Copyrighted Materials
Copyright © 2012 Oxford University Press Retrieved from www.knovel.com

14 FILES

OBJECTIVES

After completing this chapter, the reader will be able to understand the following:

- The purpose of standard data organization methods
- Various file organizations such as sequential, indexed sequential, and direct access, and their application-specific suitability
- The advantages and disadvantages of file organizations

14.1 INTRODUCTION

The prime role of computers is problem solving and data processing. In any computer application, the basic entity is data. Data can be either simple or it may have multiple attributes. One needs to select the appropriate data structure based on the nature of the application and data.

Data can have one or more attributes (fields). For example, an entity *Number_of_Students* can only be of the integer data type, whereas an entity *Student* may have multiple attributes or fields such as *Roll_No*, *Name*, *DOB*, *City*, and *Sex* to describe it. Each field of *Student* can be of a different data type. In such situations, we need a structure that will accommodate an aggregation of dissimilar data types that represents one occurrence of such a complex entity. This object is called a *record.*

Records that hold information about similar items of data are usually grouped together into a *file*. A file is a collection of records where each record consists of one or more fields.

For example, a file *Student* can have one or more records with fields such as *Roll_No*, *Name*, *DOB*, *City*, and *Sex*. Table 14.1 indicates the fields and their associated data type for this record.

Table 14.1 Student record

Field	Roll_No	Name	DOB	City	Sex
Datatype	Integer	Array of characters	Array of characters	Array of characters	Character

We have studied the representation of and operations on various data structures such as arrays, stacks, queues, linked lists, trees, and graphs. The storage representations and data manipulations described are applied only to data entities, which reside in the main memory. In many situations, all information that is to be processed does not reside in the main memory. There are two reasons for this. First, there are some large programs and data, which cannot fit conveniently into the main memory. Secondly, it is often desirable or necessary to store information from one run of a program to the next run. Let us consider a student's information system. We need the data to be preserved even after the execution of the program is over. Therefore, large volumes of data and archival data are commonly stored in external memory as special data holding entities—*files*.

Each record contains attributes to describe one entity. Generally, all records for one entity type are usually of the same form. Mostly, each of them has the same fields in the same quantity, order, and length. Such records are known as *fixed length records.* Structures in C/C++ support this type of record.

Records that are not necessarily of the same length are known as *variable length records.* C/C++ unions support this type of record. Variable length records are less common than fixed length records, as they are more difficult to handle. They tend to complicate the storage schemes and are sometimes impractical for some structures. When variable records are used, we need to maintain more information about each record.

Magnetic tapes, floppy disks, and hard disks are a few examples of secondary storage devices. When data is organized in a file data structure, the data is *non-volatile*, which means that the data will reside on storage after data processing is over.

14.2 EXTERNAL STORAGE DEVICES

For persistent storage, large volumes of data and archival data are commonly stored in external memory as special data holding entities, namely files. Before we learn about file organization and operations on files, let us discuss the storage devices, which hold the files.

The external storage devices are those on which information or data can be stored and from which it can be retrieved. The data resides on these devices as a non-volatile memory. The storage and retrieval operations are known as writing and reading, respectively. Capacity of external storage devices is larger than that of the main memory and is also slower and less expensive per bit of information stored when compared to the main memory.

External storage devices are mainly used for the following:

1. Overlay or backup of programs during execution
2. Storage of programs for future use
3. Storage of information in files

We shall mainly concentrate on the third use, discuss the most common external storage devices in the order of their uses, and study magnetic tapes, drums, and disk drives.

14.2.1 Magnetic Tape

A tape is made up of a plastic material coated with a ferrite substance that is easily magnetized. The physical appearance of the tape is similar to the tape used for sound recording. Computer tapes are wider with several thousand feet of tape wound on one reel, where information is encoded on the tape, character by character. A number of channels or tracks run along the length of the tape, one channel being required for each bit position in the binary coded representation of a character. Information is read or written on the tape through the use of a magnetic tape drive.

A limitation of magnetic tape devices is that records must be processed in the order in which they reside on the tape. Therefore, accessing a record requires the scanning of all records that precede it. This form of access is called *sequential access*. The magnetic tape is probably the cheapest form of external bulk storage. A reel of tape can be easily placed on and removed from a tape drive, and hence it can be used for off-line storage and data.

14.2.2 Magnetic Drum

A magnetic drum is a metal cylinder, from 10 to 36 inches in diameter, which has an outside surface coated with a magnetic recording material. The cylindrical surface of the drum is divided into a number of parallel bands called tracks. The tracks are further divided into either sectors or blocks, depending on the nature of the drum. The sector or block is the smallest addressable unit. A particular sector or block is directly addressable, that is, to access a sector or block of a drum, it is not necessary to access sectors or blocks 1 to n 1, as in the case of a sequential tape. Hence, a drum is called as a *direct access* storage device.

The addressable units (sectors or blocks) on drums are rapidly accessed for data transfers, and no scanning of extraneous data is required as with a magnetic tape. Also, unlike a magnetic tape, a drum cannot be removed from its shaft or drive. Hence, the maximum storage capacity for a drum device is limited to the capacity of a single drum.

14.2.3 Magnetic Disk

The magnetic disk is a direct access storage device, which has become more widely used than the magnetic drum, mainly because of its lower cost. Disk devices provide relatively low access times and high-speed data transfer. There are two types of disk devices, namely, fixed disks and exchangeable disks. For both types, the disk unit or pack consists of a number of metal platters, which are stacked on top of each other on a spindle. The upper and lower surfaces of each platter are coated with ferromagnetic particles that provide an information storage media.

The surfaces of each platter are divided into concentric bands called tracks. Each track is further divided into sectors (or blocks) that are addressable units. There are read/write heads floating just above or below the surface of the disk while the disk is rotating. An exchangeable disk device has movable read/write heads. The heads are attached to a

movable arm to form a comb-like access assembly. When data on a particular track must be accessed, the whole assembly moves to position the read/write heads over the derived track. Although many heads may be in position for a read/write transaction at a given point in time, data transmission can only take place through one head at a time.

14.3 FILE ORGANIZATION

Files contain records which are collection of information arranged in a specific manner. File organization mainly refers to the logical arrangement of data in a file system. In order to be able to retrieve a target record from a file, it is preferred to be arranged in some defined or proper way. It is necessary to organize data records in a particular pattern. The proper arrangement of records within a file is known as *file organization*.

There are various ways in which records in a file can be stored. Files are presented to the application as a stream of bytes and at the end, it contains an EOF (end of file) mark. An attribute or combination of attribute values that are used to uniquely identify records within a file is called as a *key*. Keys are used to arrange and/or to retrieve records to/from a file. Primary key is one of the keys that can be used to identify a unique record in a file. Non-primary keys are called as secondary keys.

14.3.1 Schemes of File Organization

Various schemes for file organization are available. All these schemes decide the way in which records are stored and accessed in a file. Some of the file organizations are as follows:

Sequential file In sequential file, records are stored in the sequential order of their entry. This is the simplest kind of data organization. In sequential files, the records are stored in ascending or descending order of keys. When the records are not arranged in an organized fashion, they are stored as per their sequence of arrival; this organization is known as serial organization.

Direct or random access file Though we search the records using a key, we still need to know the address of the record to retrieve it directly. The file organization that supports such access is called as direct or random file organization. The word 'random' refers to the fact that the records are not usually stored in sequence but randomized to individual storage positions. So to get the address of the record using a key, there must be some relationship between the key and the address. With direct access file, the address for record storage and retrieval is computed by using a 'hashing' algorithm. As we retrieve the record directly with the help of the key and the hash function, without considering the position of the record in the file, the organization is known as direct access file organization.

Indexed sequential file Records are stored sequentially but the index file is prepared for accessing the record directly. An index file contains records ordered by a record key. The record key uniquely identifies the record and determines the sequence in which it is accessed with respect to other records.

Multi-indexed file In a multi-indexed file, the data file is associated with one or more logically separated index files. Inverted files and multilist files are examples of multi-indexed files.

14.3.2 Factors Affecting File Organization

File organization describes a way in which the records are stored in a file. The objective of file organization is to provide predefined and efficient means for the record storage, retrieval, and update.

The update process includes changes in some of the existing fields of records, addition of new records, or deletion of some existing records. The retrieval of data is done by specifying values for some or all the keys. A query is a combination of key values formed for retrieval of a specific record. Some factors affect file organization and similarly, file organization affects the design of algorithms as it deals with the records in the file.

The factors that mainly affect file organization are the following:

Storage device The way data is arranged in a file depends on the storage device. The magnetic tape is suitable for sequential organization. Direct access devices such as hard disks are suitable for random access file organization.

Type of query Depending on the type of query, file organization will be affected. In a simple query, values for the single key are specified. In a range query, range for the keys is specified. Accordingly, the file organization needs to be changed.

Number of keys The file may or may not have a key. Each key may have one or more fields. Accessing the desired record is made easy with the keys.

Mode of retrieval/update of record The mode of retrieval or update may be real-time or batched. In real-time retrieval, the response time for any query should be minimum. In a railway reservation system, the availability of a particular train should be retrieved in minimum time, whereas in a payroll system all records are processed in a batch.

14.3.3 Factors Involved in Selecting File Organization

Choosing a specific file organization depends on the nature of data and the algorithm used in the application. The overall combination should achieve good performance. The following are the criteria used to choose file organization:

Speed Rapid access to a single record or a collection of records

Operations Convenience of update, that is, addition, modification, or deletion of records

Capacity Efficiency of storage

Size Volume of transaction

Integrity Redundancy, being the method of ensuring data integrity

Security Special backup and recovery processes must exist to prevent exposure to the risks of loss of accuracy

A file should be organized in such a way that the records are always available for processing with no delay. This should be done in line with the activity and volatility of the information.

14.4 FILES USING C++

File handling is an important part of programming. Most of the applications have their own features to save data to the secondary storage and read from it again. File I/O classes in C++ simplify such file read/write operations.

14.4.1 File I/O Classes

The I/O system of C++ contains a set of classes that define the file handling methods. They are ifstream, ofstream, and fstream. These classes are included in the 'fstream.h' header file.

ifstream This class provides input operations.

ofstream This class provides output operations.

fstream This class provides both input and output operations.

14.4.2 Primitive Functions

There are several ways of reading (or writing) the text from (or to) a file, however, all of them share a common approach as follows.

1. Open the file
2. Read (or write) the data
3. Close the file

Opening a file Creating a file stream object to manage the stream using the ofstream, ifstream, or fstream classes is done using the following commands. The file name can be initialized while creating an object.

1. To create an object `ofile` and open a file with name `student.dat` for output only
```
ofstream ofile("student.dat");
```
2. To create an object `ifile` and open a file with name `sports.dat` for input only
```
ifstream ifile("sports.dat");
```
3. To create an object `file1` and open a file with name `employee.dat` for input and output
```
fstream file1("employee.dat");
```

4. To open a file using `open()`

```
ofstream.ofile;    // create an object ofile
ofile.open("sports");      // open file "sports" for output
fstream file;      // create object file of fstream class
file.open(filename, mode);
```

The file mode parameters are given in Table 14.2.

For example,

```
file.open("data", ios::out | ios::binary);
// Binary file with name data is opened in output mode, i.e., for writing only
```

Table 14.2 File mode paratmeters

Mode	Meaning
`ios::app`	Append to the end of file
`ios::ate`	Go to the end of file on opening
`ios::binary`	Binary file
`ios::in`	Open file for reading
`ios::nocreate`	Open fails if file does not exist
`ios::out`	Open file for writing

Reading a character from a file A single character is read from a stream using the `get()` function.

```
file.get(ch);      // read one character from the file and store it to ch
```

Writing a character to a file The `put()` function writes a character to a stream.

```
file.put(ch);      // write the character of ch to the file
```

Reading binary data from a file The `read()` function is used to read binary data from a file.

```
file.read((char *) &V, sizeof(V));      // Reads value in variable V
                                        // from file
```

Consider the following code:

```
class item_rec
{
   int id;
   char itemname[20];
};
item_rec item;      // object item
file.read((char *) &item, sizeof(item));      // Reads item record from
                                              // file
```

Writing binary data to a file The `write()` function is used to write binary data to a file.

```
file.write((char *) &V, sizeof(V));     // Writes value of V in file
file.write((char *) &item, sizeof(item));     // Writes item record
                                              // to file
```

Manipulating file pointers The `seekg()` function moves the `input(get)` pointer to a specific position.

```
seekg(offset, reference);
```

Here `offset` is the number of bytes and `reference` may be one of the following:

1. `ios::beg`—from the start of the file
2. `ios::end`—from the end of the file
3. `ios::cur`—from the current position of the file
4. `seekp()`—moves the `output(put)` pointer to a specific position
5. `tellg()`—gives the current position of the `get` position
6. `tellp()`—gives the current position of the `put` position

Checking end of file The `eof()` function is used to check the EOF.

```
if(file.eof())
cout << "\n EOF";
```

or we can check `eof` using the following statement:

```
if(!file)
cout << "\n EOF";
```

Closing a file To close a file, we can use the `close()` function as follows:

```
file.close();
ifile.close();
```

File handling in C++ is demonstrated using Program Code 14.1.

PROGRAM CODE 14.1

```
//Sample program in C++ for file handling
#include<iostream.h>
#include<stdio.h>
#include<stdlib.h>
#include<fstream.h>
#include <string.h>
// class for storing passenger record
class passenger
{
   char f_name[15], l_name[15];
   int age;
  public:
      void get_data();
      void put_data();
};

// Function for getting passenger data
void passenger :: get_data()
{
   cout << endl << "Enter First name: ";
```

```
    cin >> f_name;
    cout << endl << "Enter Last name: ";
    cin >> l_name;
    cout << endl << "Age: ";
    cin >> age;
}

// Function for displaying passenger data
void passenger :: put_data()
{
    cout << endl << " \t" << f_name << "\t" << l_name
    << "\t" << age;
}

class PassengerFile
{
    private:
        char fname[12];
    public:
        void getfilename()
        {
            cout << "\n Enter filename : " ;
            cin >> fname;
        }
        void create();
        void displayall();
};

void PassengerFile :: create()
{
    fstream file;
    int n, i;
    file.open(fname, ios::out | ios::binary);
    cout << "\nHow many records do you want to enter?";
    cin >> n;
    for(i = 0; i < n; i++)
    {
        p.get_data();
        file.write((char*) &p, sizeof(p));
        flushall();
    }
    file.close();
}
```

```
void PassengerFile :: displayall()
{
   passenger p;      // object for passenger
   fstream file;
   file.open(fname, ios::in);
   if(file.bad())
      cout<<"\nOpening error...";
   else
   {
      cout << "\nid   Fname   Lname  Age \n";
      while(!file.eof())
      {
         file.read((char*) &p, sizeof(p));
         if(!file.eof())
         {
            p.put_data();
         }
      }
      file.close();
   }
}

void main()
{
   Passengerfile pfile;
   pfile.getfilename();
   pfile.create();
   pfile.displayall();
}
```

14.4.3 Binary and Text Files

The file in C++ is either a binary file or a text file. The difference between the two is due to the format in which data is organized within the file. The text file contains plain ASCII characters. It contains text data which is marked by 'end_of_line' at the end of each record. This end of record mark helps to perform operations such as read and write easily. A text file cannot store graphical data. On the other hand, a binary file consists of binary data. It can store text, graphics, and sound data in binary format. Binary files cannot be read directly.

C++ uses the `fopen(file, mode)` statement to open a file and the mode identifies whether you are opening the file to read, write, or append and also whether the file is to be opened in binary or text mode. C++ opens a file by linking it to a stream

so we do not have to specify whether the file is to be opened in binary or text mode on the open statement. Instead the method that we use to read and/or write the file determines which mode we are using. If we use '<<' to read from the file and the '>>' operator to write to the file, then the file will be accessed in binary mode.This is illustrated in Program Code 14.2.

PROGRAM CODE 14.2

```
//Implementation of a simple text file in C++
#include <stdio.h>
#include<conio.h>
#include<iostream.h>
#include<fstream.h>
#include<string.h>
#include<process.h>
fstream fp, fp1; // declaration of file objects
//create a file by entering characters and at end enter #
class myfile
{
   char fname[30];
   public:
      myfile(char tname[30])
      {
         strcpy(fname,tname);
      }
      void create();
      void display();
      void display(char*);
      void count();
      void copy(char*);
};
void myfile :: create()
{
   char ch;
   fp.open(fname, ios::out);      /* Open the file in
   write mode */
   cout << "\nEnter the text::\n";
   do
   {
      ch = getchar();      /* read character   */
      /* write character in file */
     if(ch != '#')
```

```
         fp.write((char *)&ch, sizeof(ch));
   }while(ch != '#');
   fp.close();      // close a file
}

// Display a text file
void myfile :: display()
{
   char ch;
   fp.open(fname, ios::in);      /* Open the file in
   read mode */
   while(!fp.eof())
   {
      /* read character from file */
      fp.read((char *)&ch, sizeof(ch));
      cout << ch;      /* display character */
   }
   fp.close();
}

// Display a text file
void myfile :: display(char tname[30])
{
   char ch;
   fp.open(tname,ios::in);      /* Open the file in
   read mode */
   while(!fp.eof())
   {
      /* read character from file */
      fp.read((char*)&ch, sizeof(ch));
      cout << ch;      /* display character */
   }
   fp.close();
}

/* Function to count the number of lines, words, and
characters */
void myfile :: count()
{
   char ch;
   int c = 0, w = 0, line = 0;
   fp.open(fname, ios::in);
```

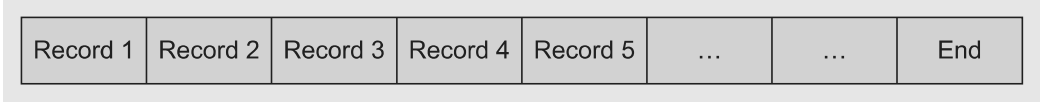
```
   while(!fp.eof())
   {
      fp.read((char *)&ch, sizeof(ch));
      c++;
      if((ch == ' ' || ch == '\n' || ch == '\t'))
         w++;
      if(ch == '\n')
         line++;
   }
   fp.close();
   printf("\nNo of lines %d \nNo of words %d \nNo of
   chars %d", line, w, c);
}

// Copy source file to destination file
void myfile :: copy(char dfname[30])
{
   char ch;
   fp.open(fname, ios::in);      /* Open source file in
   read mode */
   fp1.open(dfname, ios::out);      /* Open
   destination  file in write mode */
   while(!fp.eof())
   {
      /* read character from source file */
      fp.read((char *)&ch, sizeof(ch));
      /* write character to destination file */
      fp1.write((char *)&ch, sizeof(ch));
   }
   fp.close();
   fp1.close();
}
void main()
{
   int choice;
   myfile fobj("c:\\vst\\stud.dat");
   char fname[30];
   clrscr();
   do
   {
     printf("\n\n Menu");
     printf("\n 1. Create");
     printf("\n 2. Display");
```

```
        printf("\n 3. Count lines, words, characters");
        printf("\n 4. Copy file");
        printf("\n 5. Exit");
        printf("\nEnter your choice");
        scanf("%d", &choice);
        switch(choice)
        {
           case 1: fobj.create();
           break;
           case 2: fobj.display();
           break;
           case 3: fobj.count();
           break;
           case 4:
              char dfname[30];
              cout << "\nEnter the destination filename : ";
              cin >> dfname;
              fobj.copy(dfname);
              fobj.display(dfname);
           break;
           case 5: exit(0);
        }
    }while(choice < 5);
    getch();
}
```

14.5 SEQUENTIAL FILE ORGANIZATION

A sequential file stores records in the order they are entered. The order of the records is fixed. The records are stored and sorted in physical, contiguous blocks. Within each block, the records are in sequence. New records always appear at the end of the file. Therefore, the record found in the first position is the oldest record and the last record in the file is the one most recently added. Records in these files can only be read or written sequentially. Records may be either fixed or variable in length for this file type. This is a significant advantage of sequential files. However the search time associated with sequential files is more because records are accessed sequentially from the beginning of the file. Sequential files are compatible to the magnetic tape storage as shown in the following figure.



1. Here, if we want to access Record 5, then we have to access records from Record 1 to Record 5 sequentially.
2. If we want to add a record, it is added at the end.

Important data is usually processed in sequential files, the reason being security is easily ensured in sequential files.

14.5.1 Primitive Operations

The set of primitive operations for a sequential file is small. The file pointer or currency pointer is a logical pointer to the current record in a file. Programming languages support explicit command to move file pointer, and a file pointer is also moved implicitly by primitive operations.

Primitive operations are those provided by the basic file system, language, and operating system. The following are the primitive operations of the sequential file organization:

Open This operation opens the file and sets the file pointer to the first record.

Read-next This operation returns the next record to the user. If no record is present, then EOF condition will be set.

Close This operation closes the file and terminates access to the file.

Write-next File pointers are set to next of last record and this record is written to the file.

EOF If EOF condition occurs, this operation returns true, otherwise it returns false.

Search This operation searches for the record with a given key.

Update The current record is written at the same position with updated values.

The number of records in a sequential file is given as (size of file)/(size of a record). The basic file operations are discussed as follows:

Add

Adding a record to the sequential file is a one-operation algorithm. The new record is simply appended to the end of the file. One physical write is required for appending a record in a file. Also many records can be collected in the buffer and a block of records can be written at a time in the file. The following are the steps involved in addition.

1. Add a record.
2. Open a file in append mode.
3. Read a record from user.
4. Write a record to the file.
5. Close the file.

Search

A particular record is searched through the file using a key sequentially by comparing with each record key. The search starts from the first record and continues till the EOF. The following steps are involved in searching:

1. Open a file in read mode.
2. Read the value of the record key of the record to be searched.
3. Read the next record from the file.
4. If record key = value, display record and go to 7.
5. If not EOF, then go to 3.
6. Display 'Record not found'.
7. Close the file.

Delete

There is no reasonable way to delete records from sequential file. Deletion is done in two ways:

1. Logical deletion
2. Physical deletion

Logical deletion When disk files are used, records may be *logically deleted* by just flagging them as having been deleted. This can be done by assigning a specific value to one of the attributes of the record. This method needs one extra field to be maintained with each record. The algorithm also needs to modify and check the flag field during operations.

Another method keeps a record of active and deleted records in a *bit map* file. A bit map is a one-dimensional array in which each bit represents a record in a file. The first bit refers to the first record, and so on. Bit value '1' tells that the record is active and '0' indicates that the record is deleted. So to delete a record, its corresponding bit value in a bit map file is set to zero. However, the map array may be stored in a separate file or in the beginning of the same file, as one or more records. The following steps are involved in logical deletion:

1. Open a file in read + write mode.
2. Read the record key of the record to be deleted.
3. Read the next record from the file.
4. If record key = value
   (a) Change status or deleted flag as 1
   (b) Write record back to the same position
   (c) Go to step 7
5. If not EOF, then go to 3.

6. Display 'Record not found'.
7. Close the file.

Physical deletion (pack or reorganize) For *physical deletion* of records, we need to copy the records to another file, skipping the deleted records, and rename the file. When the number of the logically deleted records is high, then it is advisable to delete them physically which is known as *reorganization of file*. The following steps are involved in physical deletion (pack):

1. Open a file in read mode.
2. Open 'temporary' file in write mode.
3. Read the record key of the record to be deleted.
4. Read the next record from the file.
5. If record key ! = value, write the record to temporary file.
6. If not EOF, then go to 4.
7. Close both the files.
8. Delete the original file.
9. Rename temporary file as original file.
10. Close the file.

Updation (Modification)

A record is updated when one or more fields is changed by modifying the information. The following steps are involved in updation:

1. Open a file in write mode.
2. Read the record key of the record to be modified.
3. Read the new attributes of the record to be modified.
4. If record key = value, modify record and go to 7.
5. If not EOF, then go to 4.
6. Display 'Record not found'.
7. Close the file.

14.5.2 Advantages

The following are the main advantages of sequential file organization:

1. Owing to its simplicity, it can be used with a variety of media, including magnetic tapes and disks.
2. It is compatible with variable length records, while most other file organizations are not.
3. Security is ensured with ease.
4. For a run in which a high proportion of a block is hit, as compared to other file organizations, sequential file is efficient specially when processed in batches.

14.5.3 Drawbacks

The following are some drawbacks of using sequential file organization:

1. Insertion and deletion of records in in-between positions cause huge data movement.
2. Accessing any record requires a pass through all the preceding records, which is time consuming. Therefore, searching a record also takes more time.
3. Needs reorganization of the file from time to time. If too many records are deleted logically, then the file must be reorganized to free the space occupied by unwanted records.

14.6 DIRECT ACCESS FILE ORGANIZATION

Files that have been designed to make direct record retrieval as easy and efficient as possible are known as directly organized files. This is achieved by retrieving a record with a key by getting the address of a record using the key. To achieve this, a suitable algorithm, called as hashing, is used to convert the keys to addresses.

Direct access files are of great use for immediate access to large amounts of information. They are often used in accessing large databases. When a query concerning a particular subject arrives, we compute which block contains the answer and then read that block directly to provide the desired information. A random access file is one in which the records are accessed directly by referring to the address where it is placed in a file.

One way to achieve this is to use the record number or the primary key (unique identification) as an address of record. In this approach, Record_No gives the location of the record in a file. In this respect, the file looks like a one-dimensional array where each element in an array is a record and the subscript is a record number.

If the range of the record number or the primary key is larger than that of the the file size, it is difficult or rather impossible to adopt the aforementioned strategy. This happens in some applications where only some of the records are selected (randomly) out of the many records. For example, out of 1000 students from a university, the data of the 100 computer science students is to be managed. However when they are admitted at a university, they are given a unique ID called an enrollment number, which ranges from 1 to 1000. Hence, we cannot adopt this strategy of inserting the element in position as the enrollment number for direct access.

To achieve direct access by having a file size as total number of records, another technique is used. In this technique, mapping of a larger range is done to a smaller range. To do this, a function is used, which generates a natural address (whose range lies between 1 and file size) from primary key of larger range. This function is known as the *hash function*, for example, MOD (primary key MOD N). A *synonym* is defined as a key, which generates the same address as that generated by a different key. A good hashing function must minimize the creation of synonyms. We have discussed hashing in Chapter 11.

A well-designed direct access file gives a very fast response to random queries than a sequential file. Many applications need both sequential and random access files. Though

direct files can be processed sequentially, it would be much higher when sequential file is organized in a proper manner.

14.6.1 Primitive Operations

The primitive operations for the direct access file are as follows:

Open It opens the file and sets the file pointer to the first record.

Read-next It returns the next record to user. If no records are present, then EOF (end of file) condition will be set.

Read-direct It sets the file pointer to a specific position and gets the record for the user. If the slot is empty or out of range, then it gives error.

Write-direct It sets the file pointer to a specific position and writes the record to file at that position. If the slot is out of range, then it gives error.

Update Current record is written at the same position with updated values.

Close This will terminate the access to the file.

EOF If EOF condition occurs, it returns true otherwise it returns false.

We can use the `fseek()` function for direct access. The prototype of `fseek()` is :

```
int fseek(File *fp, long num-bytes, int origin);
```

The `fseek()` function sets the file position indicator. Here `fp` is a file pointer. The `num-bytes` parameter specifies the number of bytes from the origin that will become the new current position and `origin` can be one of the following as shown in Table 14.3.

Table 14.3 File position indicators

Origin	Value	Macro name
Beginning of file	0	seek _set
Current position	1	seek_cur
End of file	2	seek _ end

The `fseek()` function returns 0 when successful, and a non-zero value in case of an error. The implementation of direct access file organization is demonstrated in Program Code 14.3.

PROGRAM CODE 14.3

```
/* Direct access file. Collision handling to be done
by chaining without replacement for employee data as
empcode, empname */
```

```
#include<iostream.h>
#include<string.h>
#include<conio.h>
#include<fstream.h>
#include<process.h>
#include<math.h>
#define max 15
class employee
{
   public:
      char name[max];
      int empid;
      int chain;
      int delflag;
};

class hashfile
{
   fstream hfile;
   public:
      hashfile();
      int hash(int x){return x % 10;}
      void insert();
      void search();
      void display();
};

// function to initialize empty file
hashfile :: hashfile()
{
   int i;
   employee rec2;
   fstream iofile;
   iofile.open("hfile.dat", ios::out | ios::binary);
   strcpy(rec2.name, "\0");
   rec2.chain = -1;
   rec2.delflag = 0;
   for(i = 0; i < 10; i++)
   {
      rec2.empid = 0;
      iofile.write((char*)&rec2, sizeof(rec2));
   }
```

```
    iofile.close();
}

// function to insert a record in hash file
void hashfile :: insert()
{
    int i, flag = 0, pos, cnt = 0;
    long temp, start, size;
    fstream iofile;
    employee insertrec, rec3, trec;
    cout << "Enter name";
    cin >> insertrec.name;
    cout << "Enter no. of empid";
    cin >> insertrec.empid;
    insertrec.chain = -1;
    insertrec.delflag = 0;
    size = sizeof(insertrec);
    pos = hash(insertrec.empid);
    iofile.open("hfile.dat", ios :: in | ios :: out | ios
    :: binary);
    iofile.seekg(0);
    temp = pos * sizeof(insertrec);
    iofile.seekg(temp);
    // move to position given by hash function
    flag = 0;
    iofile.read((char*) &rec3, sizeof(rec3));
    if(rec3.empid == 0)       // slot is empty
   {
       flag = 1;
       temp = pos * sizeof(rec3);
       iofile.seekp(temp);       /* move to position
       given by hash function */
       iofile.write ((char*) &insertrec, sizeof(insertrec));
       return;
    }
    else       // slot is not empty
    {
       if(hash(rec3.empid) == hash(insertrec.empid))
       {
          while(rec3.chain != -1)
          {
             iofile.seekg(rec3.chain * sizeof (rec3));
```

```
            pos = rec3.chain;
           iofile.read((char*) &rec3, sizeof(rec3));
         }
         flag = 2;
      }
      int nextpos = pos;
      trec = rec3;
      while(iofile.read((char*) &rec3, sizeof(rec3)))
      // find next empty position
      {
         if(rec3.empid == 0)      // empty slot
         {
            iofile.seekp((nextpos+1) * sizeof(rec3));
            // move to position given by hash function
            iofile.write((char*) &insertrec,
            sizeof(insertrec));
            if(flag == 2)
            {
               iofile.seekp(pos * sizeof (rec3));
               trec.chain = nextpos + 1;
               iofile.write ((char*) &trec, sizeof(trec));
            }
            flag = 1;
            break;
         }
         nextpos++;
      }
   }
   if(flag != 1)
   {
      cout << "Error this rec was not inserted";
      cout << "The file is full after this index";
      getch();      return;
   }
   getch();
   iofile.close();
}      // end of insert

// function to search a record of hash file
void hashfile :: search()
{
   int pos = 0, t_empid;
```

```
    fstream iofile;
    employee rec1;
   cout << "Enter the empid of the book to be searched";
    cin >> t_empid;
    pos = hash(t_empid);
    // get the position of search record
    iofile.open("hfile.dat", ios::in | ios::binary);
    iofile.seekg(0);
    iofile.seekg(pos*sizeof(rec1));
    while(iofile.read((char *)&rec1, sizeof(rec1)))
    // read record at position
    {
       if(rec1.empid == t_empid)      // found
       {
         cout << "name" << rec1.name << "empid" << rec1.
empid;
          getch();
          iofile.close();
          return;
       }
       else if(hash(rec1.empid) == pos)
       // if record is stored at position
       {
          iofile.seekg(0);
          if(rec1.chain != -1)
             iofile.seekg(rec1.chain*sizeof(rec1));
             // jump at position of chain
       }
    }
    cout << "error no. such rec exist";
    getch();
    iofile.close();
}

void hashfile :: display()
{
    int i = 0;
    employee rec2;
    fstream iofile;
    cout << "\n\nserial\tempid\tname\tchain";
    iofile.open("hfile.dat", ios :: in | ios :: binary);
    while(iofile.read((char *)&rec2, sizeof(rec2)))
```

```
      cout << "\n\n" << i++;
      cout << "\t" << rec2.empid;
   {
      cout << "\t" << rec2.name;
      cout << "\t" << rec2.chain;
   }
   getch();
   iofile.close();
}

void main()
{
   int ch, pos;
   float flag = 1.1;
   hashfile file1;
   // rec.init();
   // clrscr();
   do
   {
      cout << "\n 1.Insert a rec";
      cout << "\n 2.Disp  all rec";
      cout << "\n 3.Search a rec";
      cout << "\n 4.Exit";
      cout << "\n Enter choice";
      cin >> ch;
      switch(ch)
      {
         case 1:
            file1.insert();
         break;
         case 2:
            file1.display();
         break;
         case 3:
            file1.search();
         break;
         case 4:
            exit(0);
      }
   }while(ch != 4);
}
```

14.7 INDEXED SEQUENTIAL FILE ORGANIZATION

Sequential processing of data files makes up a larger proportion of data. However, there is often a need to refer to sequential files just to satisfy the queries. Such a need can be met by processing the whole file sequentially and looking for the records that are to be retrieved. This is very efficient when the file is huge, the query may take long time, which is not affordable for the application. One solution is to improve the speed of retrieving target by using indexed sequential file.

A file that is loaded in key sequence but can be accessed directly by use of one or more indices is known as an indexed sequential file. A sequential data file that is indexed is called as indexed sequential file.

An indexed file contains records ordered by a *record key*. Each record contains a field that contains the record key. The record key uniquely identifies the record and determines the sequence in which it is accessed with respect to the other records. An indexed file can also use *alternate indices*, that is, record keys that let you access the file using a different logical arrangement of the records. For example, you could access the file through the employee department rather than through the employee number.

When indexed files are read or written sequentially, the sequence followed is that of the key values. Index is a data structure that allows particular records in a file to be located more quickly. An index can be sparse (record for only some of the search key values) or dense (index is maintained for each record), e.g., index in a book.

14.7.1 Types of Indices

Indices may be of the following three types:

Primary index It is an index ordered in the same way as the data file, which is sequentially ordered according to a key. The indexing field is equal to this key.

Secondary index This is an index that is defined on a non-ordering field of the data file. In this case, the indexing field need not contain unique values.

Clustering index A data file can associate with utmost one primary index and several secondary indices. In this organization, key searches are improved. The single-level indexing structure is the simplest one where a file, whose records are pairs, contains a key and a pointer. This pointer is the position in the data file of the record with the given key.

A key search is performed as follows: the search key is compared with the index keys to find the highest index key coming in front of the search key, while a linear search is performed from the record that the index key points to, until the search key is matched or until the record pointed to by the next index entry is reached.

Hardware for indexed sequential organization is usually disk-based, rather than tape. Records are physically ordered by primary key and the index gives the physical location of each record. Records can be accessed sequentially or directly, via the index. The index

is stored in a file and read into memory at the point when the file is opened. The indices must also be maintained.

14.7.2 Structure of Indexed Sequential File

The file structure is selected according to the physical storage device. The external storage device should have the capability to access directly a record as per the key. Devices like magnetic tape can access all records sequentially. The magnetic drum or disk supports direct access.

In primary area, actual data records are stored. Data records are stored as sequential file. The second area is an index area in which the index is stored and is automatically generated. An index file consists of three areas:

Primary storage area This includes some unused space to allow for additions made in data.

Separate index or indices Each query will reference this index first; it will redirect query to part of data file in which the target record is saved.

Overflow area This is optional separate overflow area.

A number of index levels may be involved in an index sequential file. The lowest level of an index is track index, which is written at track 0, i.e., first track of the cylinder. The track index contains two entries for each prime track of the cylinders for the index sequential file. The normal entry is composed of the address of prime track to which the entry is associated and the highest value of the keys for the records is stored on that track. The track index describes how records are stored on the track of cylinder and the cylinder index indicates how records are distributed over number of cylinders. In index sequential file, records are organized in sequence of key field known as primary key. For fast searching, it is supported by index. Index is a pair of key and address where that record is stored in the main file. Number of records are same as number of blocks of the main file.

14.7.3 Characteristics of Indexed Sequential File

The following are the characterictics of an indexed sequential file:

1. Records are stored sequentially and a separate index file is maintained for accessing the record directly.
2. Records can be accessed randomly in constant time.
3. Magnetic tape is not suitable for indexed sequential storage.
4. Index is the address of physical storage of a record.
5. When very few records are to be accessed, then indexed sequential file is better.
6. This is a faster access method.
7. Additional overhead is that the index is to be maintained
8. Indexed sequential files are popularly used in many applications such as a digital library.

Consider that an employee file is stored as an indexed sequential file. The entries are as shown in Fig. 14.1.

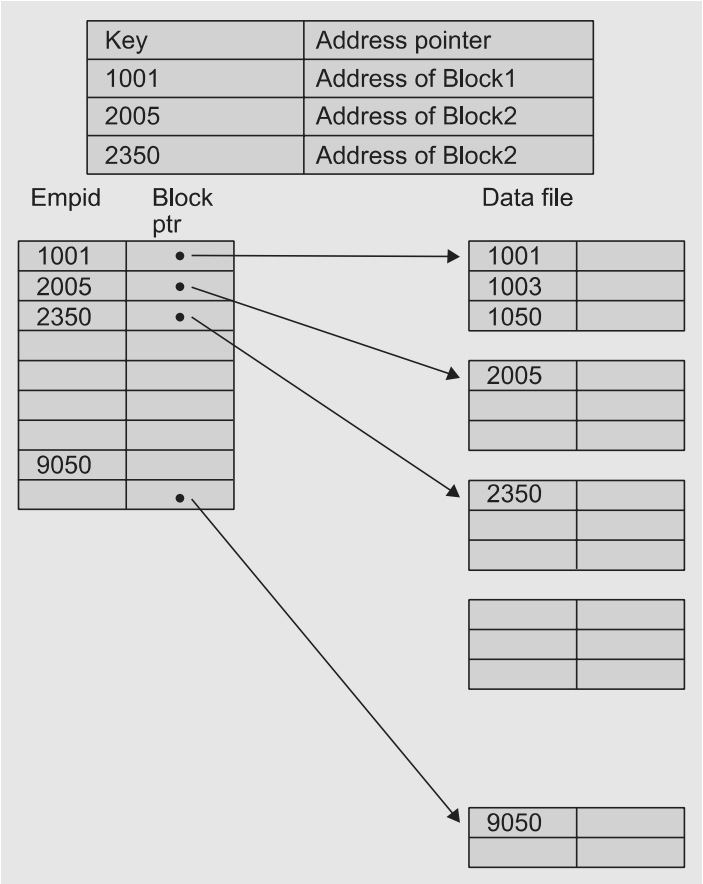


Fig. 14.1 Index sequential file organization

This organization is slower than a sequential file. For a sequential file, retrieval and access time for direct retrieval are greater than the well-designed direct access file. The advantage of such file organization is that it can handle requirements of mixed access application much better than other organizations.

Advantages

1. Accessing any record is more efficient than sequential file organization.
2. Large amount of data can be stored using this type of file organization.

Disadvantage

1. Often more than one index is needed which occupies a large storage area.

Program Code 14.4 demonstrates the implementation of index sequential file.

PROGRAM CODE 14.4

```
//Index sequential file (one to one map index)
#include <stdio.h>
#include <conio.h>
#include<iostream.h>
#include<fstream.h>
/* Item record */
struct itemrec
{
   int itemcode;
   char itemname[20];
   float cost;
};

/* Index record */
struct indexrec
{
   int itemcode;
   int position;
   int flag;
};

/* Display the contents of index file */
void displayindexfile()
{
   fstream indexfile;
   struct indexrec index;
   indexfile.open("index.dat", ios::in);
   cout << "\n Index file is n";
   cout << "\n Itemcode \t Position \t Del Flag";
   while(!indexfile.eof())
   {
      indexfile.read((char*)&index, sizeof(indexrec));
      if(indexfile.eof())
         break;
      if(index.flag == 1)
         cout << endl << index.itemcode <<
         index.position<<index.flag;
      else
         cout << "deleted=";
         cout << endl << index.itemcode <<
         index.position << index.flag;
```

```
    }
}

/* Insert a record */
void insertrecord()
{
    struct itemrec item;
    struct indexrec index;
    stream indexfile, itemfile;
    long position;
    cout << "\n Enter itemcode";
    cin >> item.itemcode;
    cout << "\n Enter itemname";
    cin >> item.itemname;
    cout << "\n Enter cost";
    cin >> item.cost;
    /* Get the position of the new record in item file */
    itemfile.open("item.dat", ios::in);
    itemfile.seekg(seek_end);
    position = itemfile.tellg()/sizeof(item);
    itemfile.close();
    /* Add a record in item file */
    itemfile.open("item.dat", ios::in | ios::out |
    ios::app);
    itemfile.write((char*)&item, sizeof(itemrec));
    itemfile.close();
    /*Add a record in index file */
    indexfile.open("index.dat", ios::in | ios::out |
    ios::app);
    index.itemcode = item.itemcode;
    index.position = position;
    index.flag = 1;
    indexfile.write((char*)&index, sizeof(indexrec));
    indexfile.close();
}

/* Search a record */
void search()
{
    int searchitcode;
    struct itemrec item;
    struct indexrec index;
   fstream indexfile, itemfile;
    long position, found = 0;
```

```
    cout << "\n Enter itemcode to be searched";
    cin >> searchitcode;
    indexfile.open("index.dat", ios::in);
    while(!indexfile.eof())
    {
       indexfile.read((char*)&index, sizeof(indexrec));
       if(index.itemcode == searchitcode && index.flag
       == 1)
       {
          found = 1;
          break;
       }
   }
    if(found == 1)
    {
       itemfile.open("item.dat", ios::in);
       /*Take the position from index file and go to
       that record in item file */
       itemfile.seekg((index.position) * sizeof(item));
       itemfile.read((char*)&item, sizeof(item));
       cout << "\n Item Record is";
       cout << "\nItemcode \t Item name \t Cost";
       cout << item.itemcode << item.itemname <<
       item.cost;
       itemfile.close();
    }
    else
      cout << "\n Record not found";
       indexfile.close();
}

/* Delete a record */
void deleterecord()
{
    int searchitcode;
    struct indexrec index;
    fstream indexfile;
    long position,found = 0, c;
    cout << "\n Enter itemcode to be deleted";
    cin >> searchitcode;
    indexfile.open("index.dat", ios::in | ios::out |
    ios::app);
    c = 0;
    while(!indexfile.eof())
```

```
    {
       indexfile.read((char*)&index, sizeof(indexrec));
       if(index.itemcode == searchitcode && index.flag
       == 1)
       {
          found = 1;
          break;
       }
       c++;
    }
    if(found == 1)
    {
       indexfile.seekg(c * sizeof(index));
       index.flag = 0;       // Make a delete flag 0
       indexfile.write((char*)&index, sizeof(index));
    }
    else
       cout << "\n Record not found";
       indexfile.close();
}

void main()
{
    int choice;
    clrscr();
    do
    {
       cout << "\n 1. Insert \n 2. Search \n 3. Delete a
       record";
       cout << "\n 4. Display Index file \n 5. Exit";
       cout << "\n Enter choice : ";
       cin >> choice;
       switch(choice)
       {
          case 1 : insertrecord();
          break;
          case 2 : search();
          break;
          case 3 : deleterecord();
          break;
          case 4 : displayindexfile();
        break;
       }
```

```
    }
    while(choice < 5);
}
```

14.8 LINKED ORGANIZATION

In linked organization, the physical sequence of records is different from the logical sequence of records. The next logical record is obtained by following a link value from the present record. Records are linked according to increasing primary key, so insertion and deletion is easy. If index is not maintained, then direct searching is difficult and only sequential search is possible.

14.8.1 Multilist Files

To make searching easy, several indexes are maintained as per primary key and secondary keys, one index per key. The record may be present in different lists as per key. Consider the following file of office staff in Table 14.4.

Table 14.4 Staff data

Staff ID	Occupation	Salary	Record
106	Clerk	5000	A
150	Accountant	4000	B
360	Clerk	3000	C
400	Accountant	3500	D
700	Clerk	2000	E

We can maintain indices on the staff ID. We can group staff ID with ranges 101–300, 301–600, 601–900, and so on. Now all the records with staff ID in the same range will be linked together as shown in Fig. 14.2.

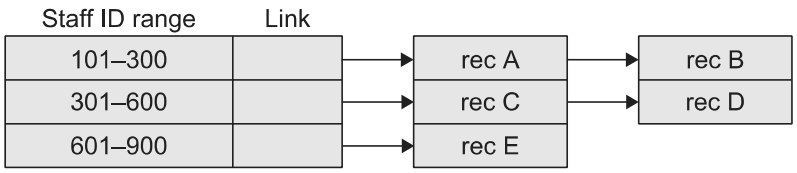


Fig. 14.2 Sample multilist file

Now each record will have values of all the fields as well as link to the next record in the group.

We can have multilist structure for file representation by maintaining different indices on different keys and allow records to be in more than one list. Suppose indices are maintained on occupation and salary fields, then the multilist structure will look as shown in Fig. 14.3.

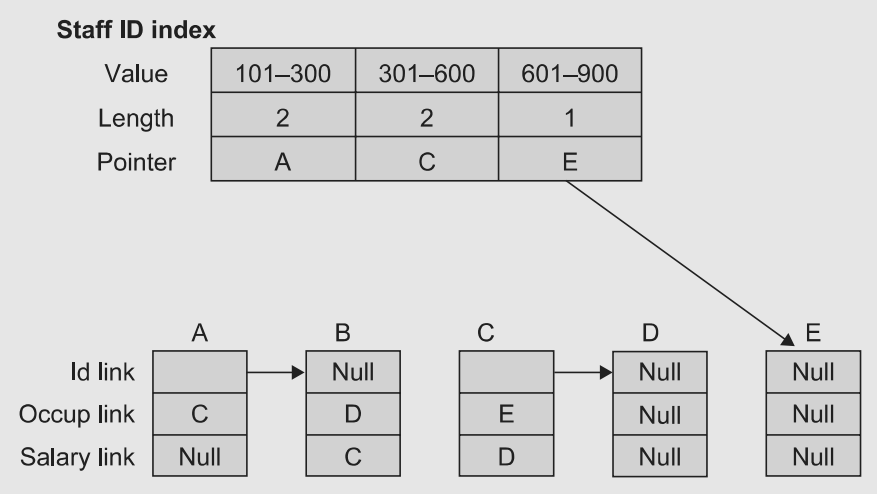


Fig. 14.3 Linked organization

Table 14.5 lists the staff details and links for the following values of occupation and salary:

Occupation index

Value	Clerk	Accountant
Length	3	2
Pointer	A	B

Salary index

Value	<= 2000	<= 4000	<= 6000
Length	1	3	1
Pointer	E	B	A

Table 14.5 Staff data and links

Record	Staff ID	Occupation	Salary	Occupation link	Salary link
A	106	Clerk	5000	C	–
B	150	Accountant	4000	D	C
C	360	Clerk	3000	E	D
D	400	Accountant	3500	–	–
E	700	Clerk	2000	–	–

When multilists are maintained, then length of the link is also maintained in the index. When two lists are searched simultaneously, then the search time can be reduced by searching the smaller list.

The logical order of records in the list may or may not be important according to the application. If salary index is not maintained in increasing order, then insertion can be done at the beginning or at the end of the list, otherwise we have to find a proper position in the link to insert new record. Also because only single link is maintained, deletion is difficult. This problem can be overcome by maintaining double link. But these

links require storage. So, if space is of importance, then the alternative is the coral ring structure.

14.8.2 Coral Rings

In this, doubly linked multilist structure is used as shown in Fig.14.4. Each list is circular list with headnode.

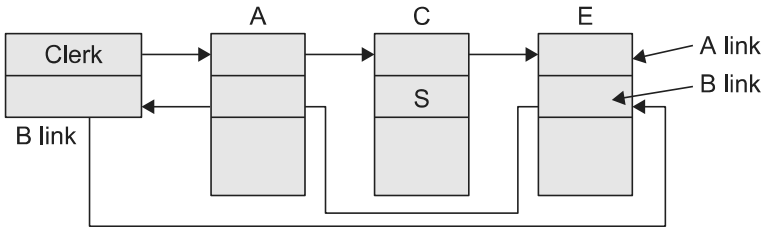


Fig. 14.4 Sample doubly linked list

'A link' field is used to link all records with same key value. 'B link' is used for some records back pointer and for others it is pointer to head node. 'S' is headnode of the list of 'Clerk'. Owing to these back pointers, deletion is easy without going to start. Indexes are maintained as per multilists.

14.8.3 Inverted Files

The concept of the inverted files and multilists is similar. The difference is that, in multilists records with the same key value are linked together and links are kept in each record. But in the inverted files, the link information is kept in the index itself.

For example, consider the same file of office staff used in the link organization. The indices for fully inverted file are shown in Fig. 14.5.

Staff ID index (increasing order)	Occupation index
106	A
150	B
360	C
400	D
700	E

Accountant	B, D
Clerk	A, C, E

Salary index	
2000	E
4000	B, C, D
6000	A

Fig. 14.5 Inverted files

All these are dense indices and contain an entry for each record in the file. But now because links are kept in the indices, index entries become variable length and therefore index maintenance becomes more complex than for multilists.

The inversion process is associated with the information of inverted list. Normally, a record is searched via a primary key. For example, if staff ID is a primary key, then records can be searched using staff ID. But the inverted list provides staff ID and further a particular staff's name and other details can be accessed through index.

In inverted files, the record is accessed in two steps. First, the indices are searched to obtain a list of required records and then second, records are retrieved using these lists. The number of disk accesses required is equal to the number of records being retrieved plus the number to process the indices.

In inverted files, only the index structures are important. The records can be organized sequentially, random, or linked according to primary key. If a list of records is not very large, then it can be kept in main memory while processing. Inverted files may also result in space saving when record retrieval does not require retrieval of key fields. Then key fields may be deleted from the records. One of the major disadvantages of the inverted files is that the item values being inverted generally have to be included in both the inverted list and the master file.

14.8.4 Cellular Partitions

To decrease file search time, the storage media may be divided into cells. A cell may be an entire disk or a cylinder. Lists are localized to lie within a cell. If a cylinder is used as a cell, then all records on the same cylinder may be accessed without moving the read/write heads. We divide multilists organized on several different cylinders into several small lists which are stored on the same cylinder.

For example, consider Table 14.6, an example of a multilist structure with cellular partitioning for student–teacher data.

Table 14.6 Multilist structure with cellular partitioning

	Primary key	Secondary key	
Position	Student ID	Course teacher ID	Link
1	100	A	o
2	200	B	Null
3	300	C	Null
4	400	A	Null
1	500	D	O
2	600	B	Null
3	700	A	Null
4	800	D	Null
1	900	E	Null
2	1000	C	Null
3	1100	D	Null
4	1200	A	Null

Cell 1

Cell 2

Cell 3

An entry is created in the secondary index whenever the item value occurs one or more times in a cellular partition. The relative secondary index records for the data in Table 14.6 are shown in Table 14.7.

Table 14.7 Teacher's data and secondary index

Course teacher ID	Position	Cell no.	Length of link
A	1	1	2
	3	2	1
	4	3	1
B	2	1	1
	2	2	1
C	3	1	1
	2	3	1
D	1	2	2
	3	3	1
E	1	3	1

The course teacher ID 'A' has entries in each cell, at two positions in cell 1, at one position in cell 2, and at one position in cell 3. Therefore, the entry of 'A' has three rows in the secondary index. The course teacher ID 'E' has entry only in cell 3 at position 1, so in the secondary index, 'E' has only one row.

A multilist structure with cellular partitioning is primarily useful when there are a large number of records residing in a cell. If there are few records in each cell, then the link field can be omitted.

The length field and the relative record position can also be omitted. One such structure is cellular serial structure shown in Table 14.8.

Table 14.8 Cellular serial structure

Course teacher ID	Cell no.
A	1
	2
	3
B	1
	2
C	1
	3
D	2
	3
E	3

One more type of structure is the cellular inverted list which is represented as a binary matrix. Each matrix element is either 0 meaning that the item is absent, or 1 meaning that the item is present in the cellular partition. The structure is shown as in Table 14.9.

Table 14.9 Cellular inverted list

Secondary index item	Cell no.		
A	1	1	1
B	1	1	0
C	1	0	1
D	0	1	1
E	0	0	1

For cellular multilist structures, index entries may have to be updated with the addition or deletion of records or individual secondary index items. Such changes are minimal when cellular serial or cellular

inverted lists are used. A major advantage of cellular partitioning is that several read operations can be initiated simultaneously and these operations can be overlapped with the query processing. But the disadvantage is that if there are many records per cell, then access time may be large.

RECAPITULATION

- Data processing is one of the core tasks of computers. Large volumes of data and archival data need to be preserved even after execution of program is over. Such data is commonly stored in the external memory as special data holding entities, files.
- Magnetic tapes, floppy disks, and hard disks are a few examples of secondary storage devices. When we organize data in a file data structure, the data is non-volatile, which means data will reside on storage after data processing is over.
- Files contain records. In order to be able to retrieve a target record from a file, it is preferred to arrange in some defined or proper way. Necessity is to organize data records in a particular pattern. The proper arrangement of records within a file is called as file organization.
- Various schemes for file organization are available such as sequential, direct access, and index sequential organization. All these schemes decide the way in which records are stored and accessed in a file.
- In sequential files, records are stored in ascending or descending order of key and stored as per their sequence of arrival. This type of organization is known as serial organization. When data arises in sorted order, the serial organization becomes sequential organization.
- To get faster access, the records are organized randomly in a file by computing the address using key and hash function. File organization that supports direct access to record by computing its address using key is called as direct or random file organization.
- In index sequential file, the records are stored sequentially. For each record, its corresponding address is saved as index in index file for accessing the record directly.
- C++ supports file operations through library functions.

KEY TERMS

Direct access file organization The file organization that supports direct access to record by computing its address using key is called as direct or random file organization.

File Records that hold information about similar items of data are usually grouped together into a file. A file is a collection of records where each record consists of one or more fields.

File organization File organization refers to the logical arrangement of data in a file system. Various schemes for file organization are available such as sequential, direct access, and index sequential organization.

Index sequential file organization An index file contains records ordered by a record key. The record key uniquely identifies the record and determines the sequence in which it is accessed with respect to other records.

Sequential file organization The simplest kind of data organization, sequential file organization is the one in which records are stored in the sequential order of their entry arising in ascending or descending order of key.