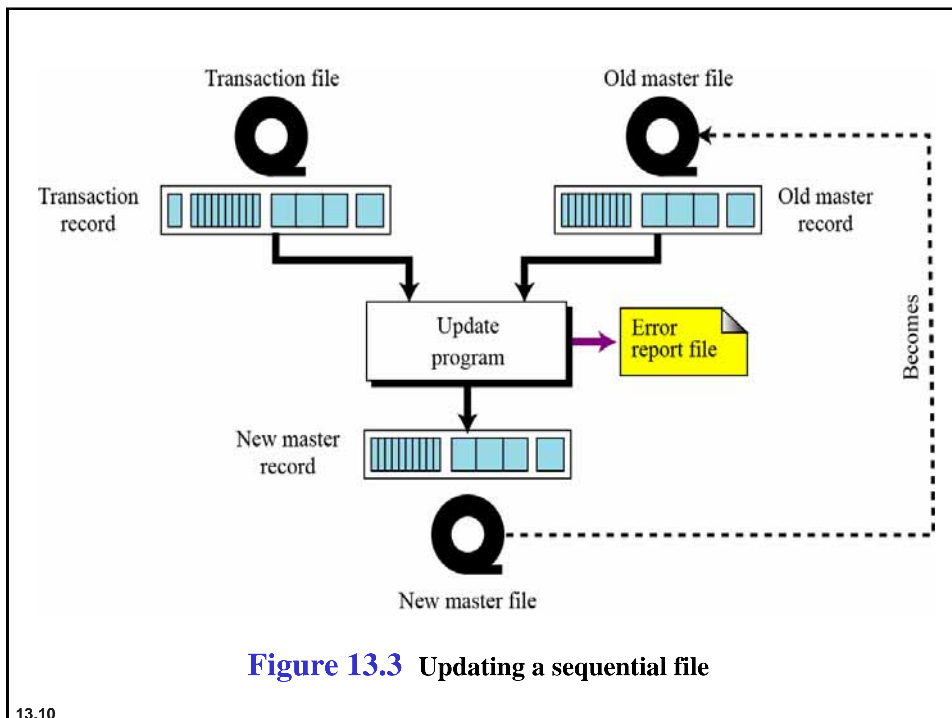


Figure 13.3 **Updating a sequential file**

13.10

Processing file updates

To make the updating process efficient, all files are sorted on the same key. This updating process is shown in Figure 13.4.

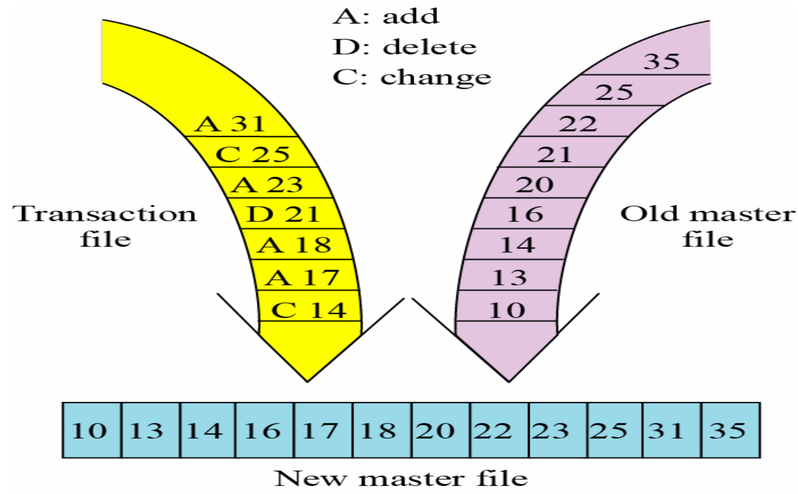


Figure 13.4 Updating process

13-2 INDEXED FILES

To access a record in a file randomly, we need to know the address of the record.

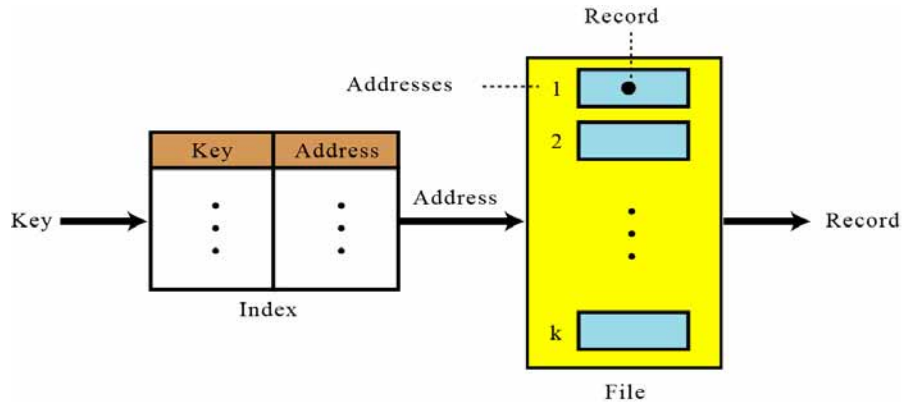


Figure 13.5 Mapping in an indexed file

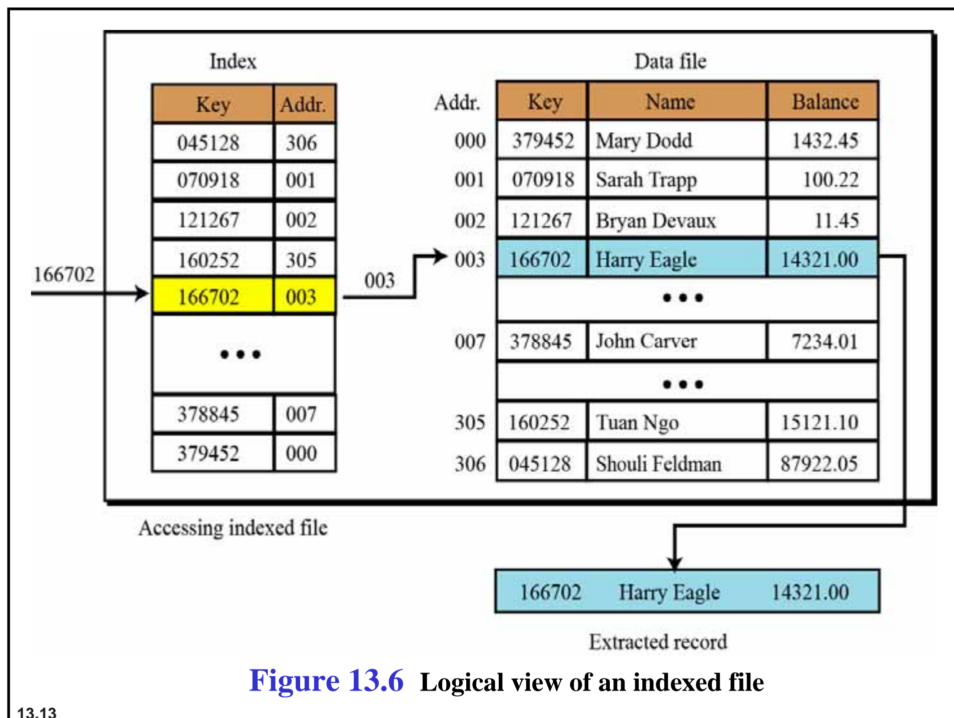


Figure 13.6 **Logical view of an indexed file**

Inverted files

One of the advantages of indexed files is that we can have **more than one index, each with a different key**. For example, an employee file can be retrieved based on either social security number or last name. This type of indexed file is usually called an **inverted file**.

12-2 HASHED FILES

A hashed file uses a mathematical function to accomplish this mapping. The user gives the key, the function maps the key to the address and passes it to the operating system, and the record is retrieved (Figure 13.7).

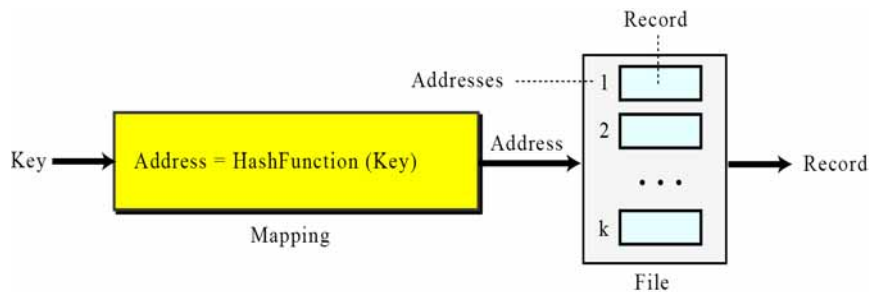


Figure 13.7 **Mapping in a hashed file**

Hashing methods

For key-address mapping, we can select one of several hashing methods. We discuss a few of them here.

Direct hashing

In direct hashing, the key is the data file address without any algorithmic manipulation. The file must therefore contain a record for every possible key. Although situations suitable for direct hashing are limited, it can be very powerful, because it guarantees that there are no synonyms or collisions (discussed later in this chapter), as with other methods.

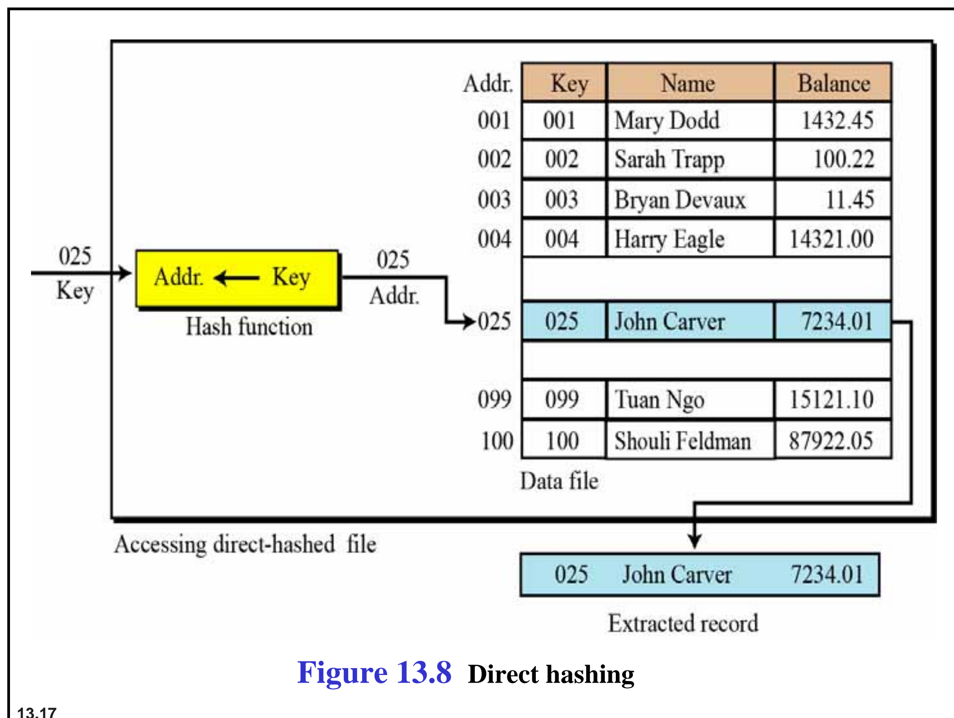


Figure 13.8 **Direct hashing**

13.17

Modulo division hashing

Also known as division remainder hashing, the modulo division method divides the key by the file size and uses the remainder plus 1 for the address. This gives the simple hashing algorithm that follows, where list_size is the number of elements in the file. The reason for adding a 1 to the mod operation result is that our list starts with 1 instead of 0.

$$\text{address} = \text{key} \ \mathbf{mod} \ \mathit{list_size} + 1$$

13.18

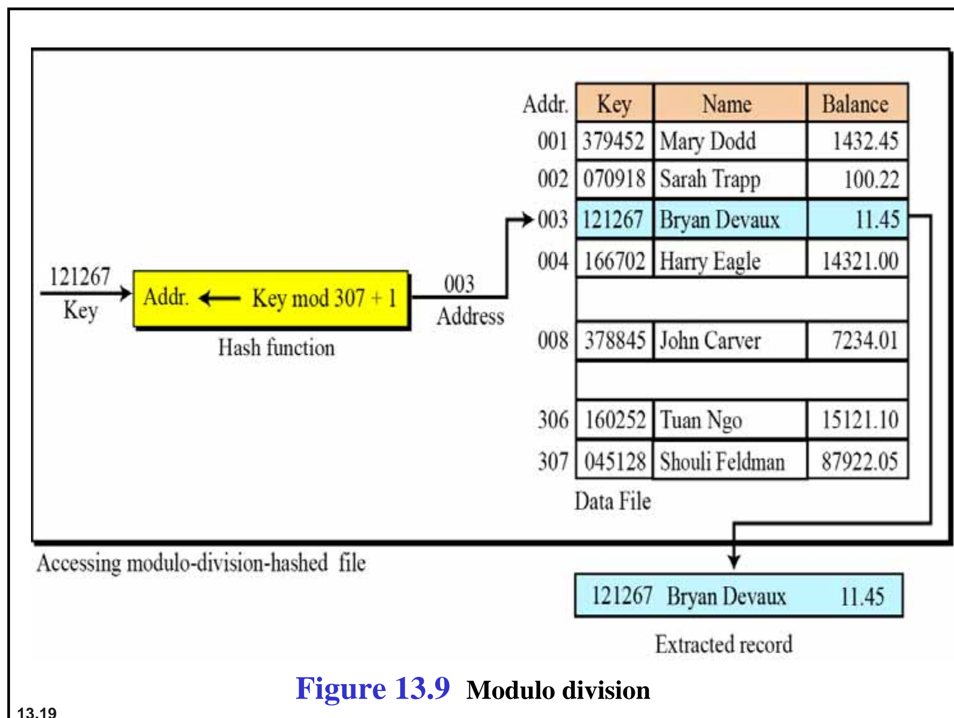


Figure 13.9 **Modulo division**

13.19

Digit extraction hashing

Using **digit extraction hashing**, selected digits are extracted from the key and used as the address. For example, using our six-digit employee number to hash to a three-digit address (000–999), we could **select the first, third and fourth digits** (from the left) and use them as the address. Using the keys from Figure 13.9, we hash them to the following addresses:

125870 → 158 122801 → 128 121267 → 112

Other hashing methods

Other popular methods exist, but we leave the exploration of these as exercises.

13.20

Collision

Generally, the population of keys for a hashed list is greater than the number of records in the data file.

For example, if we have a file of 50 students for a class in which the students are identified by the last four digits of their social security number, then there are 200 possible keys for each element in the file (10,000/50). Because there are many keys for each address in the file, there is a possibility that more than one key will hash to the same address in the file. We call the set of keys that hash to the same address in our list synonyms. The collision concept is illustrated in Figure 13.10.

13.21

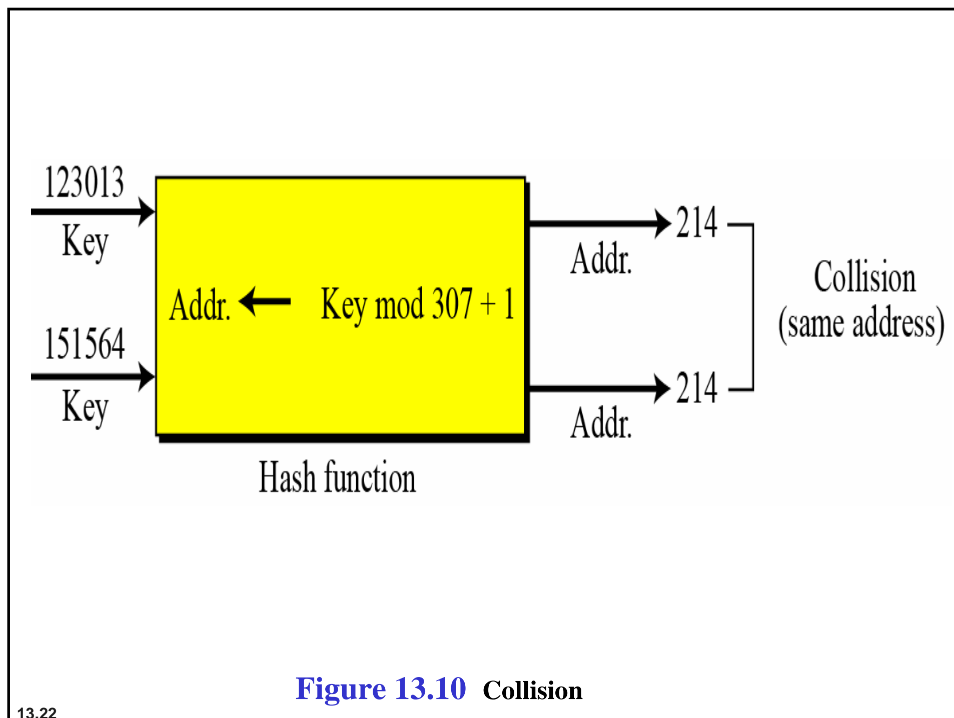


Figure 13.10 Collision

13.22

Collision resolution

With the exception of the direct method, none of the methods we have discussed for hashing creates one-to-one mappings. This means that when we hash a new key to an address, we may create a collision. There are several methods for handling collisions, each of them independent of the hashing algorithm. That is, any hashing method can be used with any collision resolution method. In this section, we discuss some of these methods.

13.23

Open addressing resolution

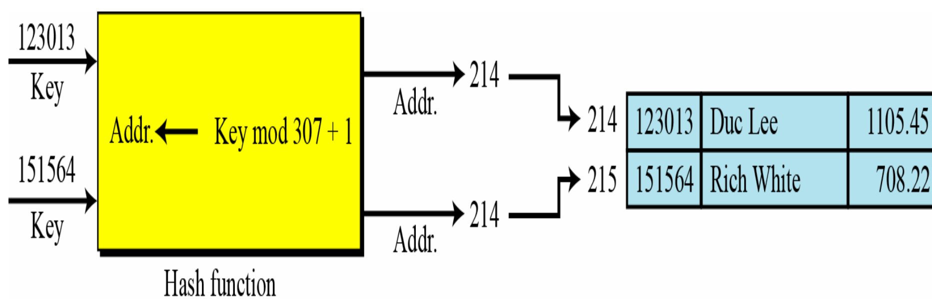


Figure 13.11 Open addressing resolution

13.24

Linked list resolution

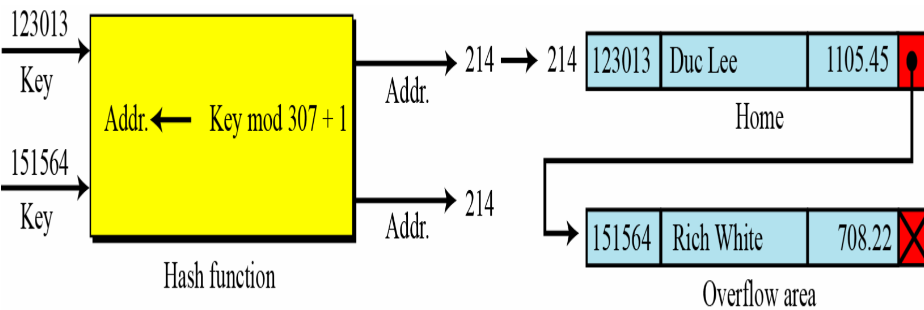


Figure 13.12 **Linked list resolution**

13.25

Bucket hashing resolution

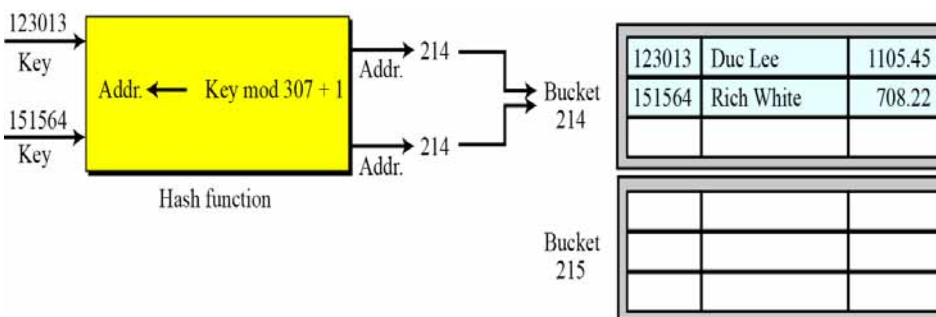


Figure 13.13 **Bucket hashing resolution**

13.26

13-5 DIRECTORIES

Directories are provided by most operating systems for organizing files. A **directory** performs the same function as a folder in a filing cabinet. However, a directory in most operating systems is represented as a special type of file that holds information about other files. A directory not only serves as a kind of index that tells the operating system where files are located on an auxiliary storage device, but can also contain other information about the files it contains, such as who has the access to each file, or the date when each file was created, accessed or modified.

13.27

Directories in the UNIX operating system

In UNIX, the directory system is organized as shown in Figure 13.14.

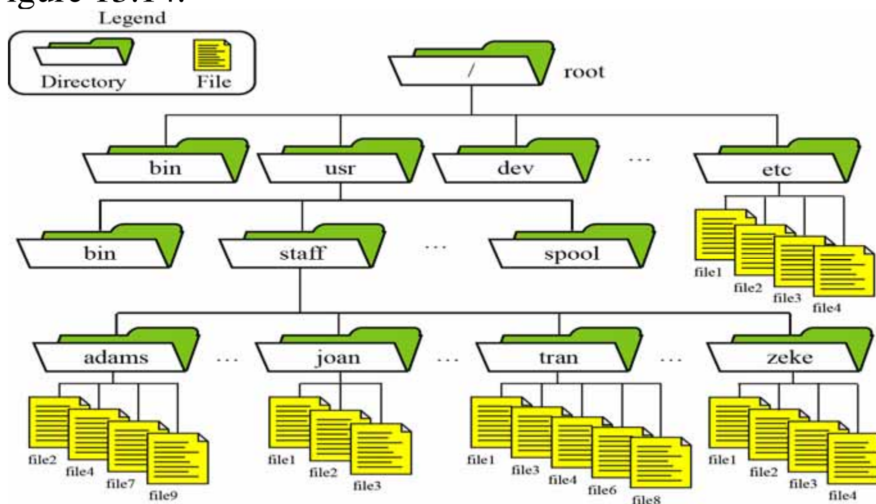


Figure 13.14 An example of the directory system in UNIX

13.28

Special directories

There are four special types of directory that play an important role in the directory structure in UNIX: the **root directory**, **home directories**, **working directories** and **parent directories**.

Paths and pathnames

The file's path is specified by its **absolute pathname**, a list of all directories separated by a slash character (/). UNIX also provides a shorter pathname, known as a relative pathname, which is the path relative to the working directory.

Relative pathname:	joan/file3
Absolute pathname:	/usr/staff/joan/file3

13.29

13-6 TEXT VERSUS BINARY

Two terms used to categorize files: **text files** and **binary files**. A file stored on a storage device is a sequence of bits that can be interpreted by an application program as a text file or a binary file, as shown in Figure 13.15.

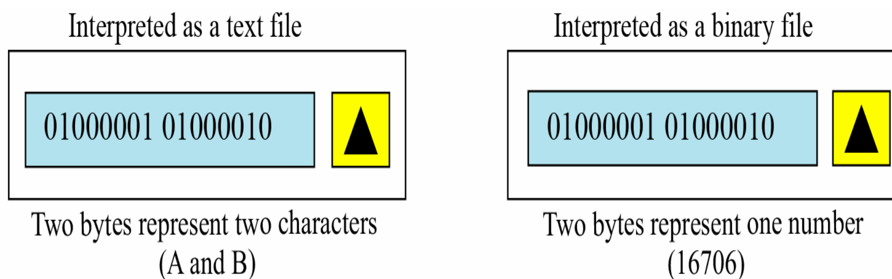


Figure 13.15 **Text and binary interpretations of a file**

13.30

Text files

A text file is a file of characters. It cannot contain integers, floating-point numbers, or any other data structures in their internal memory format. To store these data types, they must be converted to their character equivalent formats. Some files can only use character data types. Most notable are file streams (input/output objects in some object-oriented language like C++) for keyboards, monitors and printers. This is why we need special functions to format data that is input from or output to these devices.

13.31

Binary files

A binary file is a collection of data stored in the internal format of the computer. In this definition, data can be an integer (including other data types represented as unsigned integers, such as image, audio, or video), a floating-point number or any other structured data (except a file).

Unlike text files, binary files contain data that is meaningful only if it is properly interpreted by a program. If the data is textual, one byte is used to represent one character (in ASCII encoding). But if the data is numeric, two or more bytes are considered a data item.

13.32