Elements of Data Management Systems

GEORGE G. DODD

General Motors Research Laboratories, Warren, Michigan*

Many different data management techniques have been designed, described in the literature, and marketed. With each technique there are claims of added flexibility and speed. However, because new terms are invented to describe each new technique, the observer is left in a state of confusion when he tries to understand how they work.

A description is given of the basic types of data management techniques, as well as the relation of each to the hardware on which it is used. Then it is shown how these basic elements can be used as building blocks to describe and build more complex data management systems. Finally, there is a discussion of the languages used for programming data management systems.

Key words and phrases: data structures, sequential organization, random organization, list processing, rings, trees, pointers, data pages, direct access devices, programming languages, storage media, space management, hierarchical files

CR categories: 1.3, 3.73, 3.74, 4.22

INTRODUCTION

Data management problems are what the Internal Revenue Service has when it receives 65 million income tax returns every April 15. They are encountered by librarians in cataloging the 17,000 new books published annually, as well as by library users in trying to locate one of them. Business management personnel have them in reviewing the daily, weekly, and monthly reports that arrive from every department in the plant, or every plant in the company. Even computers have them in compiling code for symbolic language programs.

A number of data management systems have been described during the past few years [2, 3, 5, 15, 21, 22], all of which store, retrieve, and update data. A number of seemingly different data organization schemes to support these and other data management systems have also evolved; Figure 1 shows a few of their names.

* Computer Technology Department

These data organizational methods are utilized for two reasons. The demands of the users who are storing, updating, and retrieving data differ. In some situations, such as information retrieval systems, data must be retrieved rapidly but updating of existing data can proceed at a slower pace. In other situations, such as the real-time recording of missile test data, it is necessary to store rapidly large volumes of data which are retrieved later at a slower pace. Each type of demand may call for a different data organization which will satisfy the requirements of the application and at the same time keep overhead at a minimum. Another reason for the existence of more than one data organization is that the characteristics of the media on which the data is stored are not always compatible with the demands of the application. Different data organizations are therefore used to bridge the gap between the logical user requirements and the physical realities of storage media and computer hardware.

CONTENTS

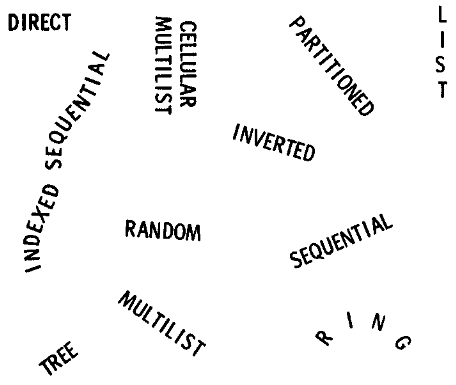


FIG. 1. Data organization schemes

It is convenient to note that all data organizational methods are built up from three basic organizations: sequential, random, and list. We begin by describing and comparing the basic organizations and then demonstrating how they are used as foundation blocks to build the more complex data organizations. As a result of this discussion it will become apparent why the more complex data organizations are often desirable. Then there is a description of the characteristics of the media on which data is stored and an examination of the capabilities of the programming languages available for implementing data management systems.

DEFINITIONS

A few terms are defined here for clarity. These correspond to the USASI Standard COBOL [17] definitions.

Data management systems deal with elementary data items, records, and files. Each piece of raw information the sytem stores, retrieves, and processes is called an *elementary data item*. In general, several elementary data items are used to describe something stored by a data management system.

A collection of such elementary data items is called a *record*. If the system were to store information about an employee, the employee record would contain the name, city, occupation, and age of each person recorded in the system, as shown in Figure 2. The grouping of data items into records is desirable, since for each item processed by the system there is a certain amount of hardware and software overhead. Collecting the items together into records reduces the overhead on a per item basis. A record may contain only one elementary data item in cases where it is desirable to deal with each datum individually.

For bookkeeping purposes records are collected into logical units called *files*. Often each file corresponds to the storage medium on which it is recorded. However, it is possible to store several files on one medium, e.g. disk.

The arrangement and interrelation of records in a file form a *data structure*. Different data structures are used depending upon the operations to be performed on a file.

To put this in proper perspective, a data management system deals with records in a file. These records contain information, called elementary data items, which are the object of processing.

SEQUENTIAL ORGANIZATION

Perhaps the best known data organization is the sequential organization, wherein records are stored in positions relative to other records according to a specified sequence. To order the records in a sequence, one common attribute of the records is chosen. When it is a data item within the record, the attribute selected to order the records in the file is called a *key*. By selecting a different key for a file and sorting on the basis of that key, the sequence of the records in the file may change. Figure 3 shows a sequential organization in which the name data item of each record is used as the key for ordering the file. Records may be stored without keys; this occurs when records are stored in order of their arrival into the system, in which case each record is positioned sequentially following the preceding record. In both cases the logical order of records in the file and the physical order of records on the recording medium is the same.

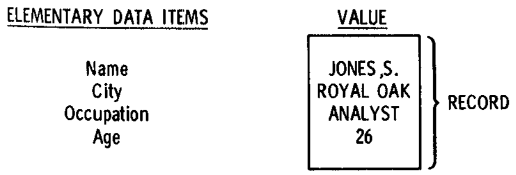


FIG. 2. Example of elementary data items in a record

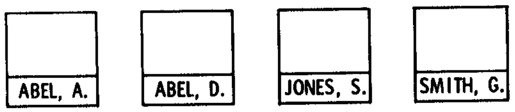


FIG. 3. Sequential organization

A sequentially organized file may be composed of records of different types, but the records are grouped into a file because they have a common functional purpose. The entries in a telephone directory are an example of a sequential organization in which the key for sequencing is the surname data item of the record.

The advantage of sequential organization is fast access per relationship during retrieval. If the access mechanism is positioned to retrieve a particular record, such as the "Jones" record in Figure 3, it can rapidly access the next record in the data structure according to the relationship which was established when the data was stored, in this case, alphabetical order. Where the records are stored in sequence on a direct access medium (e.g. disk or drum) or in high speed memory, a binary search is possible. The data is sampled in the middle, eliminating half the cases in one comparison; the remaining half is then sampled in its middle and the process repeated until a sequential search of a small portion of the file is possible and it is established that the desired record is or is not present.

The advantage of being able to rapidly access the next record becomes a disadvantage when a file is to be searched until a record having a particular key value is encountered. Here the first record is examined; if the key is not correct, the next record is examined and the process continued until the correct record is found.

Updating values in a record stored in a sequential organization is also difficult. Should the new record be shorter or longer than the original record, adjacent records in the file may be destroyed or become inaccessible to the hardware when the record is rewritten. Updating of blocked records, which are described in the "Storage Media" section, is impossible unless the entire block is rewritten. For these reasons a file having sequential organization is usually updated by copying records from one file to another, making updates to the records as needed. It becomes expensive to update one record in a sequentially organized file; usually updating occurs only when a number of records are to be altered.

It is difficult to insert new records into or remove old records from a sequentially organized file. The insertion process requires that already stored records be pushed apart to make room for a new record, resulting in the copying of the entire file. The converse is true for removing records, in which case existing records are pushed together. New records can be added out of sequence to the end of the file, however, and sorted into the proper sequence at a subsequent updating of the file. List organizations, which are described later, provide an easier insertion/removal facility and avoid the copy problems of sequential organization.

It is often desirable to store and retrieve records using more than one key. A case in point might be a record of a personnel file which is to be stored and accessed either by the name or the occupation of the person the record describes. This is most easily accomplished by storing the data in duplicate files so that the first file is searched if the first key is known and the second file is searched if the second key is known. This duplication, however, uses twice the storage space and doubles the maintenance cost, since records to be updated must be accessed and changed in both files.

In summary, sequential organization allows rapid access of the next record in the file but offers difficulties in updating a file and retrieving records out of sequence. To overcome this disadvantage, a second data management technique has been developed; it is called random organization.

RANDOM ORGANIZATION

In random data organization, records are stored and retrieved on the basis of a predictable relationship between the key of the record and the direct address of the location where the record is stored. The address is used when the record is stored and used again when the record is retrieved. Figure 4 shows the record with the key "Jones, S." being stored in a particular position on a direct access medium. Since the relationship between the key and the address of the record is very important for dealing with randomly organized data, let us examine the three methods for acessing records on direct access media [15]: direct address, dictionary look-up, and calculation.

Direct address. In the simplest form, the direct address is known by the programmer and is supplied at storage and retrieval times. The hardware then uses this address to access the record on the storage medium.

Dictionary look-up. Figure 5 illustrates the dictionary look-up and calculation methods by which a record's direct address is obtained prior to storage or retrieval. When the dictionary method is used, both the record's key and its direct address are stored in the dictionary as shown in the upper part of Figure 5. When a record is stored or retrieved, the key is found in the dictionary and the corresponding direct address is used. For example, the key "Jones" is compared with the dictionary and the direct address 0131 is found. The direct address 0131 is then used to store or retrieve the record.

The use of a dictionary insures that

each record has a unique address. However, to do this the dictionary must be large enough to include all potential direct addresses and it may occupy as much space as the data itself. Also, the step-by-step sequential search of a dictionary may offset the gains offered by having unique record addresses. If the dictionaries are very large, they are often accessed with a binary search, or they may have a tree organization (to be described later) to reduce search time.

Calculation. The calculation method [9, 12] involves converting the key of the record into a direct address, which is not necessarily unique. In Figure 5 a simple calculation method is used. Each letter of the key is replaced by a number—J by the number 10 (because J is the 10th letter of the alphabet), O by 15, N by 14, E by 5, and S by 19. These numbers are added and the result is converted to a binary number, whose rightmost 5 bits form two octal digits which are the rightmost 2 digits of the address, with the remaining bits forming the digits in the left part of the address. The result is the direct address of the Jones record.

The direct address and dictionary methods always yield a unique address. However, the calculation method can cause two different keys to calculate to the same direct address. In Figure 5 the record having the key "Jones" and the record having the key "Senoj" calculate to the same direct address because of the calculation algorithm used. This causes "overflow" and may be handled by putting a pointer in the first record at a particular address to point to the overflow record, which is at some other location on the storage medium. The overflow pointer is the direct address of the next record. In this example, both "Jones" and "Senoj" calculate to the address 0131. The storage or retrieval mechanism will access address 0131 and examine the key of the record stored there. If the key is that of the record desired, the record is retrieved. If the key is not that of the record desired, the overflow pointer is used to retrieve another record having the same calculation address. The key of this second record is examined to see if it is the correct record and the process continued until the correct record is found. Other calculation methods may be used to reduce the occurrence of overflow and to produce a more uniform distribution of records in the storage medium.

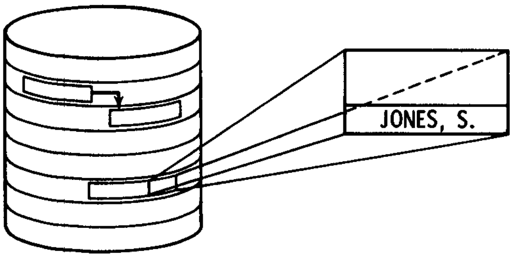


FIG. 4. Random organization

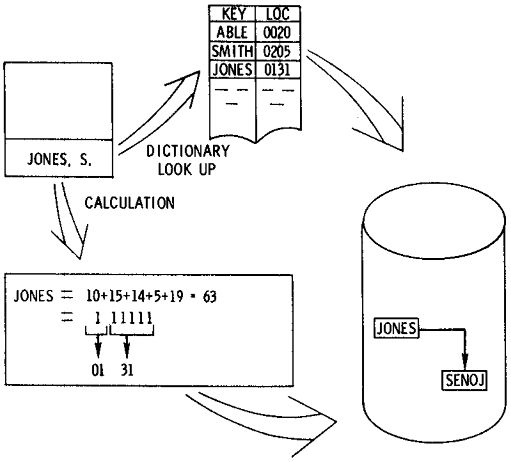


FIG. 5. Dictionary look-up and calculation methods to obtain a record's direct address

Feldman [7] introduces the concept of calculation with multiple keys. In his file structure all records and all elementary data items have names. Both the record name and the elementary data item name are used as keys in the calculation algorithm. To store and retrieve a "name" data item in a "personnel" record the keys NAME and PERSONNEL would be used to calculate the direct address at which the value of the name is stored. This technique has been expanded and generalized in the TRAMP system [21].

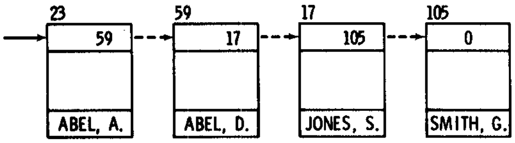


FIG. 6. Simple list structure

The advantage of using random organization is that any record can be retrieved by a single access. Other records in the file do not have to be accessed and their keys examined, as in the sequential method, when trying to retrieve a particular record. Individual records can be stored, retrieved, and updated without affecting other records on the storage media. However, all records are generally of a uniform length. This requirement is sometimes imposed by the way hardware addresses reference the storage media and also permits easier handling of overflow records.

Random organizations are used most often in conjunction with direct access storage media, and the direct address of a record corresponds to a hardware address on the storage media. Although random organization does allow for rapid access of a particular record with a known key, it is not suited for rapidly accessing a number of records. This limitation is imposed by the time taken by the hardware access mechanism to locate a record. Other difficulties are (1) handling the overflow problem when calculation is used for obtaining the direct address, and (2) manipulating large and unwieldy dictionaries when dictionary look-up is used.

LIST ORGANIZATION

The use of pointers to handle the overflow problem in a random organization suggests a third form of data organization, called the list organization. There are three main types of list organization: simple list, inverted list, and ring. The basic concept of a list is that pointers are used to divorce the logical organization from the physical organization. In a sequential organization the next logical record is also the next physical record. However, by including with each record a pointer to the next logical record, the logical and physical arrangement can be completely different. A *pointer* may be anything which will allow the accessing mechanism to locate a record. If the records are stored on direct access media, it is the direct address of the record; if the records are in core memory it is the core address of the record.

Simple List Structure

Figure 6 is an example of a simple list in which four records are stored in a logically sequential order. They are not, however, in the same physical order. The first record is at location 23, the second record at location 59, and the third and fourth records at locations 17 and 105, respectively. The logically sequential organization is obtained by using pointers. Initially, there is a pointer to the first record The first record contains a pointer, 59, to the second record, which contains a pointer, 17, to the third record, and so forth through the list. The last record in the list is designated by a special symbol, in this case, 0.

Since any data item in a record may be treated as a key, many lists can pass through a single record. Figure 7 shows several records. Each record contains four elementary data items: name, city, occupation, age. Each record is also a member of two lists. In the top list the records are logically related by the name of the city, and in the bottom list they are logically related by the name of the person. In each of these lists, the ordering is alphabetical. By allowing records to be on multiple lists, duplication is avoided.

If a record is to be updated, it can be updated in both lists automatically by updating it in one list. Records can be placed anywhere within a list by changing the pointer of the record preceding the new record and inserting a new pointer in the record just inserted. For example, if a record were to be inserted between the "Jones"

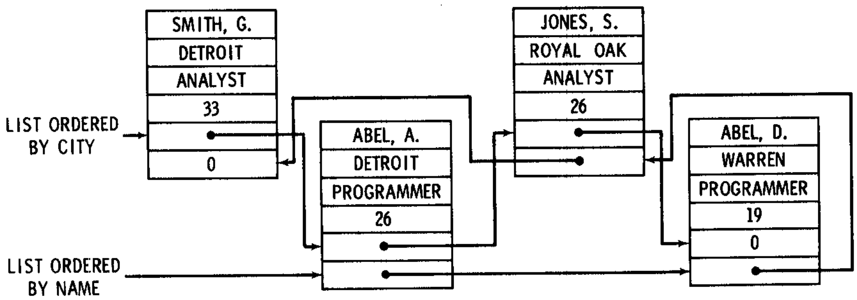


FIG. 7. Two lists passing through a record

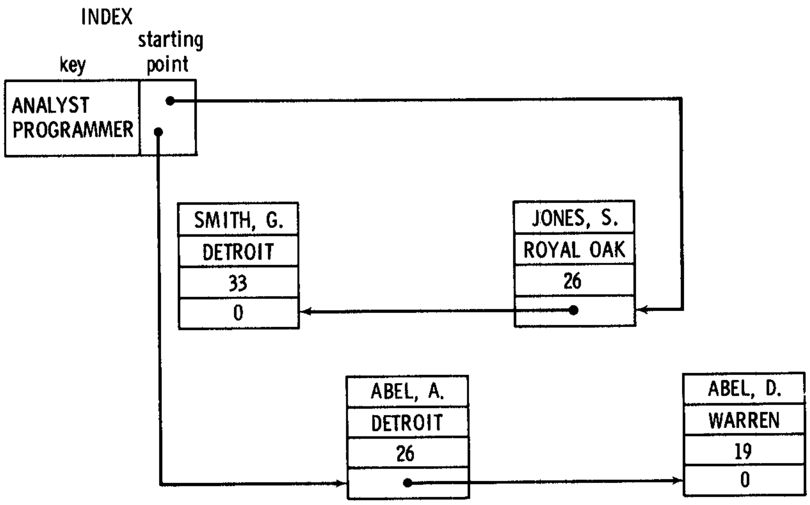
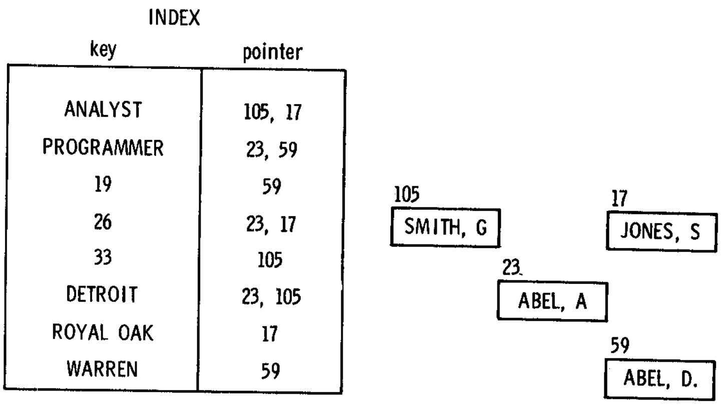


FIG. 8. Partially inverted list

and the "Smith" records in Figure 6, the pointer in the Jones record would be set to point to the new record, and the new record would be given the pointer value 105, which then points to the Smith record. The removal of a record from a list requires changing the pointer in the preceding record to point to the record following the one being deleted. When a record participates in a single list, deletion is not difficult, because the record is initially accessed from the preceding record. However, if a record is a member of several lists, it becomes more difficult to find the preceding records of all the lists, and extra pointers must be stored so that the preceding as well as the next record of the list may be found.

With large files the lists themselves tend to be long, and extended searches may be required if the list length is not controlled in some way. Where the list length is not restricted, there is only one starting point for each list. Where the list length is restricted, the effect is to create sublists, each of which has its own starting point. This reduces the search at the expense of maintaining an expanded index of starting points. Figure 8 shows a partially inverted file in which the occupation datum of each record has been put into an index. Each entry in the index has a starting point to the list of records having the same value as the key in the index; Jones and Smith are "analysts" and Abel and Abel are "programmers." Observe that the



INDEX

key	pointer
ANALYST	105, 17
PROGRAMMER	23, 59
19	59
26	23, 17
33	105
DETROIT	23, 105
ROYAL OAK	17
WARREN	59

FIG. 9. Inverted list structure

occupation data item has been removed from each record and resides in one place in the index at the expense of adding a pointer to each record. If the pointer is smaller than the replaced data field, the record occupies less space.

Inverted List Structure

When the restriction on list length is taken to its ultimate conclusion, the list length is restricted to 1, and each key appears in the index. The index thus points directly to the record sought and no pointers are required. The list has now become inverted. Figure 9 is an example of an inverted list. Here all pointers are kept with the index. If, for example, all the records having the key "analyst" are desired, the word "analyst" is found in the index and the pointers 105 and 17 are obtained. These point to the actual data records shown on the right side of the figure. By keeping the pointers with the index, a request for information can be obtained by first manipulating the pointers in the index. Once the correct data pointers are found, they are used to retrieve the data from the file. For example, let us assume that a data request for information on all the analysts who live in Royal Oak is received. The "analyst" entry in the index shows that records having pointers 105 and 17 satisfy this requirement. An examination of the "Royal Oak" entry in the index shows that the record having a pointer of value 17 satisfies this requirement. By comparing the entries under these two indices, it is concluded that only the record having the pointer 17 satisfies the initial request. An access of the file is then made and the record having the name "Jones" is retrieved. By keeping the pointers with the index it is possible to process a request without having to go to the file. For instance, let us process the request to retrieve the names of the programmers living in Royal Oak. The "programmer" entry in the index shows that pointers 23 and 59 are relevant, while the "Royal Oak" index entry shows that the pointer 17 is relevant. Comparing these two lists of pointers leads to the conclusion that there is no information in the file on programmers who live in Royal Oak and the access to the file never takes place.

The inverted list organization makes available every data item as a key. Such an organization requires a dictionary of all data values in the system containing the addresses of all locations where those values occur. The dictionary can be as large as or larger than the data itself. The virtue of such a system is that it allows access to all data with equal ease. Consequently, it is more suitable for situations where the data retrieval requirements are less predictable, for example, decision-making and planning functions, than it is for specific processing functions. Although the inverted list approach lends itself to easy retrieval, storing and updating data is more difficult, because of the maintenance of the large dictionaries. It is best to combine an inverted list organization with either a sequential or random organization. In this way records are inverted on only one or two keys rather than all the keys. Dictionaries become smaller and it is still possible to access all records in the file.

Ring Structure

Rings are an extension of a list organization. In Figures 6, 7, and 8 it is noted that the end of the list is designated by a 0 value for the last pointer. In a ring the last record points back to the first record of the ring and a special symbol is kept in the first record to designate that this is the first, or the starting point of the ring, as shown in Figure 10. It is possible to follow the ring to find any record, such as the preceding record, the next record, or the starting record of the ring. Figure 11(a) shows how multiple rings can pass through a record. The primary ring goes from left to right; however, a second ring going from right to left also connects each record. It is therefore possible to pick up the following or preceding record by obtaining and using the appropriate pointer. Extra pointers can also be stored in each record to point to the starting point of the ring. The CORAL structure [18] shown in Figure 11(b) makes use of a ring structure in which each record points to the next record in the ring. All records of the ring except for the starting record have a second pointer, which either points to the starting record or is used as a backward pointer. Back pointers point to the closest previous element with a back pointer, so that they form a complete ring. Since rings are most often processed in the forward direction, alternation of the less useful back pointers and starting record pointers retains the advantages of both of these less useful pointers in half the space.

The ring structure proves to be very powerful, as it provides a facility to re-

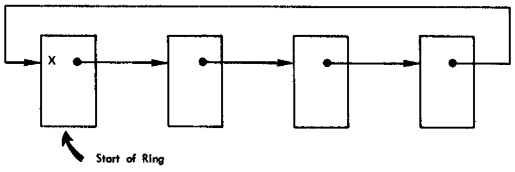
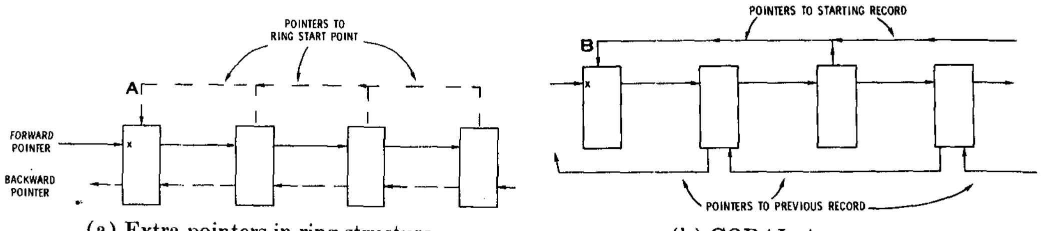


FIG. 10. Ring structure

trieve and process all the records in any one ring while branching off at any or each of the records to retrieve and process other records which are logically related. Such records would also be stored in a ring structure and in turn permit the same facility; this nesting is carried through to any level required by the logical relationships in the data. These more complicated ring organizations form what is called a *hierarchical storage organization.*

The need for a hierarchical storage can be illustrated by examining the data structure in Figure 9, where the records are accessed after processing the pointers in an index. Assume that a record is retrieved by the key "analyst." Once the record is found, however, it is not possible to obtain anything from the record except the name. The age of the individual and the city in which he lives are kept in the index, and the record keeps no pointers from it back to the index entries containing this other vital information. The file can be restructured into a hierarchical organization, as shown in Figure 12, so that related information can be obtained. The new structure is used in the following manner. Assume that information on all records having the key "analyst" is desired. The "occupation" ring is first searched until the record having the key "analyst" is found. This record is the starting point of another ring connecting all the records of the individuals who are analysts, in this case Jones and Smith. Once the Jones and Smith records have been retrieved, other information can be obtained by following the C ring to its starting point and retrieving the name of the city from the starting record. By following the C ring


POINTERS TO RING START POINT
A
FORWARD POINTER
BACKWARD POINTER
x
POINTERS TO STARTING RECORD
B
x
POINTERS TO PREVIOUS RECORD

(a) Extra pointers in ring structure (b) CORAL ring structure

Fig. 11

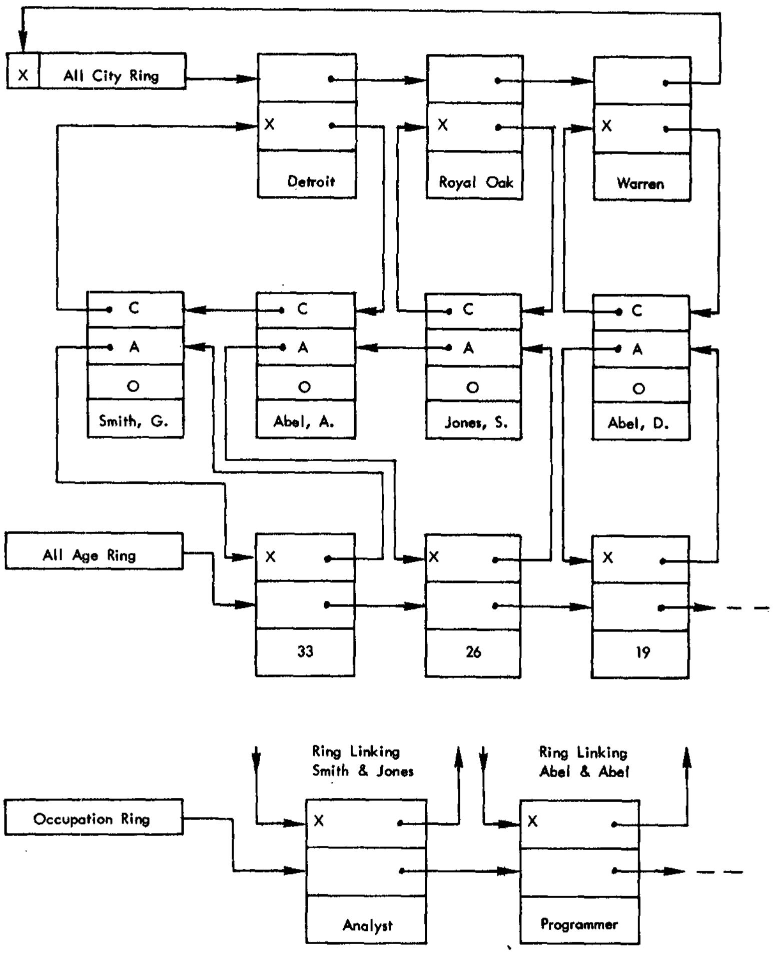


Fig. 12. Hierarchical structure

passing through Jones, it is determined that the city in which Jones lives is Royal Oak. The ages can be obtained in a similar manner. This form of hierarchical structure permits the storage and retrieval mechanisms of a data management system to start with any record in the file and move up or down the hierarchy.

It is easier to insert records into and remove records from rings at arbitrary points than it is with other list structures. This is possible because the adjacent records whose pointers are modified during the operation can be found by traversing the ring until the desired record is found. In contrast, a simple list must always be examined starting at the first record, since given an arbitrary point in the middle of a list it is not possible to find the preceding record.

The prime disadvantage of all the list organizational methods is the space overhead caused by the pointers. In inverted lists, pointers reduce the number of occurrences of a key and may result in the saving of space. The cost of the pointer must be weighed against the advantages of storage and retrieval when contemplating the use of list organizations.

Media Space Management

When new records are stored in a list or a randomly organized file, space that is not being used by some other record must be found for the new records. When records are removed from a file the space they occupy can be made available for reuse. In both cases, a space management mechanism is invoked for keeping track of all the space in the storage medium. Several factors govern the operation of this mechanism:

(1) The new record should be located so that minimum access time is needed to retrieve it. For example, overflow records in a random organization on a disk should be placed so that the disk arms will not have to move to locate the next overflow record. Records in a multilist file (which is described below) should be so located in pages that the crossing of page boundaries while searching a list is minimized.

(2) If records are of variable size, space for them should be allocated in such a way that the storage medium won't be fragmented into small pieces of free space that are unusable for future space requests.

(3) The compacting of pieces of free space into larger blocks should take place at intervals frequent enough so that large blocks of space will be available for reuse (thus helping to reduce the problem of fragmentation). At the same time, it should be recognized that there is an overhead associated with compacting space, which should be kept to a minimum.

Meeting these three criteria can present a nasty problem when dealing with random and list organizations.

A number of strategies have evolved to deal with space management. LISP [12, 20] sequentially allocates all the space on the storage medium. When none is left a "garbage collection" routine looks around and claims all discarded space for reallocation. Other techniques [10, 14, 18, 19, 23] dynamically reclaim space occupied by deleted records. If properly handled, space management is not a serious problem. However, it does present a small overhead when storing and deleting records in both random and list organizations.

EXAMINATION OF COMPLEX STRUCTURES

Now that sequential, random, and list methods of data organization have been considered, let us look at some of the more exotic data management techniques shown in Figure 1 and see how they can be implemented by the use of sequential, random, and list data structures.

Multilist File

Figure 13 depicts the organization of a multilist file. In this file a sequential index contains the key values by which records are indexed. Associated with each key value is a pointer to the list of records having that key value. Records are stored in fixed-size blocks called *cells* or *pages*. They are addressed by a page number and a record number within the page: the record identifier 0,4 references the fourth record in page 0, and the record identifier 1,3 references the third record in page 1.

The grouping of records into pages is an important concept used in storing list structures on direct access media. Rather than using a record's direct address as a value in a pointer, a page number and record number are specified. The page number is

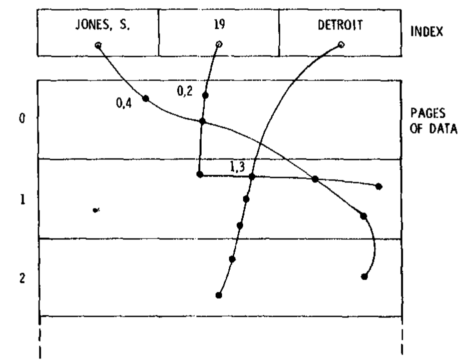


FIG. 13. Multilist file

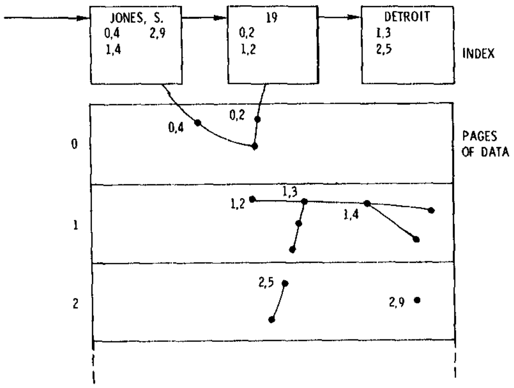


FIG. 14. Cellular multilist file

used as a direct address to retrieve the page and the record number is used to locate the record within the page. In this way a group of records can be retrieved at one time and the access time per record is reduced. Saving is achieved only if a number of records in the page are desired—it is inefficient to retrieve an entire page to process only one record. In using a multilist file it is therefore preferable to store related records within a common page. To eliminate unnecessary accessing of pages, it may be desirable to limit list length so that all the records on a list fit in a single page.

In summary, a multilist file consists of a sequentially organized index with each index entry pointing to a list of records. Records in the file are blocked into logical units called pages. An entire page is accessed when a record is stored, updated, or retrieved.

Cellular Multilist File

When working with the multilist file quite often several pages are accessed to retrieve a record. A data organization that limits lists so that they do not cross page boundaries is shown in Figure 14. This is called a cellular multilist because each list is contained in one and only one cell (page). This structure is processed in a manner similar to that used for a fully inverted list: the values of the indices are retrieved and only those pages containing record lists satisfying all the search criteria are retrieved. In Figure 14 the index is organized as a list, while in Figure 13 it is organized in a sequential fashion. This is only to demonstrate that either type of index organization is possible. In the cellular multilist the list length is restricted so as to fit within a page; if the list length is limited to 1, the file becomes inverted, as shown in Figure 15.

While the multilist file is simpler to program and update, the retrieval time is longer than that encountered with the inverted list or the cellular multilist. The longer times are caused by excessive page retrievals encountered while following a list from page to page in pursuit of a particular record.

Indexed Sequential File

An indexed sequential file organization is shown in Figure 16. This file has the property that records can be accessed either by the use of indices or in a sequential fashion. In the top part of the figure the indices are shown as being sequentially contiguous. Each index contains the address of a record in the file. If a record is to be retrieved by its index, the indices are searched until the desired key is found, and the pointer from that index to the record is used as a direct address to retrieve the record. The records are also stored on the storage media in a sequential fashion, permitting the records to be accessed in a sequential order.

Indices in an indexed sequential file are often oriented to the characteristics of the storage media. In some systems there is a cylinder index as well as a track index for each data item stored in the file [9].

Tree Organization

A tree data organization is one in which several levels of indices are used. The index on the first level points to a collection of indices on a second level. One of these second-level indices is used to find a collection of indices on a third level, and so forth, until the actual data value has been found at the bottom of one particular branch on this tree. Figure 17 is an example of a multilevel dictionary having a tree organization. In this figure the indices on the various levels are organized in lists. If the record identified by "ACM.DETROIT.JONES" is desired, the following actions take place. The first list is searched until a record having the key value "ACM" is found. From this record a pointer to the second-level index list is obtained. The records on the second-level list are sequentially examined until the key "DETROIT" is found, and the pointer in the "DETROIT" record is used to access the index list on the third level. The keys of the records on this level are searched until the record having the key "JONES" is found. In this record is the direct address of the record identified by "ACM.DETROIT.JONES". Of course, the indices can be organized sequentially as well as in a list.

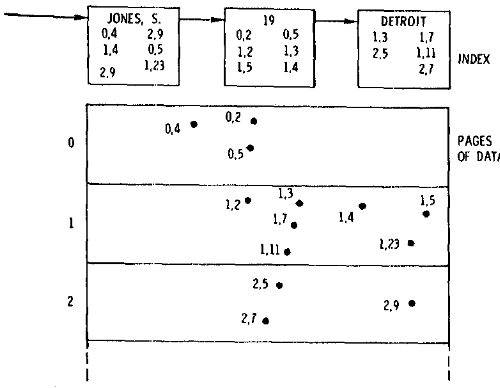


FIG. 15. Inverted file

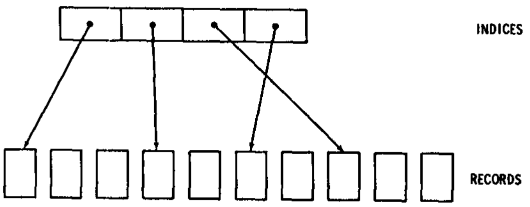


FIG. 16. Indexed sequential file

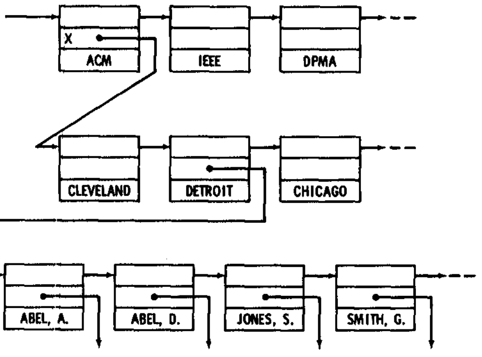
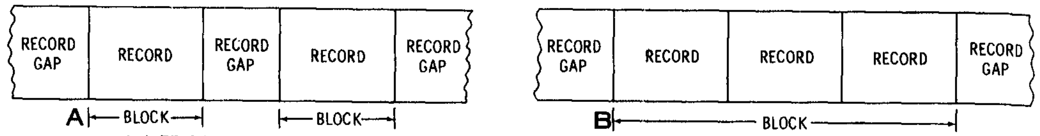


FIG. 17. Tree structure

The tree organization is conveniently used in maintaining large dictionaries. Portions of the dictionary can be added, deleted, or changed without disturbing the rest of the organization.

STORAGE MEDIA

It is important to clearly distinguish between storage devices and storage media. *Storage devices* are the hardware units, such as tape units, disk drives, and data cell drives, that participate in the storing and retrieving of data. *Storage media* are the material on which data is stored; magnetic and paper tapes, disk packs, film strips in RACE units, punch cards, and magnetic cores are examples of storage media. Storage media may be capable of being removed from a storage device, such as a tape reel from a tape drive, or may be permanently affixed to the storage device, such as drums or magnetic cores. With this in mind, let us now examine the types of storage media found in computer systems.



(a) Unblocked records (b) Blocked records

FIG. 18

Sequential access media are those whose access mechanism can locate only the next or prior record, i.e. the one physically adjacent to the current location of the access mechanism. Examples are magnetic tape or punched tape, which are moved in a sequential fashion past the reading or writing mechanism. When the tape is at a given location, the next record of data is located by moving the tape one position. Some devices permit the reading of the media in either direction. In this case either the prior (immediately preceding) or the next (immediately following) record can be accessed. Sequential access devices excel in reading or writing the next record on the medium being used in the device and are used most often with sequential files.

Direct access media are those in which a hardware address is used to position the access mechanism. This address is used when data is written onto the media and again when it is retrieved. Core memory is an obvious example of a direct access medium, since every location in the memory has an address. Magnetic disks and drums are also considered to be direct access devices, since each part of the recording surface can be addressed. However, once the access mechanism has positioned itself at an address, data is read or written in a sequential manner onto a track of recording medium passing beneath the access mechanism. Direct access devices excel in locating a particular data record having a known address. They are also used to retrieve a page of records in the multilist organization, where the page number corresponds to the address of the page on the direct access medium

Figure 18(a) illustrates the physical appearance of data which has been recorded onto any storage medium except magnetic core. The record gaps are placed on the storage medium by the hardware. They are used to separate the areas on the tracks and to permit certain equipment functions to take place as the gap moves past the access mechanism of the device. Within each block of recorded data is one record of data, which may be of any length. It is often advantageous to group several data records into one block, as shown in Figure 18(b). Blocking improves the speed with which records may be read or written because the overhead involved in locating a block is distributed over the access and use of all records within the block. It also saves record gap space, since the gaps between records within a block are eliminated.

LANGUAGE FACILITIES FOR PROGRAMMING DATA MANAGEMENT SYSTEMS

There are many programming languages exclusively aimed at handling a particular type of data organization. The common programming languages, such as PL/I, FORTRAN, and COBOL, all have mechanisms for dealing with sequentially organized files. By means of the READ statement the next record is retrieved, and the WRITE statement stores a record in the next location in a sequentially organized file. In FORTRAN the file can be backspaced by the BACKSPACE statement and returned to its starting point by the REWIND statement. PL/I and COBOL do not have a backspace statement; by means of the CLOSE statement the file is returned to its starting point.

List languages, such as LISP[12] and IPL-V[14], were initially developed for experimentation in the more mathematical and scientific areas of computing, such as heuristic processes, general problem-solvers, self-organizing machines, and automata theory. They were not aimed at data management, and while they handle lists very well, they are cumbersome in the handling of the underlying data. Primitive machine-level-type operations are used for arithmetic and it is not easy to manipulate and test data in expressions. In some implementations the programmer must be concerned with moving the lists to and from high speed memory if the size of the lists is greater than the memory. PL/I provides a pointer capability, enabling the programmer to dynamically allocate space for a record and to store pointers between records, thus creating his own list and ring structures. Records are stored in an area (similar to a page), and areas can be read from and written on external storage media using the PL/I input/output facilities. Although PL/I requires the maintenance of pointers and the performance of all input/output by the programmer, it is very useful for constructing new types of list organizations. SLIP [23] is a list processing language for handling two-way ring structures and exists as a set of subroutines callable from FORTRAN.

Perhaps the least often provided feature of all programming languages is the ability to handle random data organizations. The COBOL language provides some rudimentary statements, which are not fully implemented by any manufacturer. PL/I has a class of statements to READ, WRITE, and REWRITE records on direct access media. However, these statements are limited to work within the provisions of the OS/360 job control language and do not let the programmer have access to all the capabilities of the physical hardware.

There are two languages which permit the programmer to organize his data in a ring structure. APL [6] consists of six statements which have been added to the PL/I language, and IDS [1] has a similar number of statements, which have been added to the COBOL language. These languages conduct ring operations on the same high level as array operations are conducted in FORTRAN and PL/I. They permit the programmer to define the ring relationships between data records, to create new data records within the data set, to insert data records on rings, and to search rings to find data records having particular data values, without having to become involved in the mechanics of pointer manipulation. After the records have been retrieved, the data values may be modified or used in any arithmetic calculation. Complementary statements for removing records from rings and deleting records from the file are also provided. Both these systems work in the paging environment handled automatically by the system; so the data base and the associated ring structures can be much larger than the high speed memory itself. Since these languages are supersets of PL/I and COBOL, they also suffer from the restrictions of their host languages. The concept, however, is one that could be added to any language.

There is no one language that is best suited for programming all data management systems. Data management systems have been programmed in assembly language and JOVIAL, as well as the other languages described above. Only a study of the language facilities, the types of hardware to be used, the types of input queries and the way they will be submitted to the system, the capabilities of the storage media, and the compiler efficiencies will dictate which language to use for programming a particular data management system.

SUMMARY

There has been a brief discussion of the purposes of data management systems. A description of the primitive data management techniques—sequential, random, and list—has been given. Sequential data organizations have matching physical and logical storage requirements, permitting rapid access of the next logical record in the file. Random data organizations are characterized by rapid access of data based

on a key indicating the location of the data on the storage media. List organizations divorce logical relationships from the physical storage mechanisms by means of pointers, permitting data records to efficiently take part in multiple-relationship files.

The mechanics of how more complicated structures can be built using the primitive data organizations has been considered. Rings, inverted lists, trees, hierarchical, and other types of data structures have been explained. Storage media were discussed and it was observed that sequential access media are suited for sequential organization, while direct access media are used for random organizations and list organizations. Programming languages to deal with data management systems were examined and it was concluded that no one language can handle the universe of problems satisfactorily.

Because of the large number of references available on the subject, only a few of the more prominent ones have been cited. Reference 3 contains an excellent bibliography on file organization with 97 entries, which can be consulted for further information. Knuth[11] presents an advanced text for the serious student containing exercises and solutions on file organization and space management techniques. Lefkovitz [24] describes several types of file organizations, including the multilist organizations, along with dictionary designs, storage and timing formulations.

ACKNOWLEDGMENTS

The author would like to acknowledge the help given by John T. Murray, who listened to the initial oral presentation of this paper and suggested how it could be made into a tutorial. Michael J. Mithen, who produced the figures in this paper, should also be commended. Special credit is due to Tax A. Metaxides, whose contributions to the COBOL Data Base Task Group Report [4] on extensions to the COBOL language were used to help formulate the initial paper.

REFERENCES

1. BACHMAN, C. W., AND WILLIAMS, S. B. A general-purpose programming system for random access memories. Proc. AFIPS 1964 Fall Joint Comput. Conf., Vol. 26, Pt. 1, Spartan Books, Washington, D.C., pp. 411–422.
2. BLEIER, R. E. Treating hierarchical data structures in the SDC time-shared data management system (TDMS). Proc. 1967 ACM Nat. Conf., Thompson Book Co., Washington, D.C., pp. 41–49.
3. CLIMENSON, W. D. File organization and search techniques. In *Annual Review of Information Science and Technology, Vol. 1,* C. Cuadra (Ed.), Wiley, New York, 1966, pp. 107–135.
4. Data Base Report. Doc. No. PB177682, Clearinghouse, U. S. Dep. of Commerce, Springfield, VA, 1968.
5. DIXON, P. J., AND SABLE, J. DM-1—a generalized data management system. Proc. AFIPS 1967 Spring Joint Comput. Conf., Vol. 30, Thompson Book Co., Washington, D.C., pp. 185–198.
6. DODD, G. G. APL—a language for associative data handling in PL/I. Proc. AFIPS 1966 Fall Joint Comput. Conf., Vol. 29, Spartan Books, Washington, D.C., pp. 677–684.
7. FELDMAN, J. A. Aspects of associative processing. Rep.-TN-1965-13, MIT Lincoln Laboratories, Lexington, Mass., Apr. 1965.
8. HOBBS, L. C. Review and survey of mass memories. Proc. AFIPS 1963 Fall Joint Comput. Conf., Vol. 24, Spartan Books, Washington, D.C., pp. 295–310.
9. Introduction to IBM/360 direct access storage devices and organization methods. C20-1649, IBM Corp., White Plains, N.Y., 1966.
10. KNOWLTON, K. A fast storage allocator. *Comm. ACM 8,* 10 (Oct. 1965), 623–625.
11. KNUTH, D. E. *The Art of Computer Programming, Vol. 1.* Addison-Wesley, Reading, Mass., 1968.
12. MCCARTHY, J., et al. LISP 1.5 Programmers Manual. MIT Press, Cambridge, Mass., 1962.
13. MORRIS, R. Scatter storage techniques. *Comm. ACM 11,* 1 (Jan. 1968), 38–44.
14. NEWELL, A. *Information Processing Language-V Manual.* Prentice-Hall, Englewood Cliffs, N.J., 1961.
15. PETERSON, W. W. Addressing for random-access storage. *IBM J. Res. Develop. 1,* 2 (Apr. 1957), 130–146.
16. POSTLEY, J. A. The Mark IV system. *Datamation 14,* 1 (Jan. 1968), 28–30.
17. Proposed USA Standard COBOL. ACM SICPLAN Notices *2,* 4 (Apr. 1967).

18. Roberts, L. G. Graphical communication and control languages. In *Second Congress on Information System Sciences*, Spartan Books, Washington, D.C., 1964.

19. Ross, D. T. The AED free store package. *Comm. ACM 10*, 8 (Aug. 1967), 481–492.

20. Schorr, H., and Waite, W. M. An efficient machine-independent procedure for garbage collection in various list structures. *Comm. ACM 10*, 8 (Aug. 1967), 501–506.

21. Ash, W. L., and Sibley, E. H. TRAMP: An interpretive associative processor with deductive capabilities. Proc. 1968 ACM Nat. Conf., P-68, Brandon/Systems Press, Inc., Princeton, N.J., pp. 143–156.

22. Sundeen, D. H. General purpose software. *Datamation 14*, 1 (Jan. 1968), 22–27.

23. Weizenbaum, J. Symmetric list processor. *Comm. ACM 6*, 9 (Sept. 1963), 524–544.

24. Lefkovitz, D. *File Structures for On-Line Systems.* Spartan Books, New York, 1969.