Chapter 8

Access to Information: Hardware and Software Approaches

0 PREVIEW

This chapter examines special-purpose devices and methodologies useful in storing and accessing collections of information items. A number of conventional storage devices such as cards, disks, and tape are introduced first. This leads to the description of more specialized hardware systems that may be used in obtaining rapid access to stored data, including parallel and associative processors and special-purpose "back-end" search processors. The chapter ends with an examination of the main procedures available for accessing and processing data bases consisting of stored texts. The principal file access methods are described, and various text matching procedures are covered which make it possible rapidly to identify portions of text that match a given query statement. Additional advanced hardware developments that may enhance the design of future information retrieval systems are included in Chapter 10.

1 CONVENTIONAL STORAGE DEVICES

This chapter is concerned with the machines and procedures that are especially designed for the processing and retrieval of bibliographic and textual informa-

tion. A historical perspective is presented in the first section followed by a description of specific devices for the manipulation of large data bases and the processing of large quantities of text.

Modern data processing activities as we know them have their origin long ago, starting most likely with the beginnings of formal trade between individuals. When a person is involved in trade, an essential skill is the ability to decide on the worth of the item being traded in relation to the item being received. For example, ancient traders were expected to know or keep track of how many cattle were required in exchange for a husband or wife [1]. As the complexity of trading grew, the need for record keeping and for mechanisms to track and process information led to the invention of various systems of symbols (for example, the Arabic numbers 1, 2, 3, . . . , 10) and of operations which could be performed on those symbols (such as addition or comparison for equality). The complexity and the speed of these processing operations has increased over time to meet growing demands. Processing aids, such as the abacus, were soon developed that furnish greater processing speed and increase human abilities to master complex problems.

Among the most noticeable advances in information processing were the greatly enhanced capabilities for performing numerical computations [2]. As an example, the analytical engine developed by Babbage in 1834 was specifically designed to aid the military purposes of Great Britain by calculating the trajectories of cannons and other firearms [3]. The modern computing era began during the Second World War when automatic computing equipment was used in Great Britain for the decoding of encrypted messages, and in the United States for the calculation of ballistic firing tables. The interest in the automatic storage and retrieval of bibliographic and textual information existed from the beginning because of the large available information collections, and the difficulties of dealing with textual materials by conventional methods.

A Punched Cards

A major innovation of concern in information retrieval occurred with the use by Herman Hollerith of punched card techniques to process the data collected for the 1890 census. Hollerith proposed using one or more paper cards for each individual responding to the census and encoding all responses by punching holes into these cards at precise locations. He then designed mechanical devices to "look at" prespecified locations on the cards to determine if a hole had been punched. The number of holes could then be automatically tabulated. Hollerith used a linear scan of the set of cards which looked at the same location in every record to determine whether a specific value was present. Hollerith's idea was enormously successful. In fact, punched cards have remained an important medium for the storage of information through most of the history of computers. Figure 8-1 shows a contemporary punched card of the Hollerith design.

Punched cards can be "read" by commercially available readers at speeds of around 1,000 cards per minute. If a data base of a million characters were stored on cards, this data base would require about 12.5 minutes to read.

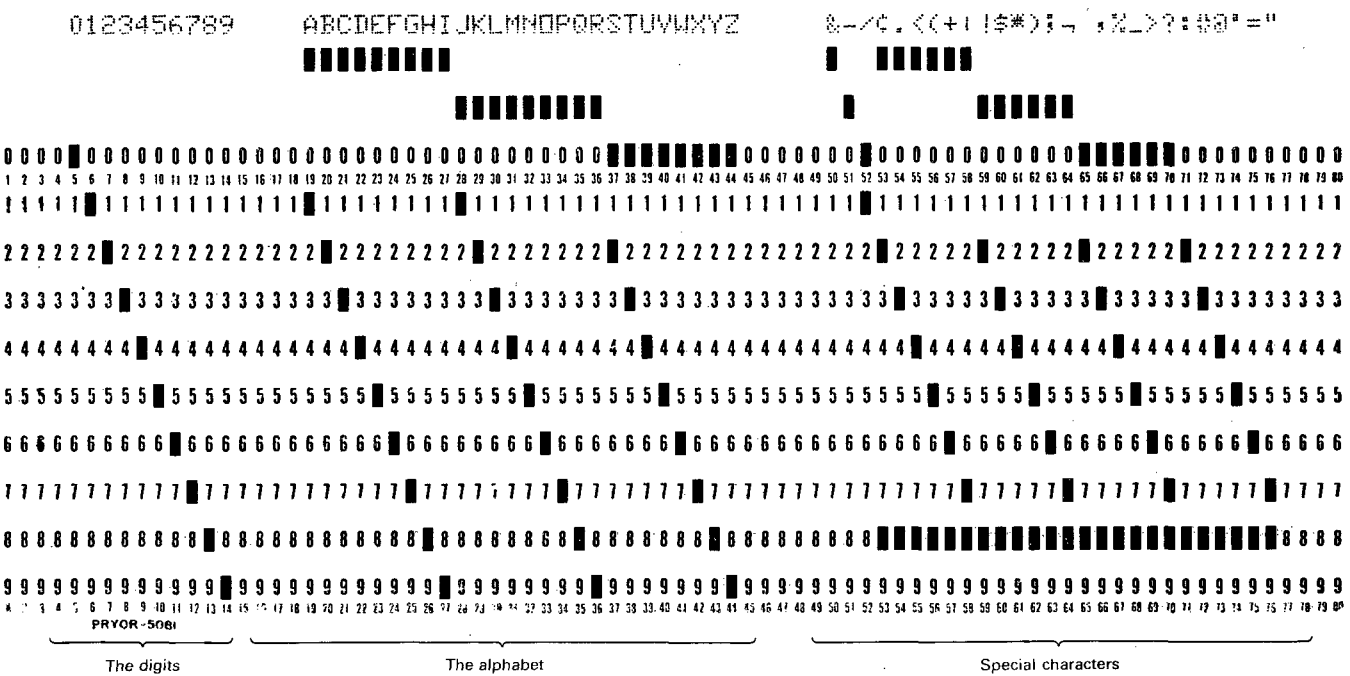


Figure 8-1 Punched (Hollerith) card.

Other storage media have been developed that prove to be more efficient, more reliable, and more cost effective than punched cards. Among these are the magnetic tapes, disks, and drums. Random access storage devices such as core or semiconductor memories have also been developed to provide rapid access to information. These devices are briefly introduced in the remainder of this section.

B Magnetic Tape

The magnetic tape devices used with computers resemble the home tape recorder. Information is stored along the length of the tape on a number of *tracks*. The stored information can then be recovered by "replaying" the tape. Each character of information is represented by a sequence of magnetic spots stored across the width of the tape. All the spots (bits) pertaining to a single character are processed together when a character is added to, or removed from, the tape. The digital characters are commonly packed together along the length of the tape at the rate of 1,600 or 6,250 characters per inch of tape. A typical magnetic tape device is shown in Fig. 8-2.

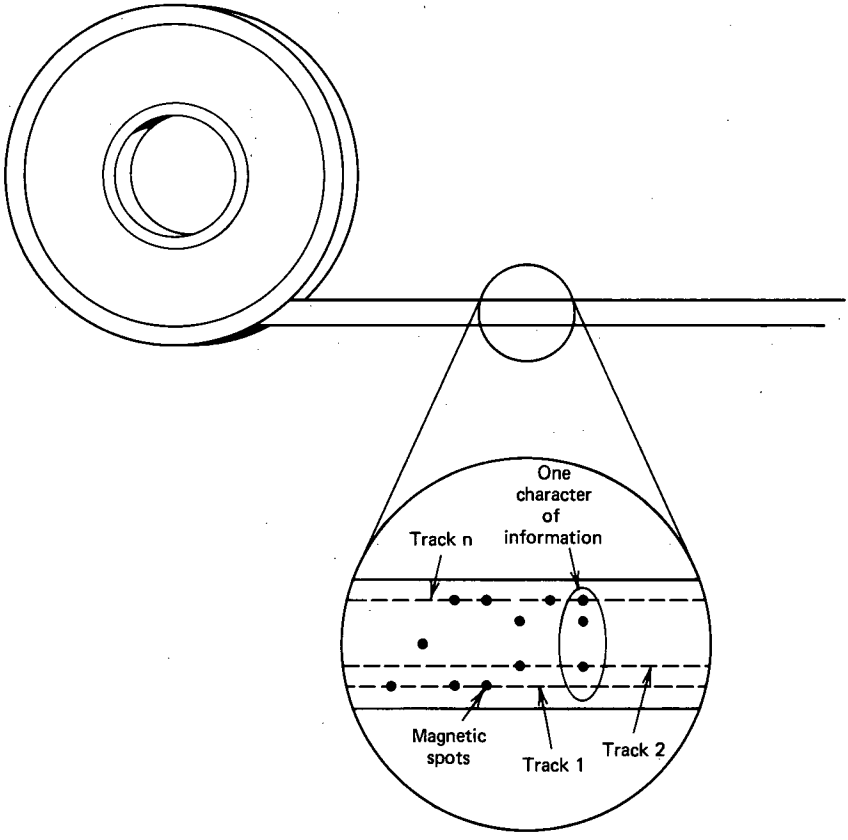


Figure 8-2 Magnetic tape.

Since the information is stored on a magnetic tape in sequential order, it is necessary to scan the tape from the beginning in order to find a specific unit of information. Such a sequential search is potentially very slow. For example, if one million characters are stored in records of 1,000 characters at a density of 1,600 characters per inch using a 3/4-inch gap between records, and if the tape reading speed is 125 inches per second, about 11 seconds are needed to read the tape. Furthermore, before any tape can be read, it is usually mounted on a tape drive by a human operator. This operation may take a minute or more, depending on the circumstances. Tape storage is, however, inexpensive: a tape reel holding 180 million characters at a density of 6,250 characters per inch may cost as little as $11.

C Magnetic Disks

If a magnetic tape appears similar to a home recording tape, then a magnetic disk resembles a phonograph record. Information is stored on a platter capable of retaining a magnetic image. This platter has concentric rings with information sequentially stored on any one ring (track). To find information on a single platter, one needs to know which track to examine and where on that track the information is stored. Information is placed on a magnetic disk by means of a *read-write* head. In many disk devices one read-write head is used per disk face; that is, if information is kept on both sides of the disk, two read-write heads may be used. To locate a specific item of information, the read-write head is moved to the appropriate track and the information is read as it passes in front of the read head. An alternative to the moving read-write head is to have a read-write head for every track of a disk. This *head per track* arrangement eliminates mechanical movement (a slow process).

The magnetic disk is a direct access device because it is not necessary to scan past *all* preceding characters of information to get to a desired unit of information. Rather it is possible to go directly to a desired track on a particular disk. A track may store between 7,000 and 13,000 characters. A standard disk arrangement consists of a group of disks arranged on a single shaft. Current capacities of disk packs are in the range of 300 million characters of information. A typical disk pack is represented in Fig. 8-3.

Information can be read from magnetic disks at the rate of 100,000 characters per second. Thus about 10 seconds is needed to read a data base of one million characters. However, since it is not necessary to examine all records sequentially on the disk, the time required to locate a specific item may be much less than is required for a tape. Disk storage is, unfortunately, much more expensive than an equivalent quantity of tape storage. Disk packs may or may not be removable from the disk drives. Removable packs may cost about $600. Nonremovable packs are less expensive; when they are used, auxiliary tapes may serve for off-line storage.

A recent innovation in disk technology is the small, soft, 5- to 8-inch, *floppy disk*. A floppy disk is easily carried or mailed because it weighs only a few ounces, and it is not harmed by a certain amount of bending or mistreat-

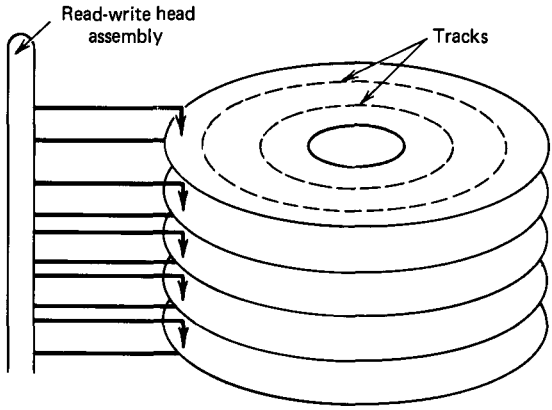


Figure 8-3 Disk pack with access mechanism.

ment. Floppy disks storing about one million characters may cost as little as $1 per disk. They can store anywhere from 1,000 to 3,000 or more characters per track, with access speeds of about 1/2 second. Floppy disks are currently used as the main bulk storage medium in many mini- and microcomputers.

D Random Access Storage Devices

Locating stored information items or transferring information from storage to the computer processing unit tends to be comparatively slow on most computer systems. This is true particularly when mechanical motion is involved, as exemplified by the movement of a tape or the spinning of a disk. Devices that require no mechanical movement for reading or writing tend to be faster. For each such device, the time needed to read or write an item may be the same for all items, no matter where located in storage. For this reason, such devices are known as random access devices.

Random access devices, such as core or metal oxide semiconductor (MOS) memories, tend to be expensive, although prices are declining rapidly. At the present time a storage array for one million characters may cost about $6,000. Compared with disks and tapes, these devices provide only limited storage capacities. That is, the capacity of a disk may be stated in billions of characters, while the core and semiconductor memories may provide several millions of characters only. Random access devices are useful because they are very fast; one can find a specific item very quickly. A typical core storage device is shown in Fig. 8-4. Information is stored in small rings, or cores, that are magnetized in one direction or the other. The cores are arranged on a plane, and several core planes may be stacked one on top of the other. The access to the information stored in a core array is accomplished by sensing the direction of the magnetism measured by the resistance to a prespecified current.

The reading rate for core storage is about five times greater than for disk devices. To read a data base of a million characters requires 2 seconds for core devices compared with the 10 seconds needed for disks. In many systems, the performance of core or MOS memories is enhanced by using still faster and

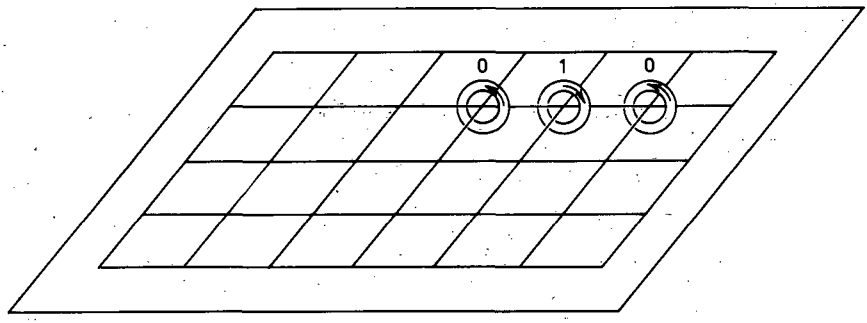


Figure 8-4 One plane of magnetic core storage.

more costly *cache* memories that speed up the transfer between core and the central processing units of the machines.

E Data Cell

In many applications a need exists for a mechanism that stores very large quantities of information in a cost-effective manner. A data cell is, in effect, a combination of magnetic tapes and small magnetic drums. As shown in Fig. 8-5, this device uses cells containing large strips of magnetic tape. Each cell and each

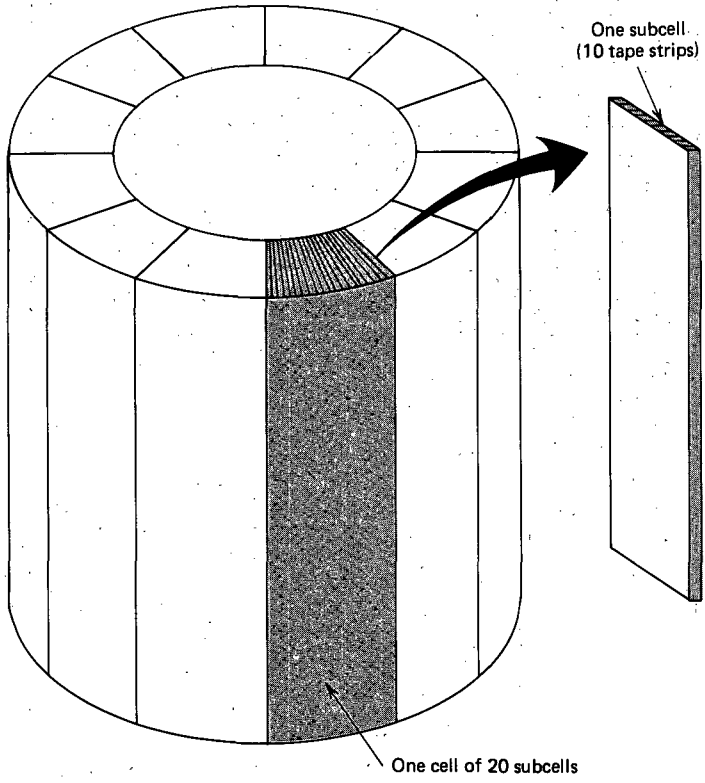


Figure 8-5 Data cell storage device.

strip of tape within a cell is individually addressable. When a tape strip is selected for reading or writing, it is wrapped around a small rotating drum; the information on the tape can then be read as the drum rotates. Billions of characters of information can be stored on a data cell rather inexpensively. Unfortunately, the data cell exhibits relatively slow speed because of the mechanical motion required to (1) find the particular cell, (2) find the particular strip of tape in the cell, and (3) load the tape strip on the drum.

As one would expect, the mechanical motion involved in the operation of the data cell makes this a relatively slow device. However, once a particular tape strip has been located and loaded, the device is capable of operating at speeds corresponding to the drum rotation time. Thus a one million character data base will require 27.5 seconds to read if the tape strip is to be located, loaded, and read.

F Access to Storage

While storage devices are often capable of storing large quantities of information, any manipulation of the stored information will traditionally occur within the central processing unit of the computer. Hence a communications path must be provided between each storage device and the central processing unit of the computer system. Most central processing units have a much greater speed than the storage devices to which they are attached. It is therefore necessary to speed up the information transfer by developing holding areas (buffers) from which the information can be read more rapidly than from the storage devices themselves and by establishing intermediate processing devices (channels) which can open and close appropriate paths, translate the information if necessary, ensure that information is channeled to the proper location, and notify the central processing unit that a particular transfer process is complete.

Figure 8-6 shows the communication processes in a computer system equipped with a variety of communications devices. The multiplexor channel interleaves information from a number of slower devices such as card readers and punches. That is, it takes advantage of the inherent slowness of the devices by selecting messages from a variety of active devices for transmission to the central processor. If a storage device is relatively fast, it is more efficient to select and transmit all the messages from that one device before working on the information from some other device. A selector or burst channel is therefore used with the faster storage devices. The tape and disk control units are used to turn the individual devices on or off and to process the information being transformed.

In many situations a specific item of information will be needed from a data base. More often than not the process required to access this item will interrupt whatever else the computer may be doing at that time. Processing resumes when the item is located and transferred to the central processing unit. If the operations are reasonably efficient and the required data bases are small, such a processing mode may prove acceptable. For instance, many microcomputer

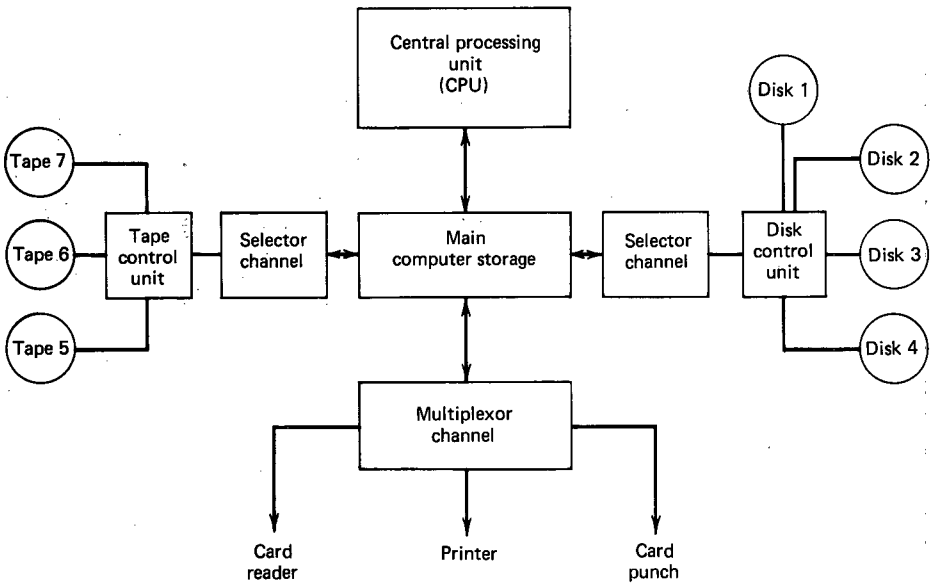


Figure 8-6 Typical computer system with input-output and storage.

systems use tape cassettes for storage; information can be transferred from the cassettes at the rate of 30 characters per second. Accessing a data base of 1,250 characters (about 250 English words or 1 printed page of text) will require 41.66 seconds. Such a time frame may be acceptable to the computer hobbyist, but it may prove unacceptable to a scientist. In many practical situations, stored data bases of a million characters are common. The accessing process described earlier could then stretch to many hours.

The use of buffers and channels will, of course, speed up the process. However, even when additional facilities are provided for increased access speed, the program design and the form used to store the data often determine the speed of the information system. In particular, even if the computer system were capable of performing computations in a few billionths of a second, the true processing time would be much slower if the machine were forced to wait for periodic input from the data base.

A second significant feature of most computer systems is the necessity of knowing the actual storage location of a given information item before that item can be retrieved. That is, in order to find information records on a particular topic the location of the corresponding information records must be known, or else the location must be deducible from available data. If the location information is not immediately available, auxiliary index files, such as the inverted files, are used. It would be easier and less expensive if a system were able directly to find the needed records without concern for location. Presumably there would then be no need for inverted files or other indexes to locate the information. This possibility is discussed below under Associative Processors.

2 HARDWARE ENHANCEMENT OF RETRIEVAL

A Microprocessors and Processing Chips

For many years attempts have been made to design the ultimate information retrieval aid. Such a device may be expected to furnish direct answers to incoming user queries, formulate educated guesses when questions cannot be answered unequivocally, maintain and update personal files and appointment schedules, and perform computations on stored data as well as translations and other transformations on natural language text. The device itself is often pictured as being pocket-sized, requiring little energy to operate, and capable of processing unlimited quantities of information of many kinds.

The basic idea of such a "memory extender" (Memex) is now over three decades old, and information professionals have been striving to design that type of retrieval engine for a long time [4,5]. In its full scope, the memory extender remains outside the realm of practical possibilities for the immediate future. The enormous technological strides which have been made during the last decade at least in the hardware area may, however, lead one to conclude that the time when the full Memex can actually be constructed is fast approaching.

Without a doubt the most significant trend over the past few years has been the impressive decrease in both the size and the cost of existing electronic hardware devices together with a simultaneous increase in processing capability. Complete *microprocessors* are currently implemented on small silicon plates, or *chips*, each chip including tens of thousands of active processing elements (transistors) on a plate measuring substantially less than 1 inch square. Microprocessors are now so inexpensive to manufacture that they are incorporated into many devices intended for home use: they have been used in particular to control automobile systems, microwave ovens, and video tape equipment, and they provide the basic processing capability in many toys and games sold to children and adults.

A microprocessor implementation takes many different forms. In its simplest form, only the chip itself is provided, which includes storage space to save short programs and some data, as well as an arithmetic unit and a sequence control unit to handle the program decoding. If a keyboard is added for manual input and a digital readout register, one obtains a programmable device. Handheld models capable of text and numeric processing that will fit into one's pocket can be acquired for a few hundred dollars at the present time.

When substantially more internal storage capacity is furnished, typically for 32,000 characters of data, and one or two soft (floppy) disks are added for external storage together with keyboard input and cathode ray tube display, the cost of the system jumps to $2,000 to $5,000. Such a system is then usable for text and program editing, messaging, and many tasks needed in a question-answering situation. For full information retrieval usage, an output printer becomes mandatory, as well as a large (hard) disk for bulk storage of several million characters of text. The latter alone may add about $5,000 to the system cost, so that a full information retrieval configuration may cost between $10,000

and $15,000 at current prices [6–9]. The microprocessor characteristics are summarized for three typical systems in Table 8-1.

In principle, theoretical limits exist for some technological advances. However, these limitations are not expected to affect the continued gains in the performance/cost ratio of existing electronic devices for many decades. Furthermore, the conditions under which microcomputer components are currently being developed lead to predictions for increasing parallelism in the process organization and to the existence of "overkill" capacity in the equipment. More specifically, since the individual integrated circuits incorporated in a computing device are expensive to generate initially because of the human skill needed in the design effort but, once designed, are inexpensive to produce in large quantities, it may be economically advantageous to produce components that are rather more sophisticated than may be required for a particular application. The resulting electronic components may then be sufficiently versatile to be used in many different capacities.

Such a deliberate overdesign implies that each chip might be used to carry out input/output manipulations, file management, memory management, interrupt processing, and other information transfer and processing operations. If each function is assigned to a separate hardware component, a parallel processing capability of the kind mentioned later in this chapter results, where several kinds of arithmetic operations are overlapped with data fetch and input/output operations. In an information system environment, separate processors could then be assigned to distinct system operations, and the functions of each processor might be optimized within the context of the total system design [10].

One problem to be faced when new machine architectures are introduced is the need to convert existing applications programs to the new system environment. The conversion problem becomes particularly onerous when large-

Table 8-1 Microprocessor Configurations

Configuration	Typical use	Cost
Handheld calculator	Small programs and	$100–$300
Digital readout	calculations	
Input keyboard		
Several dozen storage registers		
Arithmetic unit		
Automatic sequence control		
Minimal processor	Word processing capability	$2,000–$5,000
16- to 32-bit address	Text editing	
32,000 characters of data storage	Program editing	
2 to 5 microseconds add time	Indexing	
Floppy disks for storage	Cataloging	
Cathode ray tube display		
Full microprocessor configuration	Information retrieval	$10,000–$15,000
Printer added to minimal processor	Inverted file processing	
Hard disk for bulk storage	Output printing	

scale changes in the machine architecture are involved such as those inherent in switching from a largely sequential processing chain to a parallel processing system. When the applications programs are originally written in some high-level programming language, automatic translation programs may be available to perform the required conversion. Programs written to take particular advantage of the existing hardware organization, such as machine language or assembly language programs, must, however, be converted by time-consuming intellectual procedures.

In summary, the processing technology is becoming increasingly miniaturized; at the same time, a trend exists toward special function architectures where separate, sophisticated modules are devoted to specialized functions. This may, in time, lead to a fine tuning of individual system components and to the use of components that precisely fit a particular problem situation. The time is rapidly aproaching when large classes of users will be able to access small, inexpensive computers capable of satisfying sophisticated computational and data processing requirements.

B General Characteristics of Retrieval Hardware

It was suggested earlier that small personalized machines are becoming increasingly common. If these devices are to prove useful in information retrieval, it becomes necessary to provide an interface with the large data bases that exist in various locations. Furthermore, user terminals must be available that are capable of displaying the retrieved information in a user-friendly manner.

The standard available display terminals can currently exhibit 24 lines of text at any one time, exclusive of illustrations. Because of the low resolution of the current displays, it is not possible to view full document pages in a conventional typeset format. Hence the standard display equipment is not well suited for the retrieval of full document texts. High-resolution terminals are coming into use that are capable of displaying full typeset pages with illustrations, allowing 60 lines of readable text. One may expect that good-quality color graphics can eventually be added, allowing displays of document output that the users will actually be happy to see displayed on a screen. At the present time, high-resolution displays capable of controlling the display from local memory inside the terminal are costly—somewhere between $10,000 and $20,000. These costs should decline as the equipment becomes more widespread. The wide availability of small, personal machines with high-resolution display equipment will greatly enhance the usefulness of the existing, automatic retrieval services.

It has been stated that the ideal information retrieval system should have the capacity to store between one billion and one trillion characters. The system should also be of low cost, and it should be capable of finding the stored data in a time period acceptable to a user waiting at a terminal for a response [11].

These represent stringent requirements. For example, if one assumes that

the data base consists of 1 billion characters and if a computer were to process 100,000 characters per second (about as fast as information can be read from a contemporary disk), then a user could wait for 2.7 hours for a response. Thus, either the size of the search files must be drastically reduced or some mechanism must be found for accessing these large quantities of data in a more reasonable time frame.

The standard solution to this problem consists in building auxiliary index files that identify subsections of the main files containing the information of interest to particular users. Alternatively, the use of *special-purpose search and storage* architectures may prove useful to process large information files. These special-purpose devices are in principle capable of processing data independently on a stand-alone basis. More commonly, they are attached to one or more general-purpose host machines. Each user can then access a general-purpose machine—possibly a personal minicomputer or a common large host machine—while delegating the actual search function to the special-purpose device attached to the host machine(s).

The following types of special-purpose search equipment can be distinguished:

1 *Smart peripheral* devices such as disks or drums whose normal function is the storage of information. By adding special processing hardware to the disks, information might be selected, checked, and processed right in the storage device, thereby allowing a good deal of processing to take place in parallel, and avoiding much unnecessary transfer to and from the central processing unit of the general-purpose computer(s).

2 The functions of the smart peripherals can be expanded by adding other processing capabilities, thereby creating special *back-end data base processors* connected to the general-purpose host machine. The general-purpose (host) machine normally controls the data base processor and handles transfers of information between machines; the back-end machine is charged with special-purpose tasks such as information search and special data base computations [12].

3 A number of special-purpose search computers could also be interconnected by operating in a *network mode*. The data stored in a given machine might then be accessed by other machines in the network whenever this may be required by the data base process.

In discussing each special computer configuration, it is convenient to make distinctions based on the number of processing units available and on the type of information search provided. In particular, a search conducted directly in a mass storage device may be distinguished from a search conducted indirectly in a buffer or intermediate storage area [13].

When a special-purpose search computer is used, the search is initiated by the special-purpose machine in response to a request received through a host computer system as shown in Fig. 8-7. The information is examined by the special computer to determine what items should be retrieved. Records that pass the retrievability criteria are then passed on to the host computer system. Sig-

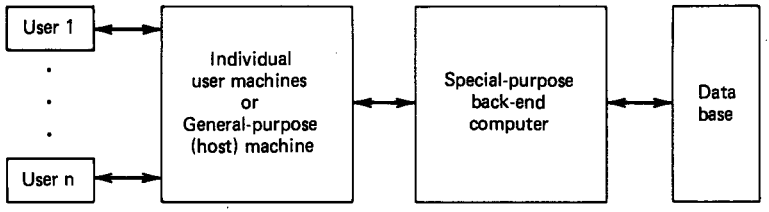


Figure 8-7 Back-end computer use for data base processing.

nificant efficiencies are achieved when the quantity of data that must be scanned in response to each query is large, because the host computer can attend to other work while the search operation takes place on the special-purpose machine.

If the special-purpose processor is integrated directly into a mass storage device, the system is then capable of a direct search. That is, the central processing unit would receive a request which is then passed along to the special mass storage device. This device will search through its data base and return to the central processing unit only those items of information meeting the retrievability criteria.

A number of different special-purpose computer configurations are discussed in the remainder of this section. In each case, the potential importance of the device for information retrieval is briefly mentioned.

C Parallel Processors

Multiple processor devices come in two main designs. The first is based on a number of parallel but independent processing units. Such a configuration may include several central processing units, each unit being capable of a wide range of functions. For instance, two different processing units may each be searching different portions of a collection as illustrated in simplified form in Fig. 8-8. The units may independently be able to determine the potential usefulness of the various documents to a user query. Alternatively, each processor might carry out different operations on the same data base in parallel, such as, for example, word stemming, thesaurus transformation, and query-document similarity calculations. By using several processors, the processing time for a full data base of information items may be decreased according to the number of processors in use [14].

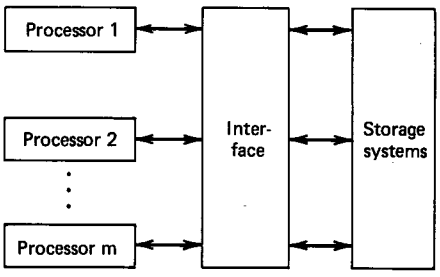


Figure 8-8 Use of separate parallel processors.

Multiple processors of this design are now in limited use in information retrieval environments. In order to increase processing efficiency through the use of multiple, independent processors, it is necessary to identify different procedures that can be carried out simultaneously on different portions of the data base. In information retrieval the complexity of the operations renders the specification of parallel procedures difficult, and it is not obvious that several processors can be kept occupied simultaneously for long periods of time.

Consider a query formulation such as ($TERM_1$ and ($TERM_2$ r $TERM_3$)). Such a query may be processed in three steps as follows:

1 Use the processors in parallel to retrieve the information items associated with $TERM_1$, $TERM_2$, and $TERM_3$. This step takes advantage of the parallel capability.
2 Perform a set union operation for the sets associated with $TERM_2$ and $TERM_3$. This is a sequential process that might be carried out on a single processor.
3 Perform a set intersection operation with the result of step 2 and the set of items retrieved by $TERM_1$. This is another sequential process.

It is clear from this sequence that the advantages of the parallel processing in step 1 may be outweighed by the extra cost and complexity due to the presence of the multiplicity of processors. In general, the number of processors to be used in a multiprocessor system should be determined by a cost-effectiveness measure. That is, one would like to be assured that each additional processing unit will actually serve to increase the efficiency of information processing.

*D Associative Processors

The second design for parallel processing consists in using an associative processor containing several processing units. In this case, however, each processing unit carries out exactly the same process as every other processor at each instant in time. The name associative processor is derived from the associative array (matrix) storage area in which the processing of the information occurs. Each row of this matrix is designed to accept an individual item of information, and the contents of all rows are processed simultaneously. For example, each row may be loaded with a unit of information (such as a different query term) to be compared with a desired value. Each item will then be compared at the same instant with the desired value and all items meeting the desired search criteria will be identified simultaneously.

Figure 8-9 shows the operation of an associative processor searching for the value BLUE among several units of text. The desired value is placed in the comparand register in a specific set of character positions to be searched. The locations to be searched in the array processor are identified by placing 1s in the appropriate positions of a mask register. The positions identified by 0s in the mask register (that is, positions 1 to 4 and 9 to 24 in the example of Fig. 8-9) are ignored in the search. When a comparison is made for a particular term,

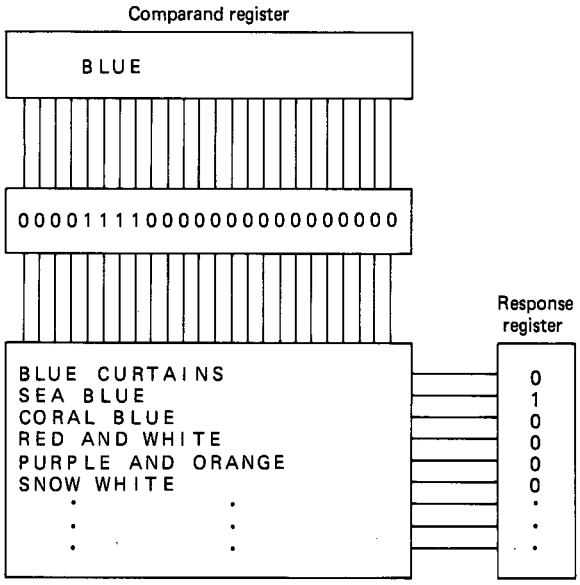


Figure 8-9 Array processor search for term BLUE (specified as positions 5 through 8 of a 24-element row).

such as BLUE, the entire list of data elements is compared simultaneously. Each matching data element is identified in the response register by the appearance of a 1 [15].

Instead of performing parallel search operations, associative processors could also be used to perform arbitrary parallel computations on all elements of the associative array. Thus, in a vector oriented system such as the SMART system, a device that could perform simultaneous calculations between a set of document vectors and a query vector would have great advantages over a system in which the calculations are carried out one document at a time. For example, the calculation of the similarity between a query in the comparand register and each document in the associative array might be performed in the following way:

1 Load the document vectors into the associative store.
2 Load the query vector into the comparand register.
3 Set the mask register so that the character positions correspond to the locations of the weights of the individual query terms.
4 Instruct the associative processor to perform a sum of product operations for the elements of the query vector with each element of all document vectors.

The last step will simultaneously multiply each term weight of the query vector with each term weight of corresponding terms in the document vectors. The resulting products are then summed and placed in the response register. The result corresponds to the numerator of the cosine similarity between the query and each document vector stored in the associative array. Note that

three of the four steps in this process are concerned with setting up the associative processor prior to the actual vector computations.

The advantages of associative processing are numerous. Note specifically that there is no need to be concerned with the location in storage of a desired item until the item has met the desired search criteria. Note also that one may conduct partial searches for terms such as BL*E, where the third character identified by an asterisk is not specified. No preliminary processing is needed since the data are examined by content rather than location. There is therefore no need to prespecify the location of documents by using an index.

Unfortunately, the use of associative processing also entails many disadvantages. A large problem is simply the necessary transfer of information into the associative processor. This is normally carried out as a sequential process which is thus relatively slow. Second, the location in which a given term value appears in the associative processor must be specified precisely. Third, the cost of associative processing equipment is high, and the existing associative memories are small.

An example of an associative array processor is the Staran, developed by Goodyear Aerospace. The original design was intended for the processing of images [16]. This design has already been shown to be useful in data base management environments.

Staran is composed of 32 separate matrices (arrays), each matrix consisting of 256 rows and 256 columns. If one assumes that each character requires 8 columns of matrix storage, then each matrix can store a maximum of 8,192 characters, or 256 rows of 32 characters each. All the matrices are connected to a single comparand and a single mask register. Each matrix does, however, have its own response register. An operation may be conducted on all the matrices simultaneously. Figure 8-10 shows the basic architecture of a system which includes a Staran associative processor.

For information retrieval one sees immediate application of a device such as Staran to provide a parallel search of an inverted index. This can be done by loading all the terms included in the inverted index in the associative array of

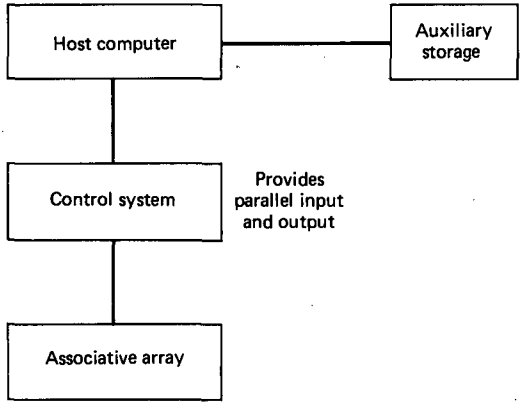


Figure 8-10 Staran organization.

the Staran and passing the query terms through the comparand register. When a match is obtained between a query term and a term stored in the array, the corresponding list of document identifiers may become immediately available. In such an application, the tradeoff is between the high cost of associative processing compared with the cost of the contemporary software methods used to perform inverted file manipulations.

*E Fast Computations Using Array Processors

Many computer applications areas are distinguished chiefly by the need for substantial computational power. For example, in signal processing, large masses of data are received as a continuous stream from external devices, such as radar or satellite equipment, and must be processed and "cleaned up." In such circumstances, the need for fast internal computation becomes overwhelming. To respond to this demand, special processors, known as "array processors" (AP), have been developed that provide very fast arithmetic operations and work in conjunction with a general-purpose computer (the host computer) to which they are attached. The computational APs are not to be confused with the associative processors used mostly for searching that were described previously [17,18].

Array processors are normally implemented as specialized high-speed floating-point machines (that is, they perform computations on floating-point numbers) working in parallel with their host computer. No character manipulation or input-output facilities are normally provided. The computational power of APs is due to two main features:

1 Parallel functional units: instead of including all arithmetic and logical functions of the processor in a single "arithmetic and logical unit" as is done in standard computers, the various functions of the central processing unit are split up into separate functional units that can all operate in parallel; thus in an array processor it is possible to perform an addition in the adder, and also a multiplication in the multiplier, and also a fetch operation to retrieve an item of data from memory, and also an instruction decoding operation, all these operations being carried out in parallel using separate functional units.

2 Pipelined functional units: some units of the array processor are *pipelined* to speed up the processing of a single function, notably addition and multiplication; this means that a given operation is carried out in steps, or stages, in such a way that a given processing unit can effectively carry out several operations at the same time, provided each operation is in a separate stage. A pipelined processing unit, such as a multiplier, consisting of three stages is shown schematically in Fig. 8-11. Three operations (multiplications) can in principle be carried out in the multiplier at the same time: the first multiplication that was started earliest is in stage 3 in the illustration of Fig. 8-11, the second multiplication started one stage later is in stage 2, and the third started most recently is in stage 1. After each time unit the pipelined processor advances by one stage; that is, a new operation can be started in each unit of time if the pipeline is filled, the results of the operation being available three stages later.

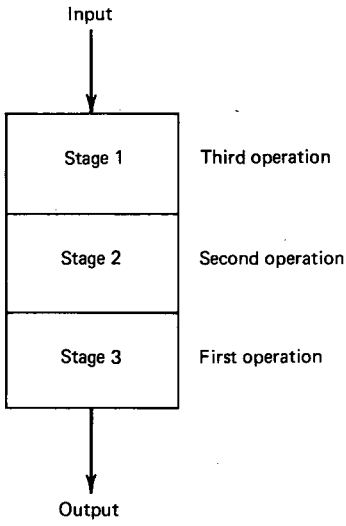


Figure 8-11 Pipelined processing unit.

Because of the limited set of functions provided, the cost of AP processing is low (typically $40 per hour) compared with the cost of a large standard computer (typically $1,000 per hour).

When an array processor is coupled to a general-purpose (host) computer as shown in Fig. 8-12, all input-output, program setup, and data base operations are normally carried out by the host. Computational tasks can, however, be assigned to the AP after transfer by the host of relevant instructions and data into the array processor. The AP then executes its program while the host waits or performs other tasks unrelated to what is going on inside the AP. When the AP finishes its task, a "device interrupt" is sent to the host; the host then reads the results out of the AP, and processing continues.

Whether it pays to use an AP with a host computer depends on whether the savings obtained by executing a routine in the AP outweigh the costs of communicating programs and data between host and AP. The following factors appear important in this connection:

1 The data manipulations should be executable as floating-point arithmetic rather than as address, character, or integer manipulations.

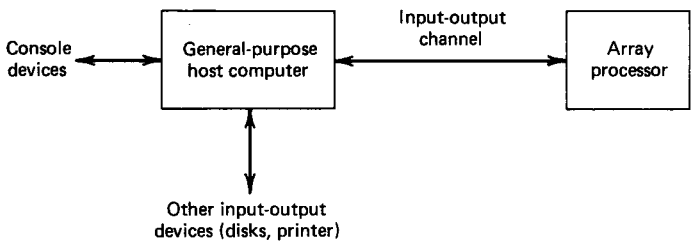


Figure 8-12 Simplified diagram of host-AP system.

2 The application should include long computations to justify the required host overhead and data transfer time.
3 The program to be executed should be small.
4 The data required by the AP should be easy to select.

Information retrieval appears, at first, to furnish a poor application for APs because of the large data base to be processed and the many data transformations as opposed to arithmetic operations to be performed. On the other hand, it is obvious from the material in the preceding chapters that the computational requirements are certainly not negligible in many information retrieval processors. Examples are the computation of similarity coefficients between vectors, the generation of term weighting functions such as the term discrimination and term relevance values, the generation of cluster centroids for clustered document collections, and the computation of recall-precision factors.

Consider, as an example, the computation of the cosine function between DOC_i and $QUERY_j$ defined as

$$COS(DOC_i, QUERY_j) = \frac{\sum_{k=1}^{t} TERM_{ik} \cdot QTERM_{jk}}{\sqrt{\sum_{k=1}^{t} (TERM_{ik})^2 \cdot \sum_{k=1}^{t} (QTERM_{jk})^2}}$$

Looking only at the numerator of the cosine function, the following operations appear to be executable in parallel on an array processor of the type illustrated in Fig. 8-13:

1 The *multiplication* of the vector element $TERM_{ik}$ with the element $QTERM_{jk}$
2 The *addition* of the previous factor $TERM_{i,k-1} \cdot QTERM_{j,k-1}$ to the sum of the previous $k - 2$ products
3 The *memory fetching* operation required to extract terms $TERM_{i,k+1}$ and $QTERM_{j,k+1}$ from memory

In these circumstances, the parallelism of the AP unit appears to be immediately useful. Detailed analyses of the use of parallel computing facilities in information retrieval remain to be carried out [19].

*F Content Addressable Segment Sequential Memory (CASSM)

The content addressable segment sequential memory (CASSM) was designed and built at the University of Florida as a general-purpose device, of benefit to all nonnumeric applications including information retrieval. The basic CASSM design is presented in Fig. 8-14. The CASSM design provides a number of distinct *cells*, each cell consisting of a storage unit and a separate processing unit. During a search operation, the various cells operate in parallel, and all the pre-

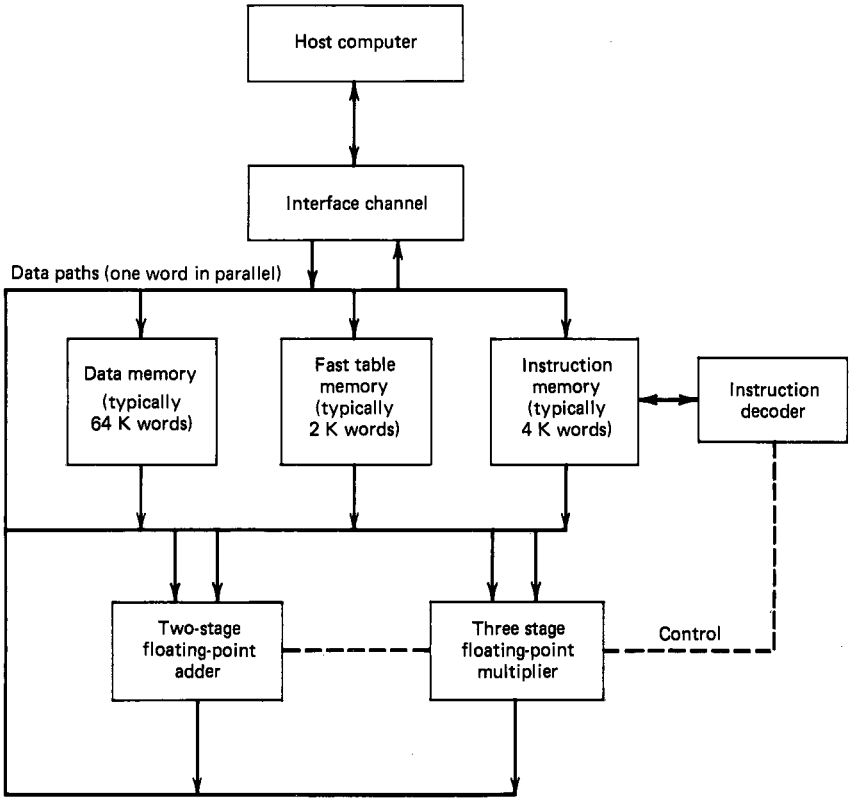


Figure 8-13 Typical floating-point array processor (constants, data, and instructions are kept in separate memories; multiplier and adder are pipelined; integer arithmetic, instruction decoder, adder, and multiplier are separate functional units).

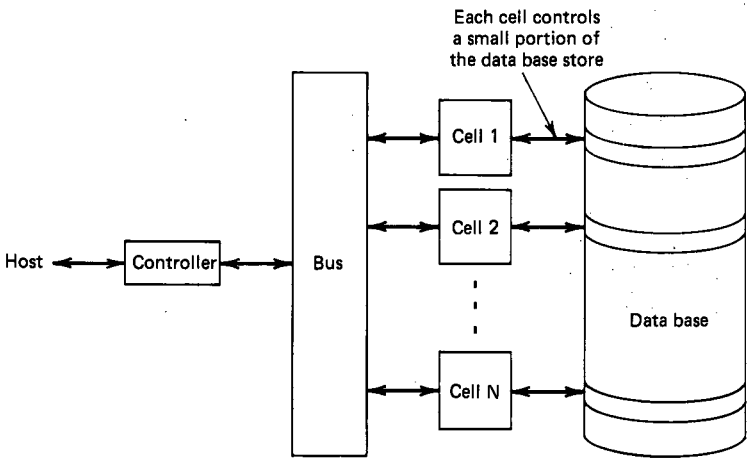


Figure 8-14 CASSM system organization. (*Adapted from reference 13.*)

liminary processing needed to choose the items to be retrieved is done in the CASSM device rather than the host machine. The actual implementation of the CASSM prototype was carried out on a special disk modified to include reading and writing units for each track of the disk [20].

An initial assumption of the CASSM design is that the entire data base will reside within the CASSM device used for data base storage. Thus, as the size of the data base increases, the storage and processing facilities of CASSM must also increase. A given data base may of course be spread over several cells.

CASSM is intended for use with a general-purpose computer acting as a "smart" peripheral device. The general-purpose computer accepts the user's commands or requests, translates these to a set of CASSM instructions, and passes these instructions to CASSM. The special-purpose device then carries out the instructions, all cells executing an instruction simultaneously.

Execution of an instruction on CASSM is broken down into three phases: (1) the instruction phase in which an operation is sent to each cell of CASSM, (2) an execution phase in which the instruction is carried out, and (3) a deactivation phase in which the current operation is removed. Each phase requires one rotation of the disk, but because CASSM is designed to carry out the three operations concurrently, one complete operation actually takes place per revolution.

A search for a specific item of information requires that the item pass a test qualifying it as a potential record of interest. That is, CASSM keeps track of the types of records it stores and checks to see if a record is of the proper type. If the record passes this test, the appropriate field within a record is located and its value examined. Only if all these conditions are satisfied will a record be retrieved.

For example, a data base may contain information on both automobiles and owners of automobiles. A search for the records for silver automobiles will therefore require:

1 A test of the record type to isolate the automobile records
2 Location of the color field for all automobile records
3 Retrieval of the record if the value of the color field is equal to SILVER

Note that this process is conducted for all records in the data base during the single revolution of a disk (about $1/60$ second).

CASSM is an experimental device and as such exists only as a prototype and simulator. The design is limited by the necessity to store the entire data base on the device. It is not clear whether the processing efficiencies could be maintained if a data base were to be loaded into CASSM in small segments in several different loading operations.

*G Relational Associative Processor (RAP)

The relational associative processor (RAP) was developed at the University of Toronto as a device for the processing of information specified by a relational data base model (see Chapter 9). RAP, like CASSM, is a smart peripheral de-

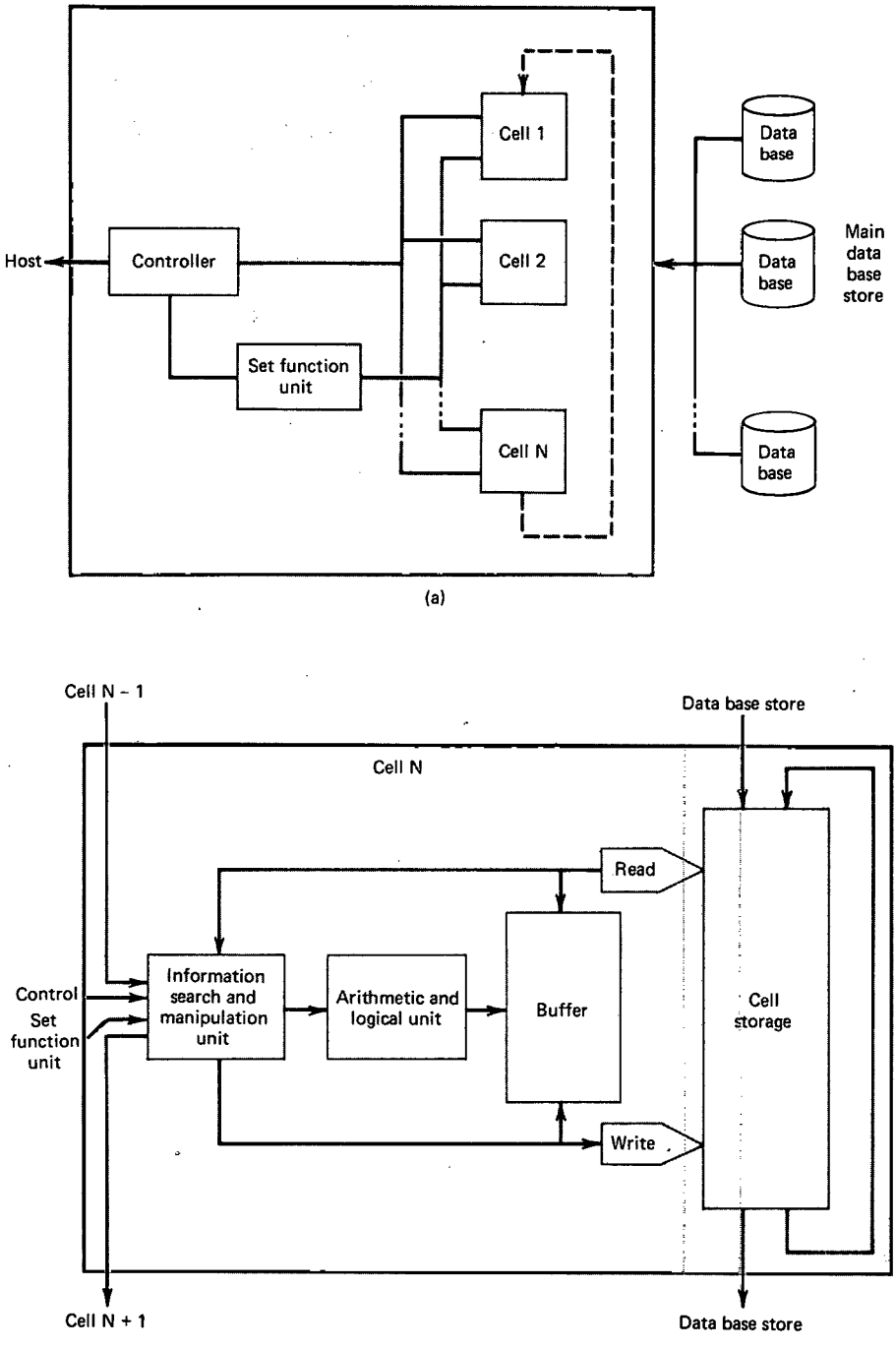


Figure 8-15 The RAP system. (a) RAP system organization. (b) RAP cell organization.

vice subdivided into cells. Each cell is capable of searching and manipulating data. The cells are interconnected as shown in Fig. 8-15a so that a cell may either manipulate data directly in its own storage or may indirectly manipulate the data in other cells. Each cell contains a buffer to connect the cell processing unit to the data base, as well as an arithmetic and logical unit and a separate search and manipulation unit as shown in Fig. 8-15b.

If the data base is small enough to fit within the cell storage areas, then RAP may be considered to be a multiple processor using direct searches. When the data base is too large for the cell storage, then the search is indirect since the necessary data must first be transferred from the main data base store. The latter case is more general; hence RAP is considered a multiple processor indirect search device.

RAP requires the general-purpose computer to interact with the user for query or command processing. These must be translated by the general-purpose device before they are sent to RAP. In operation, each cell operates simultaneously with all other cells by processing the data stored in its memory. Each information unit which meets the search criteria is then sent to a set function unit for potential integration with the results from other cells. For example, the set function unit may count the number of information items meeting the search criteria. The information items eventually selected are delivered to the general-purpose computer for presentation to the user or for further processing [21–23].

When the data base is large, RAP is used in conjunction with conventional disk drives. A section of the data base is moved to a RAP cell at each stage of operation. That is, a large data base such as the motor vehicle data base for a state government would of necessity be searched in many segments to isolate the SILVER automobiles.

The RAP operations require that all the information on one track of a disk be of the same type. That is, the automobile records would be distinct from the driver records. Thus, the search for the SILVER automobiles will not require a test of the driver records. The process needed to search for SILVER automobiles will therefore require the following steps:

1 Locating the automobile records
2 Loading the automobile records onto RAP
3 Isolating the color field
4 Retrieving the records with color field equal to SILVER

*H Data Base Computer (DBC)

The data base computer was designed at Ohio State University to manipulate very large data bases using currently available technology. The DBC is assumed to be connected to a large general-purpose computer for direct interaction with the user. The retrieval operations are divided into two cycles, including first the functions that access the data (the data loop) and the functions associated with the structure of the information (the structure loop) as shown in Fig. 8-16.

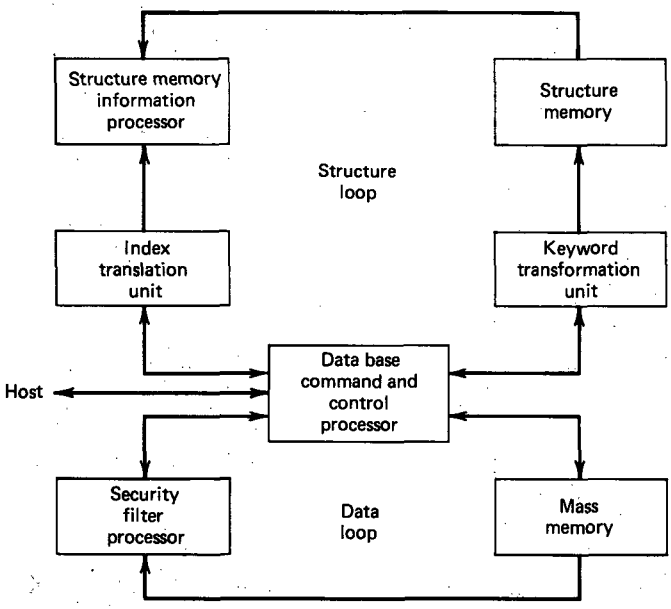


Figure 8-16 Data base computer organization.

The functions are accomplished by independent modules working on a continuous stream of data (a pipeline architecture). The central module of the DBC is the data base command and control processor. This element receives requests from the general-purpose computer system, translates these requests to commands for the various functional components of the DBC and controls the functions of the individual components. When the commands have been executed, the data base command and control processor delivers the requested information to the general-purpose computer.

The structure loop is composed of four functional units. Together these four functions translate a request into a set of physical locations to be searched. The physical locations are, in fact, blocks of mass storage which contain the information items. The data loop has two functional units which access and check the information items before they are sent to the general-purpose computer.

The structure loop first decodes keywords into one or more file names and fields within the file which must be searched. This is accomplished in the keyword transformation unit. This translation produces code numbers designating the addresses of the individual search terms.

The structure memory is, in fact, an inverted file for the data base. An entry consists of a code number and a series of three numbers of the form

LOGICAL POINTER, CLUSTER NUMBER, SECURITY SPECIFICATION

where (1) the logical pointer identifies a block of information items that are po-

tentially relevant, (2) the cluster number is assigned to a record at the time it is entered into the data base and identifies subsets within blocks, and (3) the security specification is a simple attribute which is checked prior to the retrieval of any information.

The results from the structure memory module are then passed to the structure memory information processor. This device performs Boolean and logical operations. These operations are performed in a manner similar to that described for the inverted file operations in Chapter 2. The results of these operations are sets of logical identifiers. These identifiers are passed on to the index translation unit, which translates them from logical pointers to the physical addresses of the data items.

If the information is to be directly retrieved, the data base command and control processor passes the physical addresses identified by the index translation unit on to the mass memory. The mass memory also performs Boolean and logical operations in order to identify the actual information items to be retrieved. In comparison the structure loop determines which blocks of memory are to be examined. The mass memory unit selects the appropriate block of storage and tests the individual information items to determine which of these qualify for retrieval.

Once the specific information items have been selected, the security filter processor sorts the items into a desired order and performs final security checks to ensure that the user is qualified to receive the items [24,25].

A user request for silver colored automobiles from a data base which includes both automobiles and licensed drivers might be conducted as follows:

1 The keyword SILVER is passed through the keyword transformation unit. This converts SILVER into a code which can be used to search the structure memory.

2 The structure memory is searched for all entries with this value. This search is conducted in parallel across all entries. The result is a set of addresses of blocks of records in which at least one record contains the desired value.

3 In this simple search there is no necessity to combine sets so the structure memory information processor simply passes the addresses on to the index translator unit.

4 The index translator unit is used to convert the logical addresses to specific disk cylinder addresses. These physical addresses are then passed to the data base command and control computer.

5 The mass memory device (usually a disk) is used to scan the individual blocks identified by the physical addresses. Each record is scanned for the value of its color field, and those equal to SILVER are selected and sent to the security filter, back to the data base command and control processor, and finally back to the host computer.

I Other Special-Purpose Devices

Several devices were described in the preceding subsections that are capable of performing the entire information retrieval process, or at least major portions of the process. One can, however, also consider the development of more spe-

cialized "black boxes" that perform more restricted functions. An especially onerous operation is the comparison of lists of items that becomes necessary to determine the "hits" when Boolean queries are processed with inverted files. Indeed the lists of documents corresponding to two search terms must then be merged using set union and set intersection operations when the terms are related by OR and AND operators, respectively. When the document lists are long, the list processing operations become expensive.

In these circumstances special list merge networks may be useful in which many of the required operations are carried out in parallel. Consider, in particular, two lists of n entries each. When a single comparison unit is available capable of comparing an entry on list 1 with an entry on list 2, at least n sequential comparison operations are needed to traverse the lists. Alternatively, one can conceive of n comparison units all working in parallel using different elements of the input lists: the first entry of list 1 might then be compared with the first entry of list 2 using compare unit 1; the next entries of the two input lists are treated in compare unit 2, and so on until the last (nth) elements of the lists are processed in compare unit n.

The outputs of the initial set of comparisons might be fed in pairs to n/2 additional compare units all working in parallel, followed by n/4 more compare units on the next level, and so on down to a single unit that compares the final two elements. When such a network of comparison units is used, the n-element input lists can be "traversed" in log n steps, instead of the n steps needed for the single compare unit. Various devices of this type have been proposed and tested in experimental systems [26–28].

Another possibility for effecting savings in retrieval consists in optimizing the available query statement before initiating a search operation. A query preprocessing system could then be built, designed to check the query for syntactic errors, to reduce the Boolean statements to minimal form, to pretest the queries against a small data base, and to evaluate the usefulness of some search terms prior to the actual search operation [29]. In situations where information queries are submitted by users unfamiliar with the retrieval operations a query preprocessing system might well pay for itself in improved efficiencies when the final searches are actually conducted.

As an extension of the previously mentioned special-purpose devices, one might conceive of a whole array of separate devices each designed to optimize a particular operation such as text scanning, search of inverted indexes, list merging, and query reformulation. Such an organization might then constitute a powerful, but probably also expensive, retrieval configuration [30].

3 TEXT ACCESS METHODS

*A Dictionary Search Methods for Static Files

In the preceding section various hardware devices were described that may be useful to store and retrieve large information files. This section examines specific file accessing methods that are of special importance in a bibliographic re-

trieval system dealing with natural language texts. The first and most important task concerns the search of a dictionary or index to find the entries corresponding to particular query terms. For example, given a library catalog in which the items are arranged in alphabetical order according to author name, one may want to find the entries corresponding to particular authors. Alternatively an inverted index file must be searched to determine the location of the lists of document identifiers corresponding to particular query terms.

Linear Scan Certain file access methods were introduced in Chapter 1. The present treatment stresses accessing methods that are particularly useful in a dynamic file environment where file additions and deletions must be processed together with information retrieval requests. The most obvious file accessing method that can be used to find a record identified by a particular search term consists in performing a *linear scan* of the complete file, one record at a time. This method can be used no matter how the file is ordered. However the traversal time of the complete file will be excessive when the file is large (of order n for a file of n records). A linear scan is therefore not normally usable in practice.

Binary Search The next possibility consists in using an access technique that eliminates from further consideration at each search step not just one single record as in a linear search, but a whole section of the file. The well-known *binary search* described in Chapter 1 is a case in point where an ordered file is

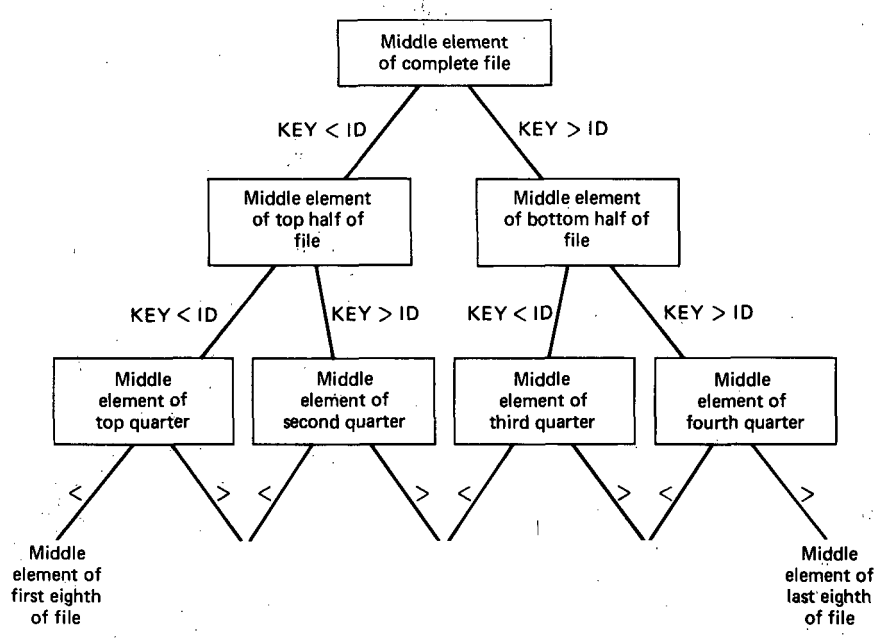


Figure 8-17 Implicit tree organization of binary search. (Search proceeds to left of node when KEY < ID and to right of node when KEY > ID.)

searched by eliminating one-half of the file still under consideration at each search step. In a binary search the query term, also known as the "key," is first compared with the record in the middle of the file. If the search key is identical with the identifier (ID) attached to the middle record, an acceptable record has been found. Otherwise, one next proceeds to the middle of the top half of the file if the key is smaller than the record ID, or to the middle of the bottom half if the search key is larger than the record ID. This process of dividing the file into two halves continues until an appropriate record is found.

The binary search may be described graphically by a tree as shown in Fig. 8-17. The elements of the ordered file which are compared with the search key are represented by the tree nodes. Whenever the value of the search key is smaller than the current record identifier, the left path is chosen from the node; conversely the right path is chosen if the search key value exceeds the value of the record ID. It is known that tree search strategies of the type illustrated in Fig. 8-17 are much faster than linear scans (of order log n instead of n as before) [31–32].

A study of the binary search strategy shows that the method used to compare the search key with a given record identifier is precisely the same at each tree node. The only difference in procedure occurring from one node to the next in the search tree is a redefinition of the portion of the file to be considered in the search. Let BEG and END designate the addresses of the first and last elements of the file, respectively, and let the address of the middle element be defined as MID = [(BEG + END)/2]. The binary search strategy for a given file F may then be summarized by the four steps of Table 8-2. Assuming that the search of the complete file is expressed by using parameters (KEY, BEG, END, FILE F), then the left path of the search tree leads to a new search with parameters (KEY, BEG, MID − 1, FILE F). That is, the end of the file is now assumed to be the record immediately preceding the middle record of the file. Moving to the right in the search tree means that the search is next carried out with parameters (KEY, MID + 1, END, FILE F). That is, the beginning of the file is now defined as the record immediately following the middle element. A process like the binary search that is carried out by repeating the same program with only a change of the parameters used is known as a "recursive" process. Recursion is an important concept in computing because the automatic equipment is especially useful when the same process can be repeated many times.

Table 8-2 Outline of Binary Search Program

		BIN SEARCH (KEY, BEG, END, FILE F)
1. If	ID[MID] = KEY	Search ends because the wanted element has been found
2. If	ID[MID] > KEY	Repeat binary search with parameters (KEY, BEG, MID − 1, FILE F)
3. If	ID[MID] < KEY	Repeat binary search with parameters (KEY, MID + 1, END, FILE F)
4. If	BEG > END	Stop the search because no entry exists in file equal to KEY

BEG: address of first element of subfile
END: address of last element of subfile
MID: address of middle element of subfile
ID[k]: identifier of kth record

Estimated Entry Search In the binary search, the next record identifier to be compared with the search key is chosen at each point from the middle of the remaining subfile. In many cases, it is possible to make a guess about the position of a file element when its identifier is known. For example, when a normal dictionary is used to find the word RETRIEVAL, one would certainly not begin by opening the dictionary somewhere in the middle. Rather one might look about two-thirds of the way down because a term starting with the letter R would occur toward the end of the file. The same is true when trying to find someone's name in the telephone book or a given bibliographic citation in a library card catalog.

When a guess can be made about the likely position of a given record in an ordered file, the binary search can be replaced by an *estimated entry* search, which is similar to a binary search except for the computation of the next record element to be considered in the search. When good guesses are made about the position of a wanted record, an estimated entry search will be faster than a binary search (but still of order log n for a file of n items).

Direct Access Search If one knows or can compute the exact position in which a wanted record appears in the file, the file can be accessed directly. A search of order 1 is then carried out instead of order n for a linear scan, or order log n for a tree search. A well-known direct search method, already discussed at length in Chapter 2, is the *indexed search*, where an auxiliary index is used containing the addresses of the records corresponding to a given search key. Obviously, there is no need to conduct the search of the main file in these circumstances, since the needed addresses are obtained from the index. Assuming that the index search can be performed free of charge—a very unrealistic assumption—only one (order 1) file access is required in the main file to retrieve the wanted item.

It was mentioned earlier that the creation of an index may be costly because any auxiliary file must be stored and maintained. An alternative well-known direct file access process, known as a "hashing," or "scatter storage," replaces the use of auxiliary indexes by a computation or transformation of the search key into a storage address where the corresponding records are stored.

One of the problems arising with the use of a hashing system is the choice of the hashing function used to transform the search keys into appropriate memory addresses. In general the number of possible search keys is very large, whereas the memory space available to store the corresponding record identifiers (known as the "hash table") is small. Hence it is necessary to transform a large number of possible keys into a small number of memory addresses. The transformation should be done in such a way that the available memory space is evenly used; that is, each memory address should have an equal chance of storing a given record. Furthermore, clusters of records exhibiting nearly equal keys should be broken up—for example, records corresponding to the search keys HOP and HOPE should not be mapped into the same address.

Two of the best-known hashing methods are the multiplication and divi-

Table 8-3 Typical Hashing Method Using Key Multiplication
(Assumed Hash Table Size is 64 Positions)

Record M:	Key 11 01 01 00
	Key squared 1 0 1 0 1 [1 1 1 1 0 0] 1 0 0 0 0
	Decimal address corresponding to $(1\ 1\ 1\ 1\ 0\ 0)_2$ equals
	$32 + 16 + 8 + 4 + 0 + 0 = 60$
Record N:	Key 11 01 01 01
	Key squared 1 0 1 1 0 [0 0 1 0 0 1] 1 1 0 0 1
	Decimal address corresponding to $(0\ 0\ 1\ 0\ 0\ 1)_2$ equals
	$0 + 0 + 8 + 0 + 0 + 1 = 9$

sion hash functions. In the multiplication method the key is multiplied by itself (squared) and the middle digits of the product are used as a record address; in the division method, the key is divided by a prime number and the remainder after division is transformed into the needed address [33]. A sample key transformation using the multiplication method is shown in Table 8-3 for two nearly equal keys using an assumed hash table size of 64 (equal to 2^6) memory positions. The two keys are 11010100 and 11010101, respectively, corresponding to the letters M and N in the well-known EBCDIC coding system. The keys differ in the rightmost binary digit only. The squaring operation produces 16-digit binary products. In the example, the middle six binary digits are then transformed into one of 64 memory addresses by conversion to decimal form. The record corresponding to key M in Table 8-3 will be located in position 60 of the hash table, whereas key N is transformed into position 9.

A hashing system (unlike a binary search scheme) can gracefully accommodate situations where several records exhibit the same key, because a complete *bucket* of records can be associated with each hash table address, rather than a single record only. A bucket provides space for several records, but even when larger buckets are used, *collisions* are unavoidable for most hash functions. A collision occurs when two or more distinct keys are transformed into the same record address. Collisions will complicate the retrieval process, because the record identifiers found in the designated buckets must be checked before the corresponding records are actually retrieved from the file. Furthermore, collisions may lead to *overflow* conditions when not enough space exists to accommodate all the records that must be located in a particular bucket. Various provisions can be made to handle the overflow: a second hashing operation can produce a new overflow bucket address located in a secondary hash table; alternatively, a pointer system can be provided to designate for each bucket the addresses corresponding to any overflow records. Overflow problems produce efficiency losses in the hashing system because extra memory accesses are then required to retrieve the needed records.

*B Dictionary Search Methods for Dynamic Files

Dynamic and Extensible Hashing The basic hashing or scatter storage method described up to now supports searches involving single keys in a static

file with few file additions or deletions. Sequential searches that handle the records in some specified order are difficult to carry out in a hashed file, because the records are scattered throughout the storage area. The scattering of the records also destroys any natural clustering properties of the records. For example, records identified by the terms CAT and CATS will not be found adjacently, even though this might be useful in certain applications.

When the file grows and the original hash table size m becomes too small, it is necessary to rehash the whole file using a larger hash table size, say of size 2m. The rehashing operation is onerous because all file records must now be moved in storage. For this reason *dynamic* or *extensible* hashing systems have been introduced which avoid overflow conditions and the repositioning of records when the file size grows [34–36]. Various schemes have been proposed: in general an auxiliary index is interposed between the key terms and the final record addresses in the main file. The hashing operation is then used to identify an address in the index rather than in the main file, and each index position in turn points to the main file addresses of the corresponding records. Instead of changing the main file configuration as new records are added, the index is made to grow; thus when a given bucket overflows, the corresponding index entry is changed to produce two entries, corresponding to two new buckets replacing the single original bucket. Similarly, when the file shrinks, two or more buckets can be merged into a single bucket by appropriate transformations in the index. When the index is used as the hash table instead of the main file, an even use of the index space corresponds to an even use of the full memory space in the standard hashing system.

Dynamic Tree Search It was mentioned earlier that the basic binary tree search is not easily adapted to dynamic file conditions. Since the file must be maintained in ordered sequence according to key values, an insertion of a new record into its proper place may cause the displacement of half the records in the file. Furthermore the tree traversal operation needed to locate the next record identifier to be matched against the query may be expensive to perform. Indeed for a file of n records, the binary tree exhibits $\log_2 n$ levels, and hence $\log_2 n$ different record identifiers must be compared with the query key. Since a new disk seek operation may be needed for each record identifier, the standard binary tree operation will be expensive to carry out for large files.

This suggests that a tree search could be more efficient when the number of levels in the search tree is small. The height of the tree, that is, the number of tree levels, can be reduced by packing more than one key value (record identifier) into each node of the tree. Furthermore some file growth can be accommodated by leaving space in each tree node for new key values corresponding to new records added to the file. The best known of the dynamic search trees obeying these specifications is the so-called B-tree [37–39].

In a B-tree of order d, the root node at the top of the tree contains at least one key value and two pointers to the next tree level corresponding to keys that are smaller and larger, respectively, than the original key value. Each node

other than the root contains at least [d/2] key values and at most d key values. The number of pointers to the next tree level is one larger than the number of keys at the corresponding node. A typical B-tree of order 4 is shown in Fig. 8-18. Each node of the sample tree can in principle accommodate four different terms, although as few as two terms may actually be used. The number of pointers to the next tree level varies therefore between 3 and 5. The natural (alphabetic) order of the keys is maintained because the pointer to the left of a given term is used to find smaller terms than the given key value (that is, occurring earlier in the alphabetic sequence), whereas the right pointer locates larger terms that are higher in the alphabetic order. For example, the left pointer from NETWORK locates CATALOG, HARDWARE, and MORPHEME, whereas the right pointer locates REVIEW and SYNONYM.

A B-tree search is carried out like a binary tree search except that all record identifiers located at a given node are compared with the available key values and the appropriate left or right pointers are chosen depending on the outcome of the key comparisons. For example, when the B-tree of Fig. 8-18 is searched with the query term FILE, the left pointer is taken from node A, followed by the pointer located between CATALOG and HARDWARE from node B (because FILE is larger than CATALOG, but smaller than HARDWARE), followed by the pointer between ENCYCLOPEDIA and GRAMMAR from node E, and so on.

Since all key values stored in a given node are obtainable in one file access, the maximum number of file accesses is of order $\log_d n$ for a file of n items, corresponding to the height of the tree. New keys can be added to or deleted

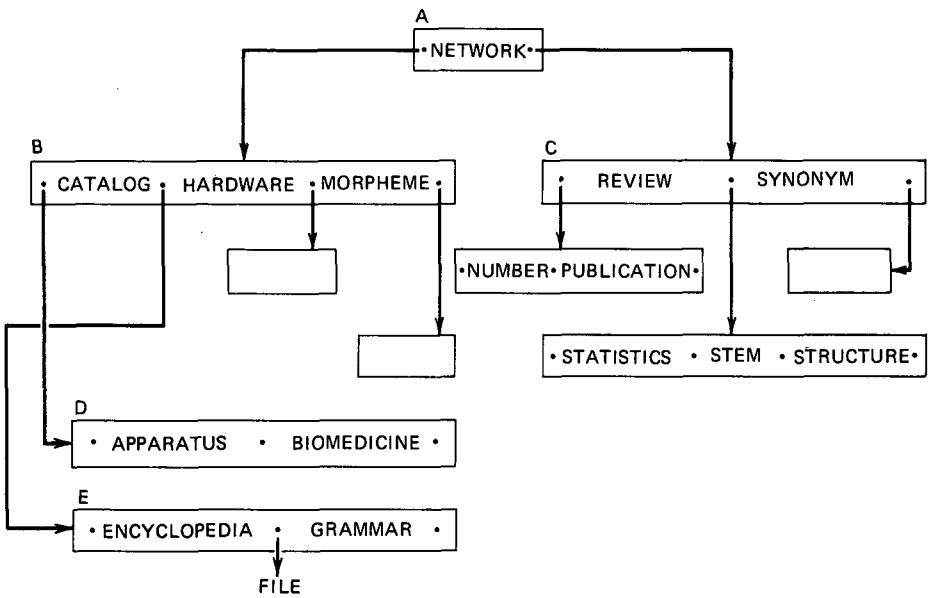


Figure 8-18 Three-level order 4 B-tree (between 2 and 4 keys per node).

from a particular node so long as the basic size restrictions are obeyed. However, when a new key value is added to a node of size d, an illegal node of size d + 1 is created. This situation is handled by splitting a node of size d + 1 into two nodes of size at least equal to [d/2]. Such a splitting operation is shown in Fig. 8-19 for a B-tree of order 4. The assumption is that the term CLASS must be added to node B of the tree of Fig. 8-19a. Since this node already contains four terms, it is split into two pieces, and a new separating term is added to the father node on the next higher tree level (node A of Fig. 8-19b). Because the father node has now grown in size by one term, it may itself have to be split. The node splitting operation may thus propagate upward along the levels of the tree, the maximum number of splitting operations being equal to the height of the B-tree.

The reverse node merging operation may become necessary when terms are deleted from a given node. This situation is illustrated in Fig. 8-20 where the term ENCYCLOPEDIA is deleted from the B-tree of Fig. 8-18. An undersized node is then produced containing only one term (node D of Fig. 8-20). This node is merged with its brother (node C of Fig. 8-20) and the term CATALOG, which is no longer necessary to distinguish the merged nodes, migrates downward by one level. The father node may then become undersized and may in turn have to be merged with a brother. The node merging operation like the earlier splitting operation may thus migrate upward along the B-tree levels.

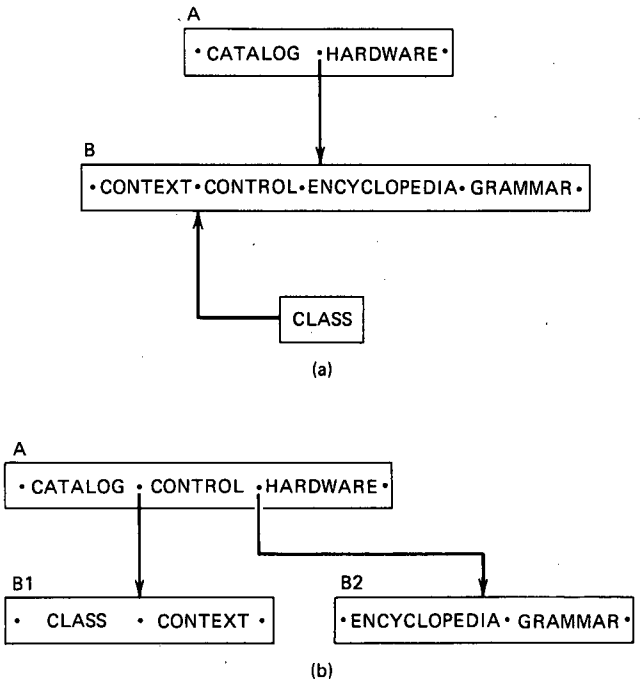


Figure 8-19 Node splitting operation for B-tree of order 4. (a) Initial condition prior to insertion of CLASS. (b) Condition following bucket splitting after insertion.

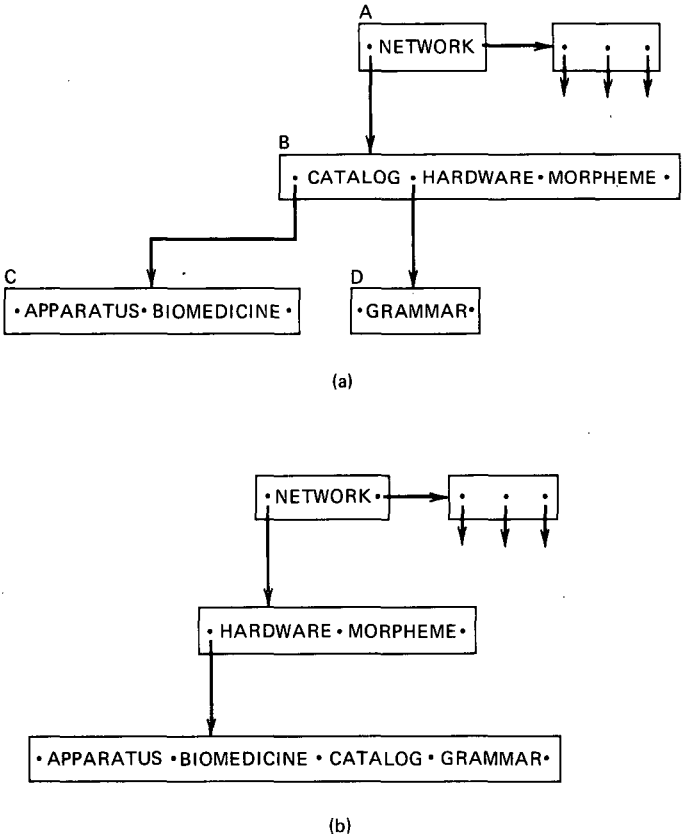


Figure 8-20 Node merging operation following term deletion. (a) Initial state following deletion of ENCYCLOPEDIA. (b) Final state following deletion of ENCYCLOPEDIA.

A search of a B-tree of order d is of order $\log_d n$ for a file of n records. The node insertion and deletion operations are also of order $\log_d n$, since each operation may affect at most one node on each tree level. A B-tree implementation requires more memory space than the conventional hashing system, because the key values and pointer structures must be kept in storage in order to be used. A B-tree search is also more time-consuming than a standard hashing operation—typically by a factor of 3 or 4; but the B-tree search is much faster than a binary search by a factor of 10 or more. Because of the simple way of reorganizing the B-trees following insertion and deletion of key values, the B-tree search system has become the standard for the implementation of dictionary and index search systems in dynamic file environments.

Many other search tree systems have been proposed to handle special situations. In the well-known *digital search trees*, or *tries*, each node of the search tree is associated with a portion of a key value rather than with a whole key. In many cases, a small number of tree levels may then be used to accommodate key values of widely varying length. For example, only three trie levels are

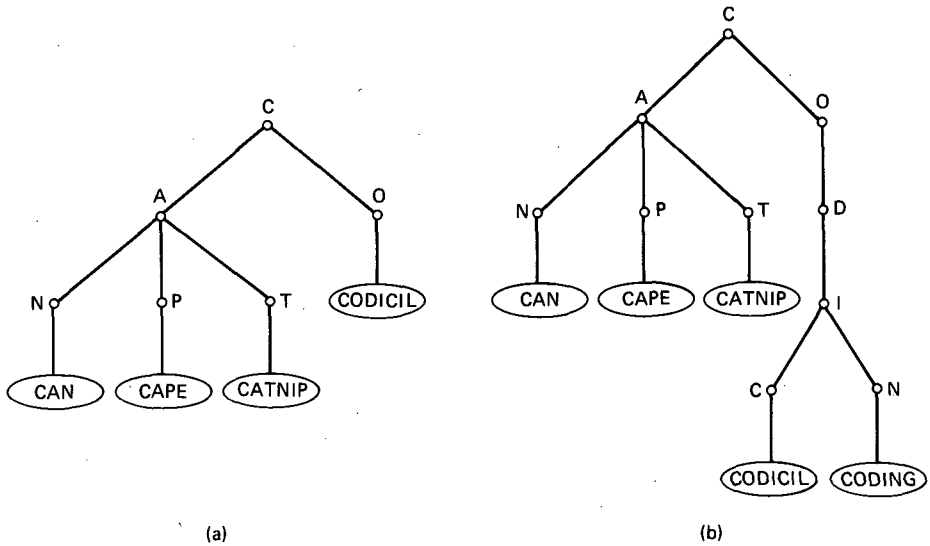


Figure 8-21 Digital search trees. (a) Initial digital search tree. (b) Digital search tree following addition of CODING.

needed to distinguish the terms CAN, CAPE, CATNIP, and CODICIL, assuming that a node is associated with a single key character, as shown in Fig. 8-21a. The basic trie structure may change drastically when new terms are added as shown in the example of Fig. 8-21b illustrating the addition of the term CODING. However procedures exist for optimizing the representation of digital search trees and for handling term additions and deletions [33,40–41].

*C Multiple Key Dictionary Search

All the foregoing dictionary access methods are usable for single key searches where only one query key is present. Single key searches are relatively simple to implement because the existing one-dimensional storage devices can easily accommodate a file maintained in order according to the single (one-dimensional) key values. In practice, search requests may include many different keys. For example, one wants to retrieve all the bibliographic records pertaining to a given author *and* published in a given year, or all the records dealing with INFORMATION as well as with RETRIEVAL.

The standard access methods, with the exception of systems maintaining dense (inverted) indexes for each possible key value, are difficult to adapt to multikey searches. In principle, one can maintain auxiliary indexes covering a combination of several search terms. For example, given three search terms A, B, or C, one could maintain seven distinct indexes covering the records pertaining to the keys A, B, and C separately, as well as to the possible term combinations A and B, A and C, B and C, and A and B and C. When a larger number of individual terms are present, the number of term combinations is however very large, and the task of identifying the most productive term combinations is difficult [42].

Another possibility consists in using a *superimposed coding* system, where each individual term is encoded by a binary number and the logical union of the various component codes designates the term combination. For example, assuming that CHOCOLATE, NUTS, and VANILLA are identified by codes 100010, 001001, and 100001, the query term CHOCOLATE CHIP COOKIE (assumed to be composed of CHOCOLATE, NUTS, and VANILLA) is represented as 101011 equal to $100010 \vee 001001 \vee 100001$ [33].

The superimposed coding system was introduced for retrieval in the well-known edge-notched card systems some 40 years ago. These systems are unfortunately afflicted by the "false drop" problem, where a given code combination retrieves many spurious records. Certain term codes may also be included in other term code combinations. In the previously used example, the term VANILLA is already included in the combination of CHOCOLATE and NUTS. A longer treatment of term coding systems is unhappily beyond the scope of this discussion.

D Text Scanning Machines

In addition to the standard dictionary or index searches where table entries corresponding to particular keywords or terms must be identified, it may also be necessary in a bibliographic retrieval system to scan linear texts in order to determine whether a given phrase or query pattern occurs in the text, and if so where. Indeed, it was noted in Chapter 2 that some conventional retrieval systems include options for a so-called string search in which the query terms are directly compared with document texts or text excerpts. In these circumstances, no indexes or auxiliary files need be maintained. Unfortunately, a full text scan is normally slow and hence inappropriate for on-line searching. Special methods and devices have, however, been developed to speed up the text search process [43–47].

One particular text search device is based on the idea also incorporated into RAP and CASSM and previously illustrated in Fig. 8-8, in that several search modules are used to process different portions of the stored texts under the supervision of a master control computer. Each search module is attached to a mass storage device such as a disk. When a command is issued by the master control computer to initiate a text search, each search module begins a sequential search of its portion of the stored texts. The text read by a search module is continuously compared with the query statements. When a document is found that satisfies a particular query statement, a report is sent to a master control computer which may either directly report the discovery to the user or accumulate the information for inclusion in a summary report to the user. To gain speed, the queries can be batched and responses can be generated simultaneously for many different queries.

The actual search process may be broken into two parts: (1) term detection and (2) query resolution. Term detection is concerned with the location of query words or pattern in the stored data base. Query resolution is the process of determining if the combination of matches found by the term detection pro-

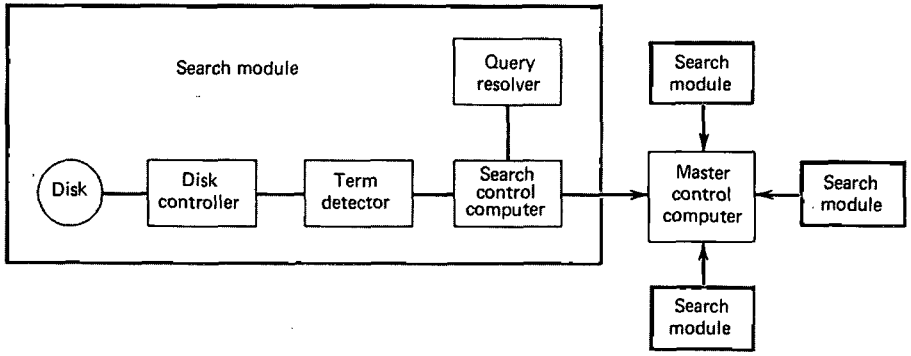


Figure 8-22 Information retrieval machine-searcher organization (several search units attached to single master control).

cess matches the term combination specified in the user's query. For example, a user may specify that two particular terms must appear in the same document (a Boolean AND operation). Additional search facilities may include term combinations based on all the Boolean operators previously discussed, as well as on word proximity, and sentence and paragraph inclusion specifications.

A typical search module for a text scanning machine could then be based on the use of four special units: a disk controller, term detector, search control unit, and query resolver, as shown in the diagram of Fig. 8-22. The search control unit carries out data transfer operations, communications between the term detector and the query resolver, and the general overall control operations for the search module. The disk controller oversees all access operations to the disk and the stored information. Equipment of the type represented in Fig. 8-22 can scan text at rates of up to 1 million characters per second using a single search module only.

**E String Matching Using the Finite State Automaton Model

A key element in a text scanning system is the term detector which must necessarily operate rapidly if large quantities of text are to be processed under operational conditions. The most simple-minded text scanning system possible is based on a character-by-character comparison of a given query (key) pattern with a text excerpt (string). The basic procedure consists in comparing the first character of the query pattern with the first character of the text string. If these first characters match, the second characters are compared, and so on. If any pair of characters does not produce an exact match, the standard procedure is to shift the query pattern over by one character position and to restart the matching procedure. This becomes expensive because the matching process must start over near the beginning of the input pattern many times. For example, given

query pattern AAAB
text string AAAAAAB

16 character comparisons are required before it becomes clear that the four input characters match the last four characters of the text string. In general because of the backtracking and restart characteristics of the standard text scanning method, up to m · n character comparisons may be required in the worst case in order to match a query pattern of length m against a text string of length n.

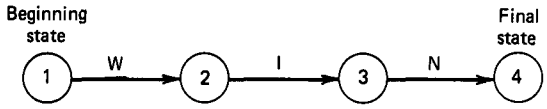
Fortunately, more sophisticated approaches to the text scanning problem exist. Algorithms have been devised that operate in linear or even sublinear time—that is, the number of character comparisons needed does not exceed (and in fact may be substantially smaller than) the number of characters in the query and text strings, instead of the square of that number as in the earlier example [48–50]. These algorithms analyze the query pattern (that is, the query terms) prior to the actual search operation and construct auxiliary modules or tables to control the actual character comparisons used to match the query patterns with the text strings.

A favorite term detection process is based on the concept of the *finite state automaton* (FSA) which was introduced in Chapter 7 as a controlling element in the augmented transition network grammars [26,48,51]. An FSA is a conceptual machine composed of five basic elements:

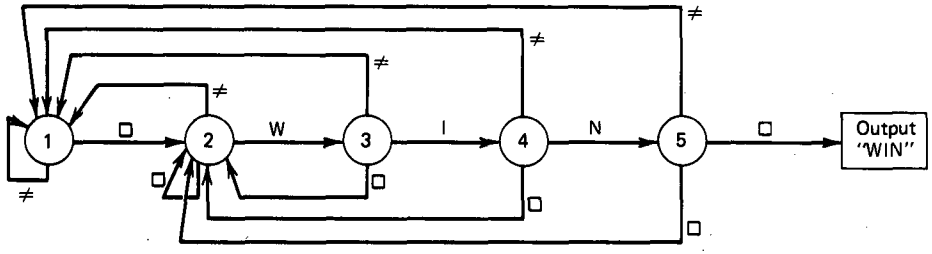
1 I—a set of input symbols acceptable to the automaton
2 S—a set of states
3 R—a set of rules which determine the next state of the automaton given its current state and an input symbol
4 B—a beginning state (an element of S)
5 F—a set of (one or more) final states (elements of S)

It will be remembered from the preceding chapter that the operation of a finite state automaton is representable in diagram form by using nodes (circles) to represent the individual states and branches (edges) between the nodes to represent transitions between states. A symbol attached to each branch then represents the element of the set of acceptable input symbols causing the particular transition from one state to the next. Figure 8-23a shows a finite state automaton with a beginning state 1 and a final state 4. The input symbols accepted by the FSA are assumed to be the letters of the alphabet, and the sample automaton of Fig. 8-23a "accepts" the input string WIN.

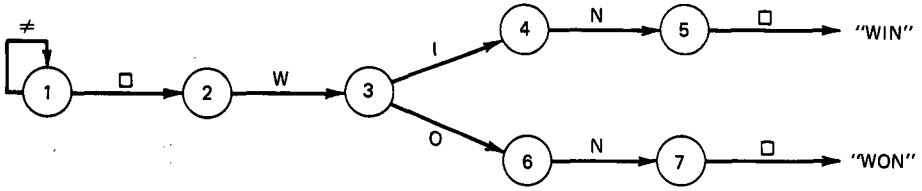
The automaton of Fig. 8-23a handles the string WIN no matter where that string occurs in the text; that is, the automaton does not discriminate between SWINE, WIND, TWIN, and WIN. Furthermore, no instructions are given in Fig. 8-23a about what happens when the input does not contain the exact letter string W followed by I followed by N. Clearly, a usable FSA capable of detecting complete words in running text must be able to detect word delimiting, or interword characters, such as blanks that might occur between words; furthermore, instructions must be provided about what is to happen when an expected character is not received at the input. The complete automaton needed to de-


Beginning state
Final state
1
W
2
I
3
N
4

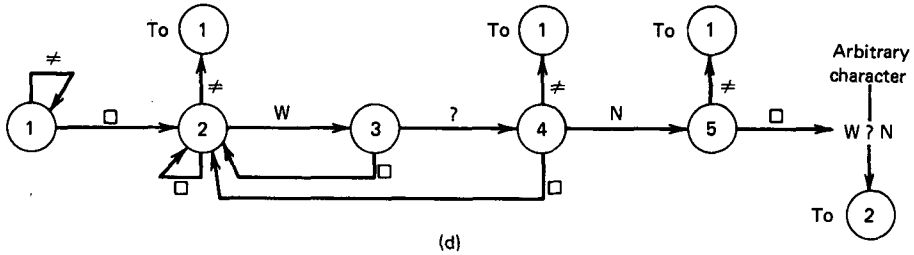
(a)


≠
□
W
I
N
Output "WIN"

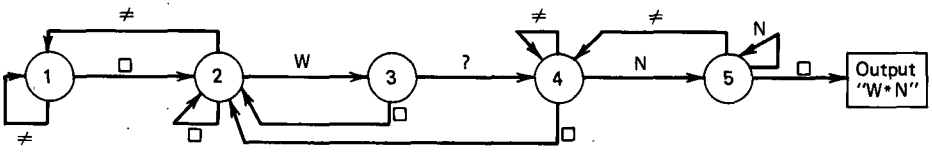
(b)


≠
□
W
I
O
N
"WIN"
"WON"

(c)


To 1
≠
□
W
?
N
Arbitrary character
W ? N
To 2

(d)


≠
□
W
?
N
Output "W*N"

(e)

tect the word WIN is shown in Fig. 8-23b. An interword delimiter is indicated in that figure by a □, and an arbitrary character other than one attached to a labeled branch leaving a given state is designated by ≠. When the string □WIN□ occurs at the input, the traversal of the automaton from beginning state 1 to final state 5, and then back to state 2, is straightforward. If the input consists of □WON□, the normal transitions would occur from state 1 to states 2 and 3. At that point, the input character is an O; the three branches out of state 3 are labeled I, □, and ≠, respectively. Since the letter O is neither I nor □, the ≠ exit is taken from state 3 back to 1, where the automaton is now prepared anew to recognize □WIN□. The recognition process also fails for the input □WINTER□, because the path taken from state 5 returns to state 1 rather than back to state 2 via the output label "WIN."

A single automaton can be built for the detection of several query words. The graph of Fig. 8-23c presents an FSA capable of detecting WIN and WON. The alternative exits back to state 2 when a □ occurs, or back to state 1 for the occurrence of any character other than one already attached to a branch (≠), are omitted in Fig. 8-23c. The full construct including the upper (≠) and lower (□) exits is similar to that of Fig. 8-23b.

The automaton of Fig. 8-23c could be simplified by eliminating one state, if the string W?N were to be recognized, where ? represents exactly one arbitrary character. The corresponding automaton is shown in Fig. 8-23d. Note that the label ? is not identical with ≠, since ? represents any character other than □, whereas ≠ represents the default exit taken for any character other than one attached to a label. The automaton of Fig. 8-23d recognizes WAN, WIN, WON, or for that matter WNN, WZN, etc.

Suppose that one were interested in recognizing patterns such as WIN, WON, WOMAN, WOMEN, or in general the sequence W*N where * represents any character string of arbitrary length. The automaton needed for this purpose is no more complicated in principle than the one used for the recognition of "WIN" in Fig. 8-23b; it is shown in Fig. 8-23e. The main difference between Fig. 8-23e and b lies in the recognition of the arbitrary string of "don't care" characters. This requires a redefinition of the default state chosen when a specifically labeled character does not occur as expected. In the example of Fig. 8-23e the initial default state is state 1. That is, if the input consists of SIN instead WIN, one returns to state 1 from state 2 when the S occurs at the input instead of the expected W. After state 4 is reached following the input of characters □, W, and ?, the default state for ≠ characters is now changed to state 4. At that point the W has already been recognized, and any arbitrary string ending in N is now acceptable following the W. A slight complication arises for strings that include several N's following the initial W, as in WINN, or WIN-

Figure 8-23 (*opposite*) Finite state automata used for string recognition. (a) Basic automaton for detection of WIN. (b) Complete automaton for detection of WIN. (c) Automaton for detection of WIN and WON. (d) Automaton for detection of W?N. (e) Automaton for detection of W*N.

Current state	□	≠	W	I	N
1	2	1	1	1	1
2	2	1	3	1	1
3	2	1	1	4	1
4	2	1	1	1	5
5	2	1	1	1	1

Figure 8-24 Tabular representation specifying new state given current state and input symbol for finite state automaton of Fig. 8-23b.

NIN. The appropriate pointers to handle these cases are included in the graph of Fig. 8-23e.

A general finite state automaton can be defined for any user-specified string of symbols. The state transitions may be represented in each case by a table using the current state as input and specifying the corresponding next state for each acceptable input symbol. An example is shown in Fig. 8-24 for the automaton of Fig. 8-23b. The tabular representation of the finite state automaton can be used for rapid comparisons of an arbitrary incoming text with query terms previously encoded into the automaton.

In principle a separate automaton can be used for each individual term in a query. Several automata can also be combined for the detection of groups of terms and of contiguous terms. Thus, two given automata A and B could each search for a distinct term. The results of the comparison for A and B might be passed on to a new automaton C. When the two search terms are not expected to occur contiguously in the input text, the search results are directly passed on to the output device. Otherwise, automaton C is used to verify the text contiguity condition. The organization of the several finite state automata is illustrated in Fig. 8-25 [47–49].

The main problem encountered in the use of finite state automata for text scanning purposes is the need to build and maintain the transition structures for each incoming query. When new queries are continuously submitted to a retrieval system, the overhead inherent in the construction of the transition matrices may lower the efficiency of the process for operational use. On the other hand, if the query set remains constant over long periods of time—such as in systems for the selective dissemination of information—the state transition matching process is straightforward and efficient.

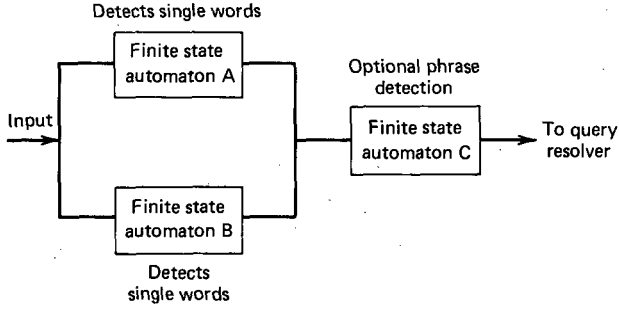


Figure 8-25 Multiple term detection using finite state automaton.

Various other efficient text scanning methods have been developed in the last few years [49–50]. The Boyer and Moore method is one of the most attractive ones. It is briefly described in the next subsection.

**F The Boyer and Moore String Matching Method

Like the previous FSA string detection system, the Boyer and Moore (BM) method depends on a prior analysis of the query pattern [50]. Specifically to use the BM process it is necessary to construct an auxiliary table including the positions in the pattern for all individual characters, as well as for all contiguous character strings consisting of more than one character.

Consider, for example, the input

1	2	3	4	5	6
B	A	N	A	N	A

A character position number may be used to designate the place of each character in the pattern. The pattern analysis would note that character B occurs in position 1, character A in positions 2, 4, and 6, and character N in positions 3 and 5. Furthermore the pair AN occurs in positions 2, 3, and 4, 5; NA occurs in 3, 4, and 5, 6; ANA occurs in positions 2, 3, 4, and 4, 5, 6; and so on. The position information is used later to determine the place where a matching operation must be restarted once a mismatch between characters has occurred.

The BM algorithm proceeds by first comparing the rightmost (rather than the leftmost) character in the pattern with a particular character in the string. If a match is found, a left shift is made and the character to the left of the rightmost pattern character is treated. A string pointer is used to designate the string character that must currently be matched with the given pattern character. This pointer is moved left by one character position when a match occurs; when a character mismatch occurs, the string pointer is moved to the right so as to designate the next string character to be compared with the rightmost pattern character. The principal complexity in the BM algorithm consists in determining exactly the pointer shift (in number of string characters to be skipped) in the event of mismatch.

Two principal rules are used to specify the permissible pointer shift when a mismatched character is detected:

1 When a mismatched string character is detected, an attempt is made to find that string character elsewhere in the pattern, and to determine the pattern shift necessary to achieve coincidence (known as the Δ_1 shift); should the string character not occur in the pattern at all, a shift by the whole length of the pattern is in order.

2 When a portion of the pattern matches some substring in the text, an attempt is made to find a repeating occurrence in the pattern of the originally matching subpattern; the shift needed to bring the new occurrence of the subpattern in coincidence with the matching substring is known as the Δ_2 shift.

In order to detect a complete match between a query pattern and a pattern of the text string, the conditions giving rise to the Δ_1 and Δ_2 shifts must both be satisfied, because a mismatched string character must eventually find a match in the pattern (hence the Δ_1 shift), and an already matching part of the pattern must again be matched following a pointer shift operation (hence the Δ_2 shift). At each point in the matching process it is then appropriate to execute a shift equal to the maximum between Δ_1 and Δ_2.

Consider the following query pattern and text string:

```
          1 2 3 4 5 6 7 8 9
Pattern:  c b a a b c a b c
String:   a b c d e f a b c a b c d e f a b c . . .
                    ↑ ✓ ✓ ✓
```

After a successful comparison between the ninth (last), eighth and seventh characters, a mismatch is found in the sixth character position. At this point, two pattern shifts are possible. Consider the Δ_1 shift first: Since the mismatched string character f occurs nowhere in the pattern, the pattern can be shifted all the way beyond the f character position, producing the following new situation:

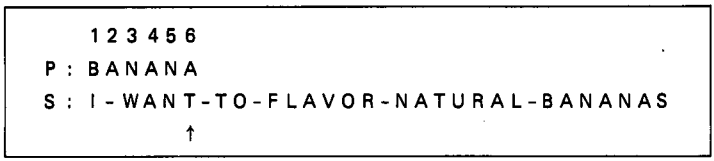
```
Pattern:                  c b a a b c a b c
String:   a b c d e f a b c a b c d e f a b c . . .
                                      ↑
```

The Δ_2 shift is determined by finding a new occurrence of the matching subpattern abc in positions 7 to 9. Since the abc subpattern is repeated in pattern positions 4 to 6, a shift of 3 will maintain the match with the originally matching substring:

```
P:          c b a a b c a b c
S: a b c d e f a b c a b c d e f a b c
                  ✓ ✓ ✓     ↑
```

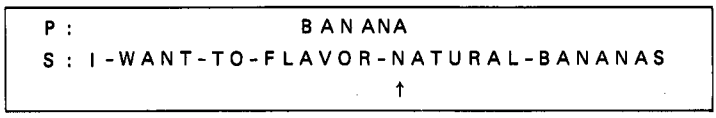
Since the largest possible pointer shift is most advantageous, the Δ_1 shift is executed in the case under consideration. A full example of the BM process is included in Fig. 8-26 [52].

It is easy to see that the number of required character matches between strings and patterns will decrease as the differences between the text string and query patterns increase (that is, as fewer string characters occur in the pattern) and as the pattern exhibits fewer repeating portions. Normally, the number of character mismatches is much larger than the number of character matches. In such a case, the number of character comparisons will be substantially less that the text string length, and the BM process will prove particularly efficient. A disadvantage of the BM process is that embedded "don't care" characters occurring in the middle or at the end of the query patterns are difficult to handle.



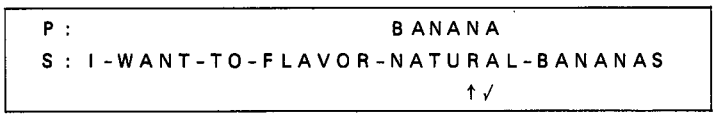
(a) T≠A and T does not occur in pattern : shift by pattern length.

(b) L≠A and L does not occur in pattern : shift by pattern length.



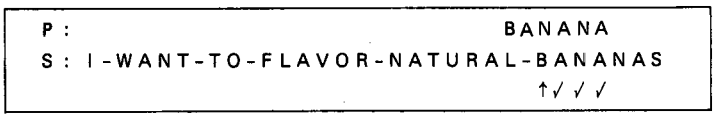
(c) N≠A, but N occurs in pattern; to bring N in coincidence with next occurrence in pattern, Δ_1 shift = 1.

(d) NA matches NA, but -≠A; next occurrence of NA in pattern is in positions 3 and 4 of pattern; to bring matching parts in coincidence Δ_2 shift = 2; since nonmatching string character does not occur in pattern, can shift pattern beyond -; that is Δ_1 shift = 4



(e) A matches A, but R≠N and R does not occur in pattern;
Δ_2 shift = 2 to reach next occurrence of A in pattern;
Δ_1 shift = 5 to shift beyond R.

(f) N≠A, Δ_1 shift = 1 to reach next occurrence of N in pattern.



(g) ANA = ANA but B≠N; for next occurrence of ANA in pattern Δ_2 shift = 2; for next occurrence of nonmatching string character B, Δ_1 shift = 2.

(h) Complete match.

Figure 8-26 Sample pattern match using BM process.

Don't care conditions occurring at the beginning of the queries do not, however, pose any special problems.

In operational environments, a decision on what process to choose for text matching purposes must depend on a comparison between the programmed string matching methods and the previously described special-purpose hardware devices or information retrieval machines that may be available for use.

4 SUMMARY

The purpose of this chapter has been to present actual information retrieval hardware devices and processing methods and to mention some new possibilities for information retrieval implementation in a changing technological environment. Technological changes have raised the possibility of greatly increased processing efficiencies. One of the most significant of these changes is the movement of some of the processing power to specialized data storage devices. That is, through the use of smart peripherals, the quantity of information that must be transferred from one storage device to another may be reduced. Information is examined where it is stored, and only that portion passing the initial screening is actually transferred for additional processing.

Second, there is interest in the development of specific devices for certain information retrieval tasks. Each step in the information retrieval process may be analyzed and special devices may help to carry out the individual steps. The result may be a series of special-purpose devices that collectively constitute a powerful retrieval system.

In addition to the special-purpose hardware, efficient data structures and file access procedures have been introduced which perform dictionary and index table searches in a dynamic file environment with a minimum of file rearrangement. The dynamic hashing and B-tree file search procedures are especially attractive in this connection.

Finally, new text matching methods implemented by hardware or by software programs can now be used for a fast scanning and detection of character patterns in running texts. These text scanning systems may in time be used to find answers to queries by a direct examination of the texts rather than by the conventional inverted file processing.

Whether these developments will find favor with the users and managers of retrieval systems ultimately depends on their economic viability as well as on their ease of use and effectiveness. One may expect results from practical tests of the new methodologies within the next few years.

REFERENCES

[1] W. Durant, The Story of Civilization: Part 1, Our Oriental Heritage, Simon and Schuster, New York, 1954.

[2] W.L. Schaaf, editor, Our Mathematical Heritage: Essays on the Cultural Significance of Mathematics, Collier Books, New York, 1963.

[3] M. Bohl, Information Processing, 2nd Edition, Science Research Associates, Chicago, Illinois, 1976.
[4] V. Bush, As We May Think, Atlantic Monthly, Vol. 176, No. 1, 1945, pp. 101–108.
[5] L.C. Smith, "MEMEX" as an Image of Potentiality in Information Retrieval Research and Development, Third International Information Retrieval Conference, Cambridge, England, June 1980.
[6] N. Knottek, Mini and Microcomputer Survey, Datamation, Vol. 24, No. 8, August 1978, pp. 113–124.
[7] T.C. Chen, Computer Technology and the Database User, IBM Corporation, Research Report RJ-2316, San Jose, California, August 1978.
[8] P.W. Williams, The Potential of the Microprocessor in Library and Information Work, Aslib Proceedings, Vol. 31, No. 4, April 1979, pp. 202–209.
[9] A.D. Pratt, The Use of Microcomputers in Libraries, Journal of Library Automation, Vol. 13, No. 1, March 1980, pp. 7–17.
[10] F.G. Withington, Beyond 1984: A Technology Forecast, Datamation, Vol. 21, No. 1, January 1975, pp. 54–73.
[11] B. Parhami, A Highly Parallel Computing System for Information Retrieval, Proceedings of the Fall Joint Computer Conference, AFIPS Press, Montvale, New Jersey, 1972, pp. 681–690.
[12] R.S. Rosenthal, The Data Management Machine—A Classification, Proceedings of Third Workshop on Computer Architecture for Non-Numeric Processing, Association for Computing Machinery, New York, May 1977, pp. 35–39.
[13] O. Bray and H.A. Freeman, Data Base Computers, Lexington Books, Lexington, Massachusetts, 1979.
[14] C.C. Foster, Computer Architecture, Encyclopedia of Computer Science, A. Ralston and C.L. Meek, editors, Petrocelli/Charter, New York, 1976, pp. 263–268.
[15] P.B. Berra, Data Base Machines, ACM SIGIR Forum, Vol. 12, No. 3, Winter 1977, pp. 4–22.
[16] K.E. Batcher, STARAN Series E, Proceedings of the 1977 International Conference on Parallel Processing, August 1977, pp. 140–143.
[17] A.L. Robinson, Array Processors: Maxi-Number Crunching for a Mini Price, Science, Vol. 203, 12 January 1979, pp. 156–160.
[18] C.N. Winningstad, Scientific Computing on a Budget, Datamation, Vol. 24, No. 10, October 1978, pp. 159–173.
[19] G. Salton and D. Bergmark, Parallel Computations in Information Retrieval, in Lecture Notes in Computer Science, Vol. 111, W. Handler, editor, Springer Verlag, Berlin-New York, 1981, pp. 328–342.
[20] G.P. Copeland, C.J. Lipovski, and S.Y.W. Su, The Architecture of CASSM: A Cellular System for Non-Numeric Processing, Proceedings of the First Annual Symposium on Computer Architecture, Association for Computing Machinery, New York, December 1973, pp. 121–125.
[21] P.J. Sadowski and S.A. Schuster, Exploiting Parallelism in a Relational Associative Processor, Proceedings of the Fourth Workshop on Computer Architecture for Non-Numeric Processing, Association for Computing Machinery, New York, August 1978, pp. 99–109.
[22] S.A. Schuster, H.B. Nguyen, E.A. Ozkarahan, and K.C. Smith, RAP2—An Associative Processor for Data Bases and Its Application, IEEE Transactions on Computers, Vol. C-28, No. 6, June 1979, pp. 446–458.
[23] S.A. Schuster, H.B. Nguyen, E.A. Ozkarahan, and K.C. Smith, RAP2—An Asso-

ciative Processor for Data Bases, Proceedings of the Fifth Annual Symposium on Computer Architecture, Association for Computing Machinery, New York, April 1978, pp. 52–59.

[24] D.K. Hsiao and K. Kannan, The Architecture of a Database Computer—A Summary, Proceedings of the Third Workshop on Computer Architecture for Non-Numeric Processing, Association for Computing Machinery, New York, May 1977, pp. 31–34.

[25] D.K. Hsiao, K. Kannan, and D.S. Kerr, Structure Memory Designs for a Database Computer, Proceedings of the ACM '77 National Conference, Association for Computing Machinery, New York, October 1977, pp. 343–350.

[26] L.A. Hollaar, Text Retrieval Computers, Computer, Vol. 12, No. 3, March 1979, pp. 40–50.

[27] L.A. Hollaar, A Design for a List Merging Network, IEEE Transactions on Computers, Vol. 28, No. 6, June 1979, pp. 406–413.

[28] W.H. Stellhorn, An Inverted File Processor for Information Retrieval, IEEE Transactions on Computers, Vol. C-26, No. 12, December 1977, pp. 1258–1267.

[29] S.E. Preece, Design for a Modular Query Pre-Processor System, Proceedings of the Annual Meeting of the American Society for Information Science, American Society for Information Science, Washington, DC., 1974.

[30] L.A. Hollaar and D.C. Roberts, Current Research into Specialized Processors for Text Information Retrieval, Proceedings of the 4th International Conference on Very Large Data Bases, September 1978, pp. 270–279.

[31] C.E. Price, Table Look-up Techniques, Computing Surveys, Vol. 3, No. 2, June 1971, pp. 49–65.

[32] W.A. Burkhard and R.M. Keller, Some Approaches to Best-Match File Searching, Communications of the ACM, Vol. 16, No. 4, April 1973, pp. 230–236.

[33] D.E. Knuth, The Art of Programming, Vol. 3: Searching and Sorting, Addison Wesley Publishing Company, Reading, Massachusetts, 1973.

[34] M. Scholl, New File Organizations Based in Dynamic Hashing, ACM Transactions on Database Systems, Vol. 6, No. 1, March 1981, pp. 194–211.

[35] R. Fagin, J. Nievergelt, N. Pippenger, and H.R. Strong, Extendible Hashing—A Fast Access Method for Dynamic Files, ACM Transactions on Database Systems, Vol. 4, No. 3, September 1979, pp. 315–344.

[36] P. A. Larson, Dynamic Hashing, BIT, Vol. 18, 1978, pp. 184–201.

[37] D. Comer, The Ubiquitous B-Tree, Computing Surveys, Vol. 11, No. 2, June 1979, pp. 121–137.

[38] J. Nievergelt, Binary Search Trees and File Organization, Computing Surveys, Vol. 6, No. 3, September 1974, pp. 195–207.

[39] R. Bayer and E. McCreight, Organization and Maintenance of Large Ordered Indexes, Acta Informatica, Vol. 1, No. 3, 1972, pp. 173–189.

[40] E.H. Sussenguth, Jr., The Use of Tree Structures for Processing Files, Communications of the ACM, Vol. 6, No. 5, May 1963, pp. 272–279.

[41] E. Horowitz and S. Sahni, Fundamentals of Data Structures, Computer Science Press, Woodland Hills, California, 1976.

[42] V.Y. Lam, Multiattribute Retrieval with Combined Indexes, Communications of the ACM, Vol. 13, No. 11, November 1970, pp. 660–665.

[43] A. El Masri, J. Rohmer, and D. Tusera, A Machine for Information Retrieval, Proceedings of the Fourth Workshop on Computer Architecture for Non-Numeric Processing, Association for Computing Machinery, New York, August 1978, pp. 117–120.

[44] D.C. Roberts, A Specialized Computer Architecture for High Speed Text Searching, Second Workshop on Computer Architecture for Non-Numeric Processing, Association for Computer Machinery, New York, 1976.
[45] R.M. Bird, J.B. Newsbaum, and J.L. Trefftzs, Text File Inversion: An Evaluation, Proceedings of the Fourth Workshop on Computer Architecture for Non-Numeric Processing, Association for Computing Machinery, New York, August 1972, pp. 42–50.
[46] R.M. Bird, J.C. Tu, and R.M. Worthy, Associative/Parallel Processors for Searching Very Large Textual Data Bases, Proceedings of the Third Workshop on Computer Architecture for Non-Numeric Processing, Association for Computing Machinery, New York, May 1977, pp. 8–16.
[47] D.C. Roberts, A Specialized Computer Architecture for Text Retrieval, Proceedings of the Fourth Workshop on Computer Architecture for Non-Numeric Processing, Association for Computing Machinery, New York, August 1978, pp. 51–59.
[48] A.V. Aho and M.J. Corasick, Efficient String Matching: An Aid to Bibliographic Search, Communications of the ACM, Vol. 18, No. 6, June 1975, pp. 333–340.
[49] D.E. Knuth, J.H. Morris, and V.R. Pratt, Fast Pattern Matching in Strings, SIAM Journal of Computing, Vol. 6, No. 2, June 1977, pp. 323–350.
[50] R.S. Boyer and J.S. Moore, A Fast String Searching Algorithm, Communications of the ACM, Vol. 20, No. 10, October 1977, pp. 762–772.
[51] R. Haskin, Hardware for Searching Very Large Text Databases, Proceedings of Fifth Workshop on Computer Architecture for Non-Numeric Processing, Association for Computing Machinery, New York, March 1980, pp. 49–56.
[52] G. Salton, Automatic Information Retrieval, Computer, Vol. 13, No. 9, September 1980. pp. 41–56.

BIBLIOGRAPHIC REMARKS

The following texts contain introductory descriptions of information technologies:

M.Bohl, Information Processing, 2nd Edition, Science Research Associates, Chicago, Illinois, 1976.This is a good introduction to computer systems, providing a large number of illustrations for easy visualization of the various devices and their uses.
J.G. Burch, Jr., and F.R. Strater, Jr., Information Systems—Theory and Practice, 2nd Edition, Hamilton Wiley Company, Santa Barbara, California, 1979. The appendix to this text provides an excellent short introduction to information technologies.

More advanced descriptions of computer hardware are included in:

D.P. Siewiorek, C.G. Bell, and A. Newell, Computer Structures: Principles and Examples, McGraw-Hill Book Company, New York, 1982, 926 pages.
H.W. Gschwind and E.J. McCluskey, Design of Digital Computers—An Introduction, 2nd Edition, Springer Verlag, New York, 1975.
G.A. Korn, Minicomputers for Engineers and Scientists, McGraw-Hill Book Company, New York, 1973, 303 pages.

The development of information technologies for applications such as information retrieval is often covered in workshop proceedings such as those on Computer Architecture for Non-Numeric Processing published by the Association for Computing Machinery. Many of the developments from these conferences are synthesized in a single text:

O. Bray and H.A. Freeman, Data Base Computers, Lexington Books, Lexington, Massachusetts, 1979.

A variety of journals regularly cover new hardware development, including in particular the journals published by the Computer Society of the IEEE (Institute of Electrical and Electronics Engineers). Of particular interest is the readable *Computer,* and the more technical *IEEE Transactions on Computers.* Software procedures useful for dictionary access and text processing often appear in literature published by the Association for Computing Machinery, notably the *Communications of the ACM* and the *ACM Transactions on Database Systems.*

EXERCISES

8-1 In information processing the main file characteristics, such as file size and record contents, normally determine the storage medium used to maintain the records as well as the file access methods. Thus, a file of 100 bibliographic records each described by author name, document title, journal name, volume number, date of publication, pagination, and short abstract may best be accommodated on a set of 3 by 5 index cards that are manually searched to find a given item. Card storage is inexpensive; the file is easily updated, and rapid file searches are possible when the file size is small. As the file grows, a manually accessed card file necessarily becomes less advantageous. Describe the desirable characteristics of an automatic device capable of storing and manipulating each of the following files:

- **a** A file of 1,000 bibliographic records organized sequentially according to the alphabetic ordering of the title. The file is moderately active—that is, a fair number of searches are conducted; but the volatility is low—there are few additions and deletions.
- **b** A file of 1,000,000 records stored in random order. The file is very active but of low volatility.
- **c** A file of 1,000,000 records stored in random order.The file is active as well as volatile.
- **d** A file of 1,000,000,000 records stored in random order. The file is moderately active, but with relatively few additions and deletions.

8-2 As the technology used to store a given file of records changes, conversion costs are incurred in the changeover from one medium to another. For example, the conversion of a manual file into a computer-based file may entail a costly keypunching operation to transfer the original information onto punched cards from where the information can in turn be transferred onto tape or disk. Identify the problems and costs incurred in the following changes of storage technology for a given file of records:

a Magnetic tape to magnetic disk
b Magnetic disk to associative array storage
c Magnetic disk to data base computer

8-3 Changes in storage technology may produce advantages in the accessing and file updating operations. Identify the benefits obtained from the file conversions listed in Exercise 8-2.

8-4 Describe in flowchart form the complete process used to answer the following queries:

(1) TERM A AND TERM B
(2) TERM A NOT TERM B
(3) TERM A WITH TERM B (WITH ≡ in same sentence as)

Assume that the following storage technology is used to store the corresponding file of records:

a An associative array processor such as the Staran
b The data base computer

8-5 Three main characteristics may be identified that are of primary importance in information retrieval:

a The file searches should be efficient as well as effective.
b The user interface should permit a relatively effortless interaction between user and system.
c The storage medium should accommodate files of substantial size.

Describe methods and techniques capable of accommodating these various aims. Are there limitations indicating that the stated requirements cannot all be met simultaneously by a single system?

8-6 Consider three retrieval systems with the following sets of keywords:

a BAT, CAT, HAT, MAT, SAT
b FAIR, FAITH, FALL, FAME, FAN, FANCY
c HE, SHE, HER, HERE, THERE, SHEAR

For each keyword system, construct a system of finite state automata capable of recognizing the various keywords in a text scanning system. Can you think of a method whereby a single set of states is used to store repeating portions of the various keywords—for example, the ending AT in part a or the sequence FA in part b?

8-7 Use the Boyer-Moore method to compare the keyword FANCY with the document title "FANATIC FANNY FARMER STRUCK MY FANCY." Show all steps and explain each of the required character shifts. How many character comparisons are needed to obtain a complete match?