Four
FILE STRUCTURES

Introduction

This chapter is mainly concerned with the way in which file structures are used in document retrieval. Most surveys of file structures address themselves to applications in data management which is reflected in the terminology used to describe the basic concepts. I shall (on the whole) follow Hsiao and Harary[1] whose terminology is perhaps slightly non-standard but emphasises the logical nature of file structures. A further advantage is that it enables me to bridge the gap between data management and document retrieval easily. A few other good references on file structures are Roberts[2], Bertziss[3], Dodd[4], and Climenson[5].

Basic terminology

Given a set of 'attributes' A and a set of 'values' V, then a *record* R is a subset of the cartesian product $A \times V$ in which each attribute has one and only one value. Thus R is a set of ordered pairs of the form (an attribute, its value). For example, the record for a document which has been processed by an automatic content analysis algorithm would be

$$R = \{(K_1, x_1), (K_2, x_2), \ldots (K_m, x_m)\}$$

The K_i's are keywords functioning as attributes and the value x_i can be thought of as a numerical weight. Frequently documents are simply characterised by the absence or presence of keywords, in which case we write

$$R = \{K_{t_1}, K_{t_2}, \ldots, K_{t_i}\}$$

where K_{t_i} is present if $x_{t_i} = 1$ and is absent otherwise.

Records are collected into logical units called *files*. They enable one to refer to a set of records by name, the *file name*. The records within a file are often organised according to relationships between the records. This logical organisation has become known as a *file structure* (or data structure).

It is difficult in describing file structures to keep the logical features separate from the physical ones. The latter are characteristics forced upon us by the recording media (e.g. tape, disk). Some features can be defined abstractly (with little gain) but are more easily understood when illustrated concretely. One such feature is a *field*. In any implementation of a record, the attribute values are usually positional, that is the identity of an attribute is given by the position of its attribute value within the record. Therefore the data within a record is registered sequentially and has a definite beginning and end. The record is said to be divided into *fields* and the nth field carries the nth attribute value. Pictorially we have an example of a record with associated fields in *Figure 4.1*.

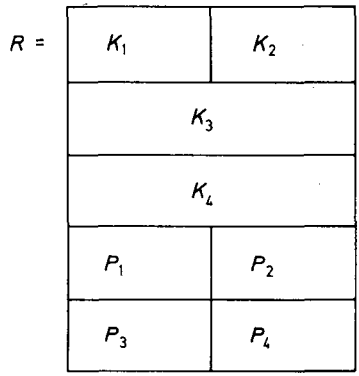


Figure 4.1. An example of a record with associated fields

The fields are not necessarily constant in length. To find the value of the attribute K_4, we first find the address of the record R (which is actually the address of the start of the record) and read the data in the 4th field.

In the same picture I have also shown some fields labelled P_i. They are addresses of other records, and are commonly called *pointers*. Now we have extended the definition of a record to a set of attribute-value pairs *and* pointers. Each pointer is usually associated with a particular

attribute-value pair. For example, (see *Figure 4.2*) pointers could be used to link all records for which the value x_1 (of attribute K_1) is a, similarly for x_2 equal to b, etc.

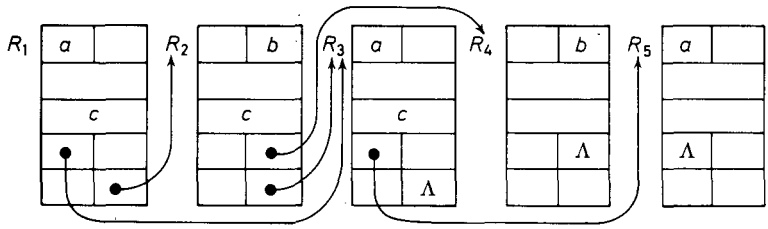


Figure 4.2. A demonstration of the use of pointers to link records

To indicate that a record is the last record pointed to in a list of records we use the *null pointer* Λ. The pointer associated with attribute K in record R will be called a *K-pointer.* An attribute (keyword) that is used in this way to organise a file is called a *key.*

To unify the discussion of file structures we need some further concepts. Following Hsiao and Harary again, we define a *list* L of records with respect to a keyword K, or more briefly a *K-list* as a set of records containing K such that;

(1) the K-pointers are distinct;
(2) each non-null K-pointer in L gives the address of a record within L;
(3) there is a unique record in L not pointed to by any record containing K; it is called the *beginning* of the list; and
(4) there is a unique record in L containing the null K-pointer; it is the *end* of the list.

(Hsiao and Harary state condition (2) slightly differently so that no two K-lists have a record in common; this only appears to complicate things.)

From our previous example:

$$K_1\text{-list} : R_1, R_3, R_5$$

$$K_2\text{-list} : R_2, R_4$$

$$K_4\text{-list} : R_1, R_2, R_3$$

Finally, we need the definition of a *directory* of a file. Let F be a file whose records contain just m different keywords $K_1, K_2, \ldots, K_m$. Let n_i be the number of records containing the keyword K_i, and h_i be the

number of K_i-lists in F. Furthermore, we denote by a_{ij} the beginning address of the jth K_i-list. Then the *directory* is the set of sequences

$$(K_i, n_i, h_i, a_{i1}, a_{i2}, \ldots a_{ih_i}) \quad i = 1, 2, \ldots m$$

We are now in a position to give a unified treatment of sequential files, inverted files, index-sequential files and multi-list files.

Sequential files

A sequential file is the most primitive of all file structures. It has *no* directory and *no* linking pointers. The records are generally organised in lexicographic order on the value of some key. In other words, a particular attribute is chosen whose value will determine the order of the records. Sometimes when the attribute value is constant for a large number of records a second key is chosen to give an order when the first key fails to discriminate.

The implementation of this file structure requires the use of a sorting routine.

Its main advantages are:

(1) it is easy to implement;
(2) it provides fast access to the next record using lexicographic order.

Its disadvantages:

(1) it is difficult to update – inserting a new record may require moving a large proportion of the file;
(2) random access is extremely slow.

Sometimes a file is considered to be sequentially organised despite the fact that it is *not* ordered according to any key. Perhaps the date of acquisition is considered to be the key value, the newest entries are added to the end of the file.

Inverted files

The importance of this file structure will become more apparent when Boolean Searches are discussed in the next chapter. For the moment we limit ourselves to describing its structure.

An *inverted file* is a file structure in which every list contains only one record. Remember that a list is defined with respect to a keyword K, so every K-list contains only one record. This implies that the directory will be such that $n_i = h_i$ for all i, that is, the number of records containing K_i will equal the number of K_i-lists. So the directory

will have an address for each record containing K_i. For document retrieval this means that given a keyword we can immediately locate the addresses of all the documents containing that keyword. For the previous example let us assume that a non-blank entry in the field corresponding to an attribute indicates the presence of a keyword and a blank entry its absence. Then the directory will point to the file in the way shown in *Figure 4.3.* The definition of an inverted file does *not*

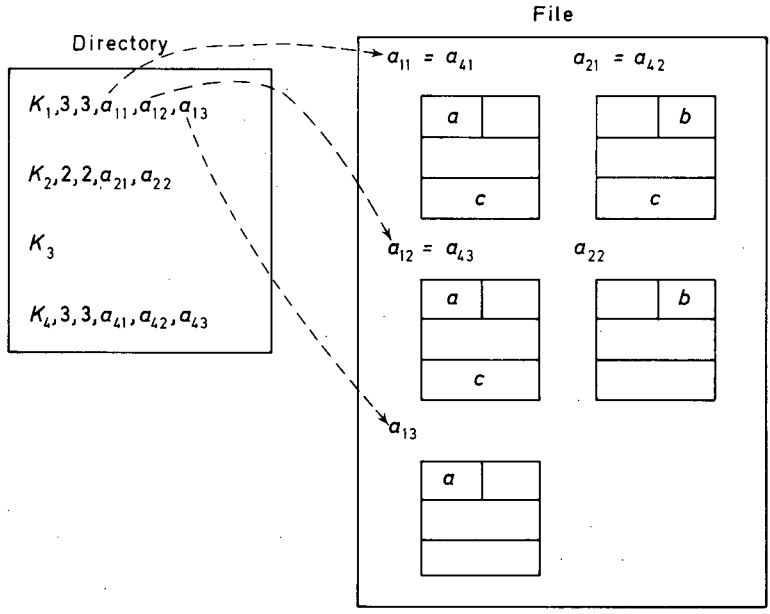


Figure 4.3. An inverted file

require that the addresses in the directory are in any order. However, to facilitate operations such as conjunction ('and') and disjunction ('or') on any two inverted lists, the addresses are normally kept in record number order. This means that 'and' and 'or' operations can be performed with one pass through both lists. The penalty we pay is of course that the inverted file becomes slower to update.

Index-sequential files

An index-sequential file is an inverted file in which for every keyword K_i, we have $n_i = h_i = 1$ and $a_{11} < a_{21} \ldots < a_{m1}$. This situation can only arise if each record has just one unique keyword, or one unique

attribute-value. In practice therefore, this set of records may be ordered sequentially by a key. Each key value appears in the directory with the associated address of its record. An obvious interpretation of a key of this kind would be the record number. In our example none of the attributes would do the job except the record number. Diagrammatically the index-sequential file would therefore appear as shown in *Figure 4.4*. I have deliberately written R_i instead of K_i to emphasise the nature of the key.

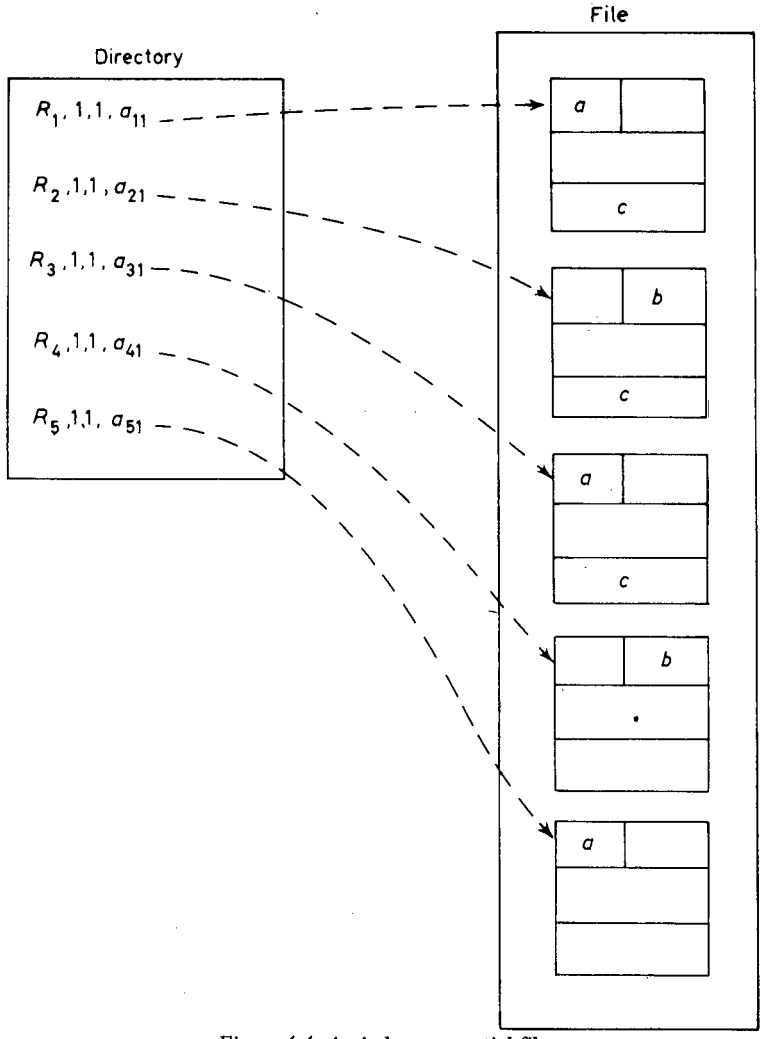


Figure 4.4. An index-sequential file

In the literature an index-sequential file is usually thought of as a sequential file with a hierarchy of indices. This does not contradict the previous definition, it merely describes the way in which the directory is implemented. It is not surprising therefore that the indexes

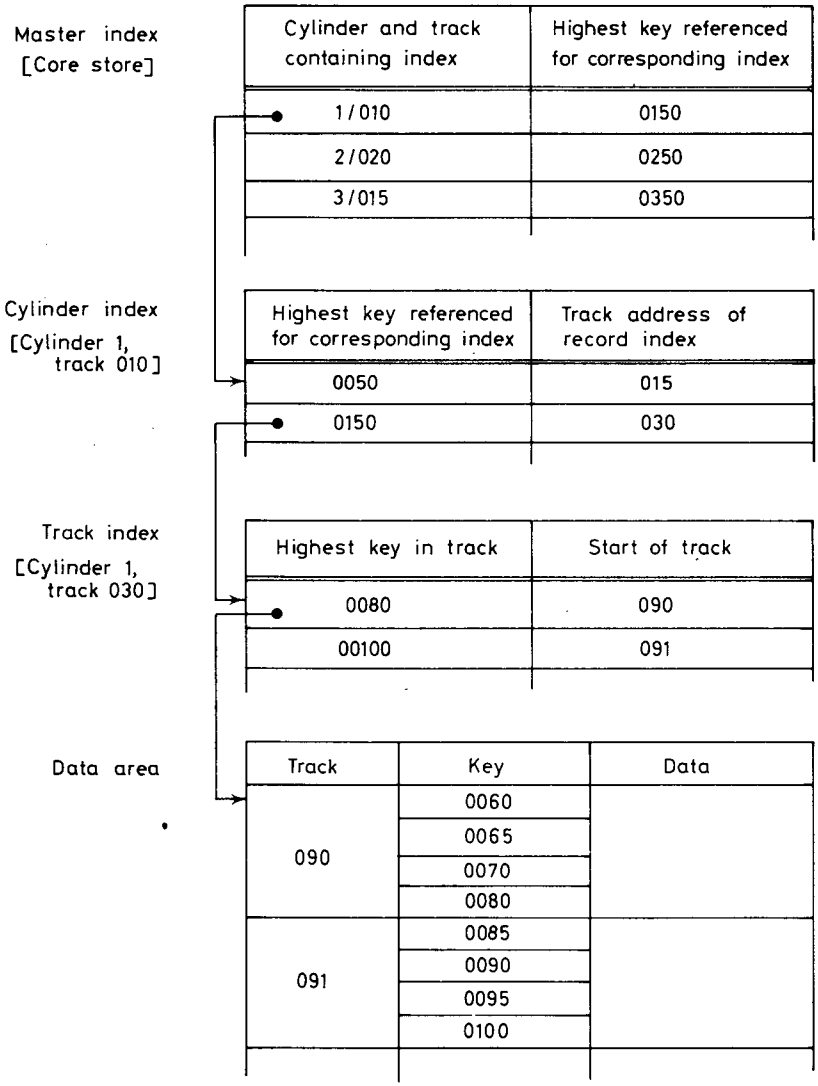


Figure 4.5. An example of an implementation of an index-sequential file (Adapted from D. R. Judd, The Use of Files, *Macdonald and Elsevier, London and New York, 1973, page 46)*

('index' = 'directory' here) are often oriented to the characteristics of the storage medium. For example (see *Figure 4.5*) there might be three levels of indexing: track, cylinder and master. Each entry in the track index will contain enough information to locate the start of the track, and the key of the last record in the track, which is also normally the highest value on that track. There is a track index for each cylinder. Each entry in the cylinder index gives the last record on each cylinder and the address of the track index for that cylinder. If the cylinder index itself is stored on tracks, then the master index will give the highest key referenced for each track of the cylinder index and the starting address of that track.

No mention has been made of the possibility of overflow during an updating process. Normally provision is made in the directory to administer an overflow area. This of course increases the number of book-keeping entries in each entry of the index.

Multi-lists

A multi-list is really only a slightly modified inverted file. There is one list per keyword, i.e. $h_i = 1$. The records containing a particular keyword K_i are chained together to form the K_i-list and the start of the K_i-list is given in the directory, as illustrated in *Figure 4.6*. Since there is no K_3-list the field reserved for its pointer could well have been omitted. So could any blank pointer field, so long as no ambiguity arises as to which pointer belongs to which keyword. One way of ensuring this, particularly if the data values (attribute-values) are fixed format, is to have the pointer not pointing to the beginning of the record but pointing to the location of the next pointer in the chain.

The multi-list is designed to overcome the difficulties of updating an inverted file. The addresses in the directory of an inverted file are normally kept in record-number order. But, when the time comes to add a new record to the file, this sequence must be maintained, and inserting the new address can be expensive. No such problem arises with the multi-list, we update the appropriate K-lists by simply chaining in the new record. The penalty we pay for this is of course the increase in search time. This is in fact typical of many of the file structures. Inherent in their design is a trade-off between search time and update time.

Cellular multi-lists

A further modification of the multi-list is inspired by the fact that many storage media are divided into *pages,* which can be retrieved one

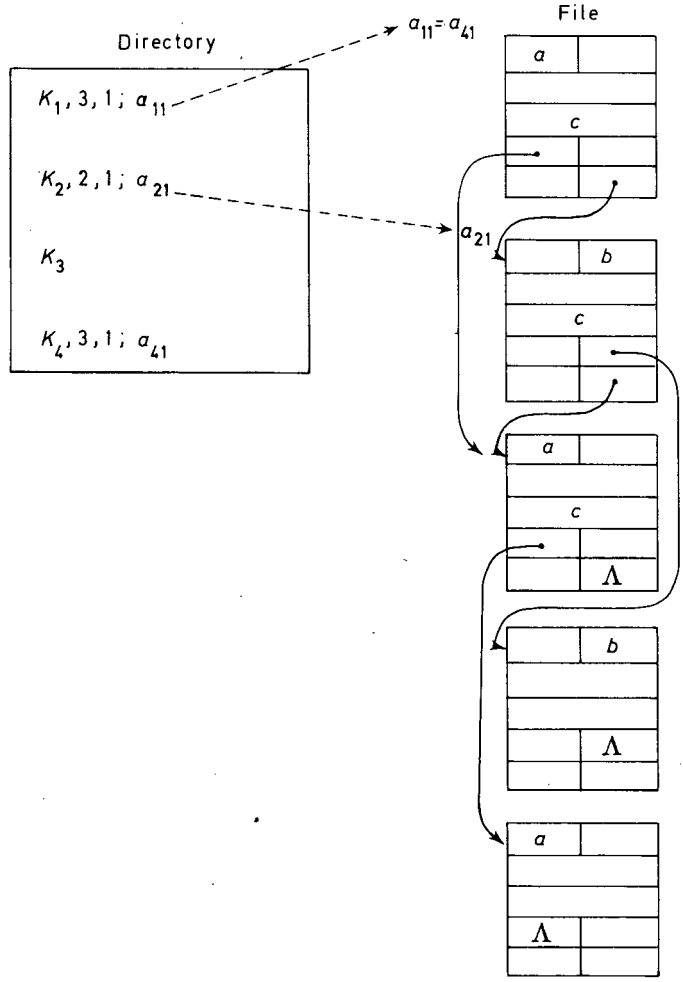


Figure 4.6. A multi-list

at a time. A K-list may cross several page boundaries which means that several pages may have to be accessed to retrieve one record. A modified multi-list structure which avoids this is called a *cellular multi-list.* The K-lists are limited so that they will not cross the page (cell) boundaries.

At this point the full power of the notation introduced before comes into play. The directory for a cellular multi-list will be the set of sequences

$$(K_i, n_i, h_i, a_{i1}, \ldots, a_{ih_i}) \quad i = 1, 2, \ldots, m$$

where the h_i have been picked to ensure that a K_i-list does not cross a page boundary. In an implementation, just as in the implementation of an index-sequential file, further information will be stored with each address to enable the right page to be located for each key value.

Ring structures

A *ring* is simply a linear list that closes upon itself. In terms of the definition of a K-list, the beginning and end of the list are the same record. This data-structure is particularly useful to show classification of data.

Let us suppose that a set of documents

$$\{D_1, D_2, D_3, D_4, D_5, D_6, D_7, D_8\}$$

has been classified into four groups, that is

$$\{(D_1, D_2), (D_3, D_4), (D_5, D_6), (D_7, D_8)\}$$

Furthermore these have themselves been classified into two groups,

$$\{((D_1, D_2), (D_3, D_4)), ((D_5, D_6), (D_7, D_8))\}$$

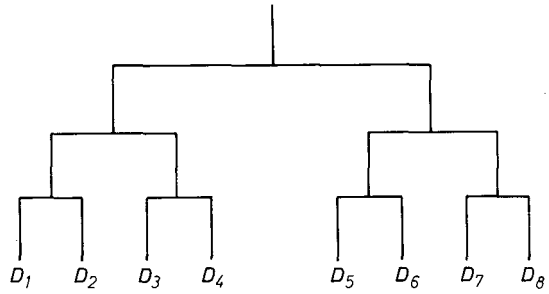


Figure 4.7. A dendrogram

The dendrogram for this structure would be that given in *Figure 4.7.* To represent this in storage by means of ring structures is now a simple matter (see *Figure 4.8*).

The D_i indicates a description (representation) of a document. Notice how the rings at a lower level are contained in those at a higher level. The field marked C_i normally contains some identifying information with respect to the ring it subsumes. For example, C_1 in some way identifies the class of documents $\{D_1, D_2\}$.

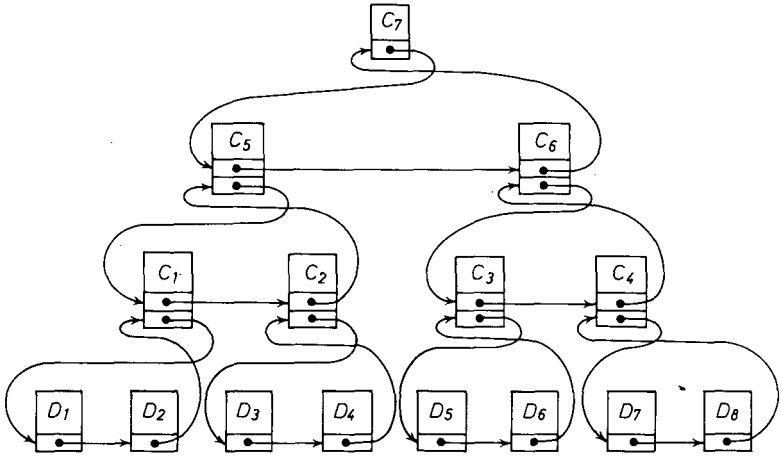


Figure 4.8. An implementation of a dendrogram via ring structures

Were we to group documents according to the keywords they shared, then for each keyword we would have a group of documents, namely, those which had that keyword in common. C_i would then be the field containing the keyword uniting that particular group. The rings would of course overlap (*Figure 4.9*), as in this example:

$$D_1 = \{K_1, K_2\}$$

$$D_2 = \{K_2, K_3\}$$

$$D_3 = \{K_1, K_4\}$$

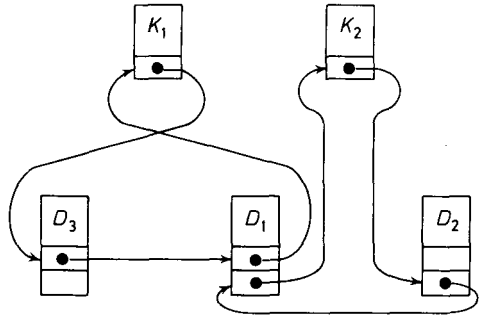


Figure 4.9. Two overlapping rings

The usefulness of this kind of structure will become more apparent when we discuss searching of classifications. If each ring has associated with it a record which contains identifying information for its members, then, a search strategy searching a structure such as this will first look at C_i (or K_i in the second example) to determine whether to proceed or abandon the search.

Threaded lists

In this section an elementary knowledge of list processing will be assumed. Readers who are unfamiliar with this topic should consult the little book by Foster[6].

A simple list representation of the classification

$$((D_1, D_2), (D_3, D_4)), ((D_5, D_6), (D_7, D_8))$$

is given in *Figure 4.10.* Each sublist in this structure has associated with it a record containing *only* two pointers. (We can assume that D_i is really a pointer to document D_i.) The function of the pointers should be clear from the diagram. The main thing to note, however, is that the record associated with a list does *not* contain any identifying information.

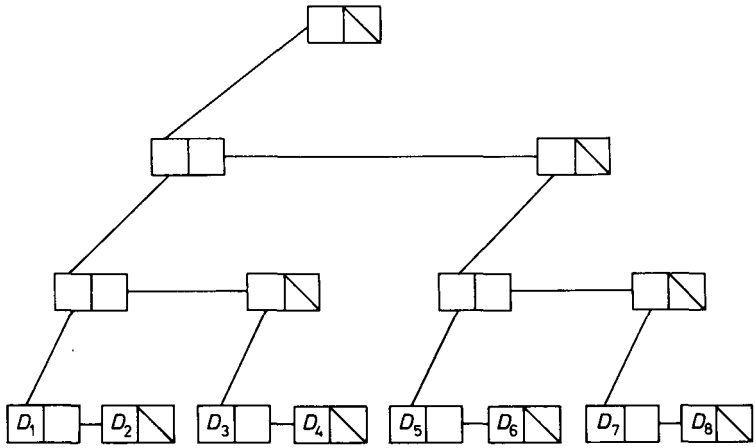


Figure 4.10. A list structure implementation of a hierarchic classification

A modification of the implementation of a list structure like this which makes it resemble a set of ring structures is to make the right hand pointer of the *last* element of a sublist point back to the head of

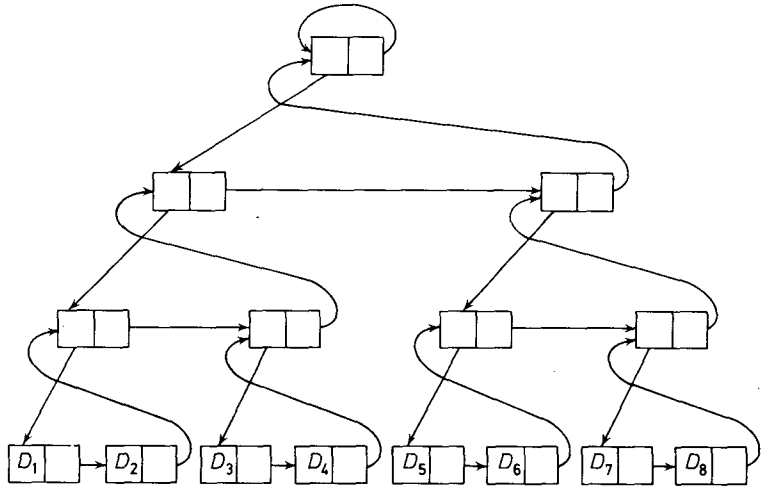
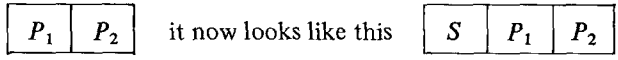


Figure 4.11. A threaded list implementation of a hierarchic classification

the sublist. Each sublist has become effectively a ring structure. We now have what is commonly called a *threaded list* (see *Figure 4.11*). The representation I have given is a slight oversimplification in that we need to flag which elements are data elements (giving access to the documents D_i) and which elements are just pointer elements. The major advantage associated with a threaded list is that it can be traversed without the aid of a stack. Normally when traversing a conventional list structure the return addresses are stacked, whereas in the threaded list they have been incorporated in the data structure.

One disadvantage associated with the use of list and ring structures for representing classifications is that they can only be entered at the 'top'. An additional index giving entry to the structure at each of the data elements increases the update speed considerably.

Another modification of the simple list representation has been studied extensively by Stanfel[7,8] and Patt[9]. The individual elements (or cells) of the list structure are modified to incorporate one extra field, so that instead of each element looking like this



where the P_i's are pointers and S is a symbol. Otherwise no essential change has been made to the simple representation. This structure has become known as the *Doubly Chained Tree.* Its properties have mainly been investigated for storing variable length keys, where each key is made up by selecting symbols from a finite (usually small) alphabet.

For example, let $\{A,B,C\}$ be the set of key symbols and let R_1, R_2, R_3, R_4, R_5 be five records to be stored. Let us assign keys made of the 3 symbols, to the records as follows:

AAA	R_1
AB	R_2
AC	R_3
BB	R_4
BC	R_5

An example of a doubly chained tree containing the keys and giving access to the records is given in *Figure 4.12.* The topmost element contains no symbol, it merely functions as the start of the structure. Given an arbitrary key its presence or absence is detected by matching

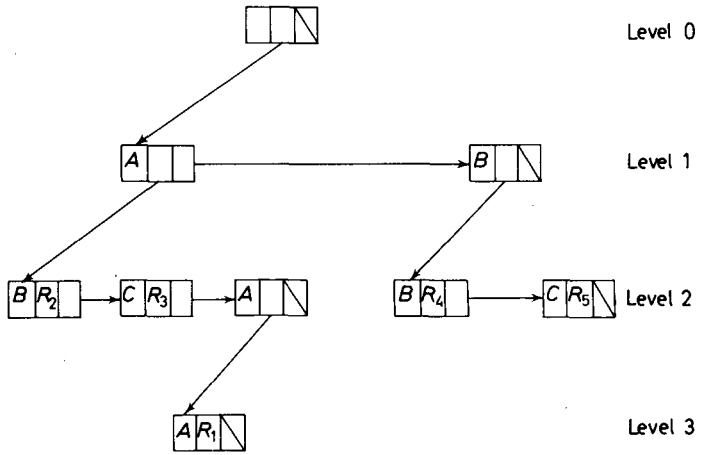


Figure 4.12. An example of a doubly chained tree

it against keys in the structure. Matching proceeds level by level, once a matching symbol has been found at one level, the P_1 pointer is followed to the set of *alternative* symbols at the next level down. The matching will terminate either:

(1) when the key is exhausted, that is, no more key symbols are left to match; or
(2) when no matching symbol is found at the current level.

For case (1) we have:

(a) the key is present if the P_1 pointer in the same cell as the last matching symbol now points to a record;
(b) P_1 points to a further symbol, that is, the key 'falls short' and is therefore not in the structure.

For case (2), we also have that the key is not in the structure, but now there is a mismatch.

Stanfel and Patt have concentrated on generating search trees with minimum expected search time, and preserving this property despite updating. For the detailed mathematics demonstrating that this is possible the reader is referred to their cited work.

Trees

Although computer scientists have adopted trees as file structures, their properties were originally investigated by mathematicians. In fact a substantial part of the *Theory of Graphs* is devoted to the study of trees. Excellent books on the mathematical aspects of trees (and graphs) have been written by Berge[10], Harary *et al.*,[11] and Ore[12]. Harary's book also contains a useful glossary of concepts in graph theory. In addition Bertziss[3] and Knuth[13] discuss topics in graph theory with applications in information processing.

There are numerous definitions of *trees*. I have chosen a particularly simple one from Berge. If we think of a *graph* as a set of *nodes* (or points or vertices) and a set of *lines* (or edges) such that each line connects exactly two nodes, then a *tree* is defined to be a finite connected graph with no cycles, and possessing at least two nodes. To define a cycle we first define a chain. We represent the line u_k joining two nodes x and y by $u_k = [x,y]$. A *chain* is a *sequence* of lines, in which each line u_k has one node in common with the preceding line u_{k-1}, and the other vertex in common with the succeeding line u_{k+1}. An example of a chain is $[a,x_1]$, $[x_1,x_2]$, $[x_2,x_3]$, $[x_3,b]$. A *cycle* is a finite chain which begins at a node and terminates at the same node (i.e. in the example $a = b$).

Berge gives the following theorem showing many equivalent characterisations of trees.

Theorem. Let H be a graph with at least n nodes, where $n > 1$; any one of the following equivalent properties characterises a tree.

(1) H is connected and does not possess any cycles.
(2) H contains no cycles and has $n - 1$ lines.
(3) H is connected and has $n - 1$ lines.

(4) H is connected but loses this property if any line is deleted.
(5) Every pair of nodes is connected by one and only one chain.

One thing to be noticed in the discussion so far is that no mention has been made of a *direction* associated with a line. In most applications in computer science (and IR) one node is singled out as special. This node is normally called the *root* of the tree, and every other node in the tree can only be reached by starting at the root and proceeding along a chain of lines until the node sought is reached. Implicitly therefore, a direction is associated with each line. In fact, when one comes to represent a tree inside a computer by a list structure, often the addresses are stored in a way which allows movement in only one direction. It is convenient to think of a tree as a *directed* graph with a reserved node as the root of the tree. Of course, if one has a root then each path (directed chain) starting at the root will eventually terminate at a particular node from which no further branches emerge. These nodes are called the *terminal* nodes of the tree.

By now it is perhaps apparent that when we were talking about ring structures and threaded lists in some of our examples we were really demonstrating how to implement a tree structure. The dendrogram in *Figure 4.7* can easily be represented as a tree (*Figure 4.13*). The documents are stored at the terminal nodes and each node represents a class (cluster) of documents. A search for a particular set of documents would be initiated at the root and would proceed along the arrows until the required class was found.

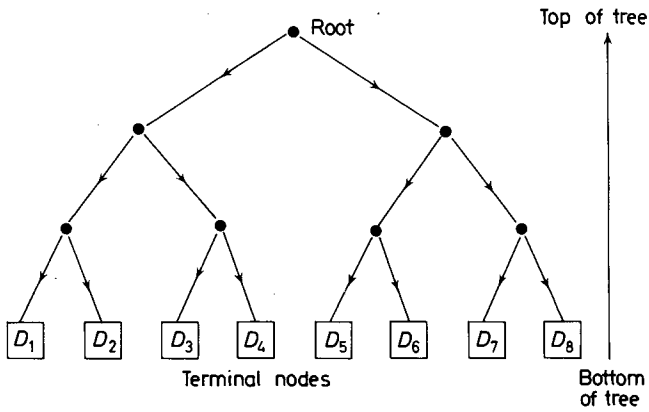


Figure 4.13. A tree representation of a dendrogram

Another example of a tree structure is the directory associated with an index-sequential file. It was described as a hierarchy of indexes, but could equally well have been described as a tree structure.

The use of tree structures in computer science dates back to the early 1950s when it was realised that the so-called *binary search* could readily be represented by a *binary* tree. A binary tree is one in which each node (except the terminal nodes) has exactly two branches leaving it. A binary search is an efficient method for detecting the presence or absence of a key value among a set of keys. It presupposes that the keys have been sorted. It proceeds by successive division of the set, at each division discarding half the current set as not containing the sought key. When the set contains N sorted keys the search time is of order $\log_2 N$. Furthermore, after some thought one can see how this process can be simply represented by a binary tree.

Unfortunately, in many applications one wants the ability to *insert* a key which has been found to be absent. If the keys are stored sequentially then the time taken by the insertion operation may be of order N. If one, however, *stores* the keys in a binary tree this lengthy insert time may be overcome, both search and insert time will be of order $\log_2 N$. The keys are stored at the nodes, at each node a left branch will lead to 'smaller' keys, a right branch will lead to 'greater' keys. A search terminating on a terminal node will indicate that the key is not present and will need to be inserted.

The structure of the tree as it grows is largely dependent on the order in which new keys are presented. Search time may become unnecessarily long because of the lop-sidedness of the tree. Fortunately, it can be shown (Knuth[14]) that random insertions do not change the expected $\log_2 N$ time dependence of the tree search. Nevertheless, methods are available to prevent the possibility of *degenerate trees.* These are trees in which the keys are stored in such a way that the expected search time is far from optimal. For example, if the keys were to arrive for insertion already ordered then the tree to be built would simply be as shown in *Figure 4.14.*

It would take us too far afield for me to explain the techniques for avoiding degenerate trees. Essentially, the binary tree is maintained in such a way that at any node the subtree on the left branch has approximately as many levels as the subtree on the right branch. Hence the name *balanced* tree for such a tree. The search paths in a balanced tree will never be more than 45 per cent longer than the optimum. The expected search and insert times are still of order $\log N$. For further details the reader is recommended to consult Knuth[14].

So far we have assumed that each key was equally likely as a search argument. If one has data giving the probability that the search argument is K_i (a key already in the tree), and the probability that the

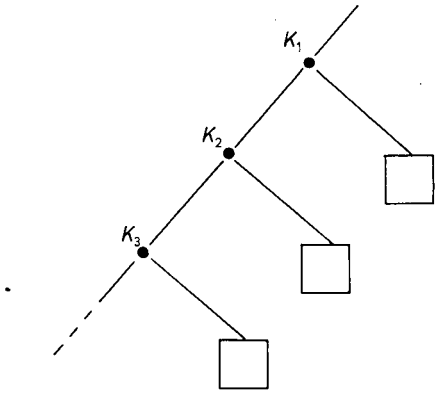


Figure 4.14. An example of a degenerate tree

search argument lies between K_i and K_{i+1}, then again techniques are known for reordering the tree to optimise the expected search time. Essentially one makes sure that the more frequently accessed keys have the shortest search paths from the root. One well-known technique used when only the *second* set of probabilities is known, and the others assigned the value zero, is the Hu-Tucker algorithm. Again the interested reader may consult Knuth.

At this point it is probably a good idea to point out that these efficiency considerations are largely irrelevant when it comes to representing a document classification by a tree structure. The situation in document retrieval is different in the following aspects:

(1) we do not have a useful *linear* ordering on the documents;
(2) a search request normally does not seek the absence or presence of a document.

In fact, what we do have is that documents are more or less similar to each other, and a request seeks documents which in some way best match the request. A tree structure representing a document classification is therefore chosen so that similar documents may be close together. Therefore to rearrange a tree structure to satisfy some 'balancedness' criterion is out of the question. The search efficiency is achieved by bringing together documents which are likely to be required together.

This is not to say that the above efficiency considerations are unimportant in the general context of IR. Many operations, such as the searching of a dictionary, and using a suffix stripping algorithm can be made very efficient by appropriately structuring the binary tree.

The discussion so far has been limited to *binary* trees. In many applications this two-way split is inappropriate. The natural way to represent document classifications is by a general tree structure, where there is no restriction on the number of branches leaving a node. Another example is the directory of an index sequential file which is normally represented by an m-way tree, where m is the number of branches leaving a node.

Finally, more comments are in order about the manipulation of tree structures in mass storage devices. Up to now we have assumed that to follow a set of pointers poses no particular problems with regard to retrieval speed. Unfortunately, present random access devices are sufficiently slow for it to be impossible to allow an access for, say, each node in a tree. There are ways of partitioning trees in such a way that the number of disk accesses during a tree search can be reduced. Essentially, it involves storing a number of nodes together in one 'page' of disk storage. During a disk access this page is brought into fast memory, is then searched, and the next page to be accessed is determined.

Scatter storage or hash addressing

One file structure which does not relate very well to the ones mentioned before is known as *Scatter Storage.* The technique by which the file structure is implemented is often called *Hash Addressing.* Its underlying principle is appealingly simple. Given that we may access the data through a number of keys K_i, then the address of the data in store is located through a key transformation function f which when applied to K_i evaluates to give the address of the associated data. We are assuming here that with each key is associated only one data item. Also for convenience we will assume that each record (data and key) fits into one location, whose address is in the image space of f. The addresses given by the application of f to the keys K_i are called the *hash addresses* and f is called a *hashing function.* Ideally f should be such that it spreads the hash addresses uniformly over the available storage. Of course this would be achieved if the function were one-to-one. Unfortunately this cannot be so because the range of possible key values is usually considerably larger than the range of the available storage addresses. Therefore, given any hashing function we have to contend with the fact that two distinct keys K_i and K_j are likely to map to the same address $f(K_i)$ $(=f(K_j))$. Before I explain some of the ways of dealing with this I shall give a few examples of hashing functions.

Let us assume that the available storage is of size 2^m then three simple transformations are as follows:

(1) if K_i is the key, then take the square of its binary representation and select m bits from the middle of the result;
(2) cut the binary representation of K_i into pieces each of m bits and add these together. Now select the m least significant bits of the sum as the hash address;
(3) divide the integer corresponding to K_i by the length of the available store 2^m and use the remainder as the hash address.

Each of these methods has disadvantages. For example, the last one may give the same address rather frequently if there are patterns in the keys. Before using a particular method the reader is advised to consult the now extensive literature on the subject, e.g. Morris[15], or Lum *et al.*[16].

As mentioned before there is the problem of collisions, that is, when two distinct keys hash to the same address. The first point to be made about this problem is that it destroys some of the simplicity of hashing. Initially it may have been thought that the key need not be stored with the data at the hash address. Unfortunately this is not so. No matter what method we use to resolve collisions we still need to store the key with the data so that at search time when a key is hashed we can distinguish its data from the data associated with keys which have hashed to the same address.

There are a number of strategies for dealing with collisions. Essentially they fall into two classes, those which use pointers to link together collided keys and those which do not. Let us first look at the ones which do not use pointers. These have a mechanism for searching the store, starting at the address where the collision occurred, for an empty storage location if a record needs to be inserted, or, for a matching key value at retrieval time. The simplest of these advances from the hash address each time moving along a fixed number of locations, say s, until an empty location or the matching key value is found. The collision strategy thus traces out a well defined sequence of locations. This method of dealing with collisions is called the *linear* method. The tendency with this method is to store collided records as closely to the initial hash address as possible. This leads to an undesirable effect called *primary clustering.* In this context all this means is that the records tend to concentrate in groups or bunch-up. It destroys the uniform nature of the hashing function. To be more precise, it is desirable that hash addresses are equally likely, however, the first empty location at the end of a collision sequence increases in likelihood in proportion to the number of records in the collision sequence. To see this one needs only to realise that a key hashed to *any*

location in the sequence will have its record stored at the end of the sequence. Therefore big groups of records tend to grow even bigger: This phenomenon is aggravated by a small step size s when seeking an empty location. Sometimes $s = 1$ is used in which case the collision strategy is known as the *open addressing technique.* Primary clustering is also worse when the hash table (available storage) is relatively full.

Variations in the linear method which avoid primary clustering involve making the step size a variable. One way is to set s equal to $ai + bi^2$ on the ith step. Another is to invoke a random number generator which calculates the step size afresh each time. These last two collision handling methods are called the *quadratic* and *random* method respectively. Although they avoid primary clustering they are nevertheless subject to *secondary* clustering, which is caused by keys hashing to the same address *and* following the same sequence in search of an empty location. Even this can be avoided, see for example Bell and Kaman[17].

The second class of collision handling methods involves extra storage space which is used to chain together collided records. When a collision occurs at a hash address it may be because it is the head of a chain of records which have all hashed to that address, or it may be that a record is stored there which belongs to a chain starting at some other address. In both cases a free location is needed which in the first case is simply linked in and stores the new record, in the second case the intermediate chain element is moved to the free location and the new record is stored at its own hash address thus starting a new chain (a one-element chain so far). A variation on this method is to use a two-level store. At the first level we have a hash table, at the second level we have a *bump* table which contains all the collided records. At a hash address in the hash table we will find either, a record if no collisions have taken place at that address, or, a pointer to a chain of records which collided at that address. This latter chaining method has the advantage that records need never be moved once they have been entered in the bump table. The storage overhead is larger since records are put in the bump table before the hash table is full.

For both classes of collision strategies one needs to be careful about deletions. For the linear, quadratic etc. collision handling strategies we must ensure that when we delete a record at an address we do not make records which collided at that address unreachable. Similarly with the chaining method we must ensure that a deleted record does not leave a gap in the chain, that is, after deletion the chain must be reconnected.

The advantages of hashing are several. Firstly it is simple. Secondly its insertion and search strategies are identical. Insertion is merely a failed search. If K_i is the hashed key, then if a search of the collision sequence fails to turn up a match in K_i, its record is simply inserted at

the end of the sequence at the next free location. Thirdly, the search time is independent of the number of keys to be inserted.

The application of hashing in IR has tended to be in the area of table construction and look-up procedures. An obvious application is when constructing the set of conflation classes during text processing. In Chapter 2, I gave an exmaple of a document representative as simply a list of class names, each name standing for a set of equivalent words. During a retrieval operation, a query will first be converted into a list of class names. To do this each significant word needs to be looked up in a dictionary which gives the name of the class to which it belongs. Clearly there is a case for hashing. We simply apply the hashing function to the word and find the name of the conflation class to which it belongs at the hash address. A similar example is given in great detail by Murray[18].

Finally, let me recommend two very readable discussions on hashing, one is in Page and Wilson[19], the other is in Knuth's third volume[14].

Bibliographic remarks

There is now a vast literature on file structures although there are very few survey articles. Where possible I shall point to some of the more detailed discussions which emphasise an application in IR. Of course the chapter on file organisation in the *Annual Review* is a good source of references as well.

A general article on data structures of a more philosophical nature well worth reading is Mealey[20]. It tries to clarify some of our notions about data thereby constructing a theory of data.

A description of the use of a *sequential* file in an on-line environment may be found in Negus and Hall[21]. The effectiveness and efficiency of an *inverted* file has been extensively compared with a file structure based on clustering by Murray[22]. Ein-Dor[23] has done a comprehensive comparison between an *inverted* file and a *tree structured* file. It is hard to find a discussion of an *index-sequential* file which makes special reference to the needs of document retrieval. Index-sequential organisation is now considered to be basic software which can be used to implement a variety of other file organisations. Nevertheless it is worth studying some of the aspects of its implementation. For this I recommed the paper by McDonell and Montgomery[24] who give a detailed description of an implementation for a mini-computer. *Multi-lists* and *cellular multi-lists* are fairly well covered by Lefkovitz[25]. *Ring structures* have been very popular in CAD and have been written up by Gray[25]. Extensive use was made of a modified *threaded list* by Van Rijsbergen[27] in his cluster-based retrieval experiments. The *doubly chained tree* has been adequately dealt with

by Stanfel[7,8] and Patt[9]. The use of *trees* in IR goes back a long way and the early papers by Salton[28] and Sussenguth[29] are well worth reading. Also interesting is the early paper by Windley[30] which gives some theoretical results about the expected search time. The use of *hashing* in document retrieval is dealt with in Higgins and Smith[31] and Chou[32].

It has become fashionable to refer to document collections which have been clustered as *clustered files.* I have gone to some pains to avoid the use of this terminology because of the conceptual difference that exists between a structure which is inherent in the data and can be discovered by clustering, and an organisation of the data to facilitate its manipulation inside a computer. Unfortunately this distinction becomes somewhat blurred when clustering techniques are used to generate a *physical* organisation of data. For example, the work by Bell *et al.*[33] is of this nature. Furthermore, it has recently become popular to cluster records simply to improve the efficiency of retrieval. Buckhard and Keller[34] base the design of a file structure on maximal complete subgraphs (or cliques). Hatfield and Gerald[35] have designed a paging algorithm for a virtual memory store based on clustering. Simon and Guiho[36] look at methods for preserving 'clusters' in the data when it is mapped onto a physical storage device.

Some of the work that has been largely ignored in this chapter, but which is nevertheless of importance when considering the implementation of a file structure, is concerned directly with the physical organisation of a storage device in terms of block sizes, etc. Unfortunately, general statements about this are rather hard to make because the organisation tends to depend on the hardware characteristics of the device and computer. Representative of work in this area is the paper by Lum *et al.*[37].

Finally, one approach to data organisation which is in spirit like the one based on automatic classification, is that presented by Codd[38] who describes the natural structure of the data in terms of n-ary relations.

References

1. HSIAO, D. and HARARY, F., 'A formal system for information retrieval from files', *Communications of the ACM,* **13,** 67–73 (1970)
2. ROBERTS, D. C., 'File organization techniques', *Advances in Computers,* **12,** 115–174 (1972)
3. BERTZISS, A. T., *Data Structures: Theory and Practice,* Academic Press, London and New York (1971)
4. DODD, G. G., 'Elements of data management systems', *Computing Surveys,* **1,** 117–133 (1969)

5. CLIMENSON, W. D., 'File organization and search techniques', *Annual Review of Information Science and Technology,* **1,** 107–135 (1966)
6. FOSTER, J. M., *List Processing,* Macdonald, London; and American Elsevier Inc., New York (1967)
7. STANFEL, L. E., 'Practical aspect of doubly chained trees for retrieval', *Journal of the ACM,* **19,** 425–436 (1972)
8. STANFEL, L. E., 'Optimal trees for a class of information retrieval problems', *Information Storage and Retrieval,* **9,** 43–59 (1973)
9. PATT, Y. N., 'Minimum search tree structure for data partitioned into pages', *IEEE Transations on Computers,* C-21, 961–967 (1972)
10. BERGE, C., *The Theory of Graphs and its Applications,* Methuen, London (1966)
11. HARARY, F., NORMAN, R. Z. and CARTWRIGHT, D., *Structural Models: An Introduction to the Theory of Directed Graphs,* Wiley, New York (1966)
12. ORE, O., *Graphs and their Uses,* Random House, New York (1963)
13. KNUTH, D. E., *The Art of Computer Programming,* Vol. 1, *Fundamental Algorithms,* Addison-Wesley, Reading, Massachusetts (1968)
14. KNUTH, D. E., *The Art of Computer Programming,* Vol. 3, *Sorting and Searching,* Addison-Wesley, Reading, Massachusetts (1973)
15. MORRIS, R., 'Scatter storage techniques', *Communications of the ACM,* **11,** 35–38 (1968)
16. LUM, V. Y., YUEN, P. S. T. and DODD, M., 'Key-to-address transform techniques: a fundamental performance study on large existing formatted files', *Communications of the ACM,* **14,** 228–239 (1971)
17. BELL, J. R. and KAMAN, C. H., 'The linear quotient hash code', *Communications of the ACM,* **13,** 675-677 (1970)
18. MURRAY, D. M., 'A scatter storage scheme for dictionary lookups'. *In*: Report ISR-16 to the National Science Foundation, Section II, Cornell University, Department of Computer Science (1969)
19. PAGE, E. S. and WILSON, L. B., *Information Representation and Manipulation in a Computer,* Cambridge University Press, Cambridge (1973)
20. MEALEY, G. H., 'Another look at data', *Proceedings AFIP Fall Joint Computer Conference,* 525–534 (1967)
21. NEGUS, A. E. and HALL, J. L., 'Towards an effective on-line reference retrieval system', Library Memo CLM-LM2/71, U.K. Atomic Energy Authority, Research Group (1971)
22. MURRAY, D. M., 'Document retrieval based on clustered files', Ph.D. Thesis, Cornell University Report ISR-20 to National Science Foundation and to the National Library of Medicine (1972)
23. EIN-DOR, P., 'The comparative efficiency of two dictionary structures for document retrieval, *Infor Journal,* **12,** 87–108 (1974)
24. McDONELL, K.J. and MONTGOMERY, A. Y., 'The design of indexed sequential files', *The Australian Computer Journal,* **5,** 115–126 (1973)
25. LEFKOVITZ, D., *File Structures for On-line Systems,* Spartan Books, New York (1969)
26. GRAY, J. C., 'Compound data structure for computer aided design: a survey', *Proceedings ACM National Meeting,* 355–365 (1967)
27. VAN RIJSBERGEN, C. J., 'An algorithm for information structuring and retrieval', *Computer Journal,* **14,** 407–412 (1971)
28. SALTON, G., Manipulation of trees in information retrieval', *Communications of the ACM,* **5,** 103–114 (1962)
29. SUSSENGUTH, E. H., 'Use of tree structures for processing files', *Communications of the ACM,* **6,** 272–279 (1963)

30. WINDLEY, P. F., 'Trees, forests and rearranging', *Computer Journal,* **3,** 84–88 (1960)
31. HIGGINS, L. D. and SMITH, F. J., 'Disc access algorithms', *Computer Journal,* **14,** 249–253 (1971)
32. CHOU, C. K., 'Algorithms for hash coding and document classification', Ph.D. Thesis, University of Illinois (1972)
33. BELL, C. J., ALDRED, B. K. and ROGERS, T. W., 'Adaptability to change in large data base information retrieval systems', Report No. UKSC-0027, UK Scientific Centre, IBM United Kingdom Limited, Neville Road, Peterlee, County Durham, U.K. (1972)
34. BURKHARD, W. A. and KELLER, R. M., 'Some approaches to best-match file searching', *Communications of the ACM,* **16,** 230–236 (1973)
35. HATFIELD, D. J. and GERALD, J., 'Program restructuring for virtual memory', *IBM Systems Journal,* **10,** 168–192 (1971)
36. SIMON, J. C. and GUIHO, G., 'On algorithms preserving neighbourhood to file and retrieve information in a memory', *International Journal Computer Information Sciences,* **1,** 3–15 (CR 23923) (1972)
37. LUM, V. Y., LING, H. and SENKO, M. E., 'Analysis of a complex data management access method by simulation modelling', *Proceedings AFIP Fall Joint Computer Conference,* 211–222 (1970)
38. CODD, E. F., 'A relational model of data for large shared data banks', *Communications of the ACM,* **13,** 377–387 (1970)