EFFICACY OF B-TREES IN AN INFORMATION STORAGE AND RETRIEVAL ENVIRONMENT

By

ARTHUR DOUGLAS CROTZER

Bachelor of Science

Austin Peay State University

Clarksville, Tennessee

1973

Submitted to the Faculty of the Graduate College
of the Oklahoma State University
in partial fulfillment of the requirements
for the Degree of
MASTER OF SCIENCE
July, 1975

EFFICACY OF B-TREES IN AN INFORMATION STORAGE AND RETRIEVAL ENVIRONMENT

Thesis Approved:

R. E. Hedrick
Thesis Adviser

D. W. Grace

D D Fisher

N N Durham
Dean of the Graduate College

PREFACE

This study investigates the efficacy of B-trees in an information storage and retrieval environment. A practical information storage and retrieval system is developed and used to test the performance of B-trees.

I would first like to thank my parents for their understanding and encouragement. Without them, neither my education nor I would have been possible.

A special note of thanks is due to my typist, Mrs. Jane Van Wye. We both successfully struggled through our first thesis. I wish her well on many more and thank her for her diligence and superior work.

I owe a great deal to my thesis adviser, Dr. G. E. Hedrick, and to the other faculty members of the Computing and Information Sciences Department. I thank Dr. D. D. Fisher and Dr. D. W. Grace for their assistance and suggestions and Dr. J. A. Van Doren for his help both in and out of class. It has truly been an enjoyable two years. I wish each of you the very best.

Among many others, it has been my privilege to become acquainted with two true gentlemen. I want to thank you for your friendship and wish each of you, Mr. William S. Davis and Mr. Alan D. Eyler, all of the success and happiness you deserve.

A final word goes to a very special young lady with a mischievous gleam in her eye. I thank you Reta, for the incentive you have given me to go ahead and complete this project.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Certain applications of computer systems require the processing of large amounts of information. This can place a heavy burden on the system to be used. One solution to the data handling problem is to use an extremely large main store. This has never been a feasible solution primarily due to the cost of main storage units. Through advanced technology it may someday be that main storage will be inexpensive enough to satisfy all demands placed on it (3) but the reality of today is that large amounts of data must be placed on secondary storage devices. Furthermore, if ready access is desired for any item of data, only particular secondary storage devices can be used. The use of secondary storage, however, introduces additional time delays which can be significant.

This project has two major objectives. They are:

(1) To investigate the performance of B-trees in a secondary storage environment; and,

(2) To develop an information storage and retrieval system in which the performance of B-trees can be tested.

B-trees are a generalization of other tree-like data structures and should be viewed in relation to the other data structures against which they compete. A data structure is a collection of data items which have some discernible, explicit or implicit, relationship. Both physical and logical considerations are involved when a data

structure is discussed in this paper. Chapter II presents an overview of data structures. This background material develops a framework in which B-trees can be viewed. A data structure is useful only if it is understood. A large portion of understanding something is knowing how it is different. It is hoped that Chapter II shows how B-trees differ from other data structures and why they should be useful in systems using secondary storage.

Chapters III and IV develop further the motion of B-trees with Chapter III emphasizing the definitions, implications, and programming considerations of B-trees and Chapter IV emphasizing the methods by which B-trees are maintained.

Chapter V describes the design considerations involved in implementing the information storage and retrieval system developed as a part of this project. The chapter provides a detailed discussion of the design of the file structures used by the information storage and retrieval system. One of the files is organized as a B-tree and is the basis of the evaluation of B-trees.

Both the system in general and B-trees in particular are reviewed in Chapter VI. The performance of the B-tree used in the system is summarized and recommendations are presented for extensions of the information storage and retrieval system and for further study of B-trees.

Four appendices are included to aid the user in working with the system implemented in the project. Appendix A, a User's Guide, illustrates proper ways to communicate with the several programs in the system. Appendix B contains program logic flowcharts for the programs with Appendices C and D showing sample outputs from the

REPORT program and sample JCL listings for all programs respectively.

CHAPTER II

OVERVIEW OF DATA STRUCTURES

Introduction

Why Use Data Structures?

The simplest and probably most understandable way to store values in a computer memory is to have a unique name for each value to be stored. There would seem, however, to be certain times when the use of single cell data structures is inappropriate. Wilde (23) presents a complete discussion of the appropriateness of single-cell data structures.

What factors cause single cell data structures to be unworkable at times? An obvious response is the fact that certain sets of values have intrinsic relationships which should be exploited. Other sets of values may contain records of variable size, complexity, or number which resist the single cell structure approach. For other applications, programming considerations may dictate that single cell data structures are inappropriate. Even the novice programmer soon realizes that certain programs require special handling.

Evidently there needs to be something that can bridge the gap between what the user of a computer system or program wants to accomplish and the manner in which the computer or programmer can accomplish it. Data structures are one means of helping to bridge

this gap. A data structure is taken here to be the manner in which the data for a program or system is stored. Data structures may contain much or little data and be complex or simple depending on the environment. Data stored in other than main memory generally is called a data set, data base, or data bank. A data set usually represents a physical data structure, whereas a data base or data bank generally represents a more complex logical-physical structure.

In this paper, several types of data structures are considered in varying degrees of detail. This chapter is intended to give an overview of the major classifications of data structures.

The aim is to present a framework in which the features of different data structures can be viewed and compared and to define a major portion of the terminology used later in the paper. Data structure categories do not have strict boundaries with some satisfying several definitions. The data structures are viewed as they might be used in an information storage and retrieval system.

An information storage and retrieval system essentially entails two types of operations. Data storage involves the insertion and deletion of records or parts of records; data retrieval involves the accessing of single or groups of records or partial records. Of course other processes may take place but the above are of major concern in this paper. One additional consideration is the physical location of the data, in either main core storage or on a secondary storage device. This topic is discussed at various times throughout this chapter because the use of large systems involving large amounts of data which need to be retrieved quickly for on-line applications is increasingly important.

Why So Many Different Types?

There seems to be an endless number of completely different types of data structures, arrays, linked chains, queues, AVL trees, etc. All of these different types are useful due to the many and varied applications of computers. A particular type may be appropriate to several applications but others may work more efficiently or be easier to program. A generally applicable rule is that for a particular problem, a highly tailored data structure may be more efficient than a generalized data structure. The variability in uses of data structures is a major catalyst in developing new data structures and hybrid data structures.

Data structures have many common features and are not really as different as the names imply.

Selection Considerations

Since the application does not dictate which data structure to use, the user must use his own judgment in making a choice. Several factors influencing the choice are mentioned below.

Harrison (10) suggests that to a large degree the efficiency attributed to a data structure determines its applicability to a particular problem. This efficiency may be either of the utilization of storage or of the time necessary to complete the tasks required. Constraints on one or the other may cause a trade-off to be necessary.

Again it is the responsibility of the programmer/analyst to define his problem clearly and choose the proper method of solution.

Another consideration might be the ease with which the programming can be accomplished. A program to be used only once may not need a sophisticated scheme. Production programs, on the other hand, would be better candidates for detailed analyses of their data retention requirements.

When considering programs that might be subject to later change, possibly by other programmers, it is well to keep in mind the complexity of the data structures, and how understandable their function will be to another programmer. A strong case might be made for a data structure that is slightly less efficient but much clearer or simpler.

Classes of Data Structures

The overall division of data structures adopted in this chapter is based on the way in which individual data elements of the structure are addressed. The two classes are distinguished as having a computed address or a link address. Rough interpretations would say that an element of the first class is addressed according to a mechanism external to the data elements themselves and an element of the second class is addressed according to information contained within one or more of the data elements. The single cell data structure implicitly defined above comes from the first class. These classifications, slightly altered, came from Harrison (10).

Computed Addresses

Vectors

As a first step in developing data structures more involved than single cells, one could collect several single cells into contiguous storage cells and call this a vector or one-dimensional array. This implies that there exists some single relationship between the elements comprising the vector. This collection of storage locations is given a name representing all the locations in the vector. Individual locations are referenced by a single subscript indicating a positional relationship in the vector. Note that the subscript is not a portion of the contents of a data element as a key generally is but simply implies positional value. Although there is no explicit indication that element N + 1 immediately follows element N in the vector, it is implicit in the definition of a vector. Depending on the language used in implementing a vector data structure, there are different regulations on the size of the vector (number of elements) and upper and lower bounds. The elements of a vector are in order by subscript but the values stored therein may be in any order whatever.

A search for a particular value stored in a vector is most simply programmed by successively comparing each element in the vector with the one to be located until a match is found or the end of the vector is reached. This technique, known as a linear search, is likely sufficient for small size vectors (10 or fewer elements) but is not for larger vectors. The number of operations to be performed in a linear search grows directly with the number of elements in the vector.

Another technique for searching exists, though, in which the number of operations grows as the base two logarithm of the number of elements. This technique, commonly referred to as a binary search, iteratively reduces the domain of search by a factor of two. This homes in on the value to be located rather quickly. An important point is that a binary search on a vector can only be performed if the elements of the vector are in order by value. This places an extra regulation that must be provided for if a binary search is desired.

The term binary search implies a method by which a search can be performed. As is shown later, the same principles guiding a binary search on a vector can be used in searching other data structures. Note that any form of search performed on a vector attempts to determine the position at which a known value is located whereas a subscript reference does just the reverse, attempts to determine the value at a known location.

Vectors generally are static data structures, i.e., subject to few insertions or deletions. This is apparent when one remembers that the elements of a vector must be contiguous in storage and an insertion or deletion necessitates an amount of data movement in order to make room for an inserted element or squeeze out the hole left by a deleted element.

Several specialized vectors which only allow insertions and deletions at the ends are quite useful in many applications. These vectors can be called stacks, queues, and deques although vectors are not the only means by which stacks, queues and deques can be implemented.

Stacks. A vector in which single items are entered and removed from a single end of the vector is termed a stack. No data movement is involved in a stack since the insertion or deletion is always at one end, and only the size of the stack changes. A stack is constructed such that the last item "pushed" onto the stack is the first item "popped" from the stack, such as cafeteria trays. "Push" means to add a single item to the stack and "pop" means to remove a single item from the stack. Since all operations take place on a single end, the stack grows or shrinks only in one direction and is only constrained by the size of the containing vector.

Queues. Queues are similar to stacks in that single values are entered or removed from the queue but are different in that the insert and delete operations are made on opposite ends of the structure. This causes the first item to be entered into a queue to be the first item removed from the queue, such as cafeteria customers in a checkout line. Two addresses, pointers, references, etc., indicating the head and tail of the queue are used in order to maintain the positions for removal and entry. In a stack only the top of the stack varies, but in a queue, both the head and tail may move, such as cafeteria customers, creating a problem if the end of the containing vector is reached but storage is actually available.

Stacks and queues are quite useful in a wide range of applications and are rather easy to use and understand. They can be generalized into a double-ended queue or deque (dequeue according to Stone (20)).

Deques. With a deque, single values can be entered or removed from either end. Thus both stacks and queues are restricted deques. Rarely are the full facilities of a deque required; usually some restricted and hence, easier to program, version is used. Knuth (13) uses the terms "left" and "right" for referring to the ends of a deque.

Multidimensional Arrays

If a set of data items are related in more than one way, a vector may not be sufficient to describe all the relationships involved. In these cases a multidimensional array may be appropriate. One might view a multidimensional array as a collection of several vectors, each vector having all but one of the relationships held constant and the single non-constant one varying throughout its possible values. An array of this type is stored in contiguous locations as is a vector, and has its individual elements identified by subscripts (one subscript for each relationship or dimension). Since the array represents a multidimensional space but is stored in a one-dimensional memory space, there must be an address calculation which determines which element of the array corresponds to a particular subscript reference. Knuth (13) presents a discussion of this calculation and also discusses some additional related topics, particularly triangular arrays. Just like vectors, which are a special case of multidimensional arrays, insertions and deletions are rather ill-advised. The principal use of arrays is when a static table of data is to be referenced not by contents but by positional relations in order to find a value.

Scatter Storage

In the above types of data structures, the value to be referenced is located by calculating its address from a value or set of values denoting its positional property in the structure. In scatter storage techniques, a key, often a part of the record to be located, is manipulated in order to obtain the address of the desired item. This manipulation may involve either a logical or arithmetic transformation of the key or a combination of the two.

Scatter storage techniques generally involve a vector of locations which contains the values to be stored. As each key value is encountered it is transformed by the hashing function into an address of one of the elements in the vector and the data item is stored at that point. The technique thus promises very quick insertion performance. A problem arises if an item is already stored at the calculated hash address. This is called a collision and can be dealt with in many ways.

There seem to be as many collision-handling techniques as there are people with imaginations. A simple scheme is to search linearly through the vector until an opening is found for the new item. This could be a long search if the vector is more than partially filled or the keys are not randomly distributed in the vector. Another scheme is to successively generate new addresses at random until an opening is found. This necessitates the same sequence of addresses to be generated for a search request. Several methods utilizing linked lists including overflow areas are in use. Morris (17) describes

many of these techniques.

One way to reduce collisions is by using a good hashing function or transform. Aside from producing addresses within the specified range of addresses, the function should ideally spread the number of address occurences uniformly over the address space. It is assumed that the possible keys outnumber the possible addresses. This is not the case with vectors or arrays which might be considered as specialized cases of scatter storage. Much work has been done on the development of useful hashing functions. Maurer and Lewis (16) review several common types of hashing functions in addition to other scatter storage related topics.

Even though the problem of collisions prevents the achievement of perfect retrieval performance, scatter storage is still a very quick means of accessing a data item. Insertion is also rather good involving little data movement with most collision-handling techniques. Deletions are a bit more troublesome. With many collision-handling techniques, an item to be deleted cannot simply be removed from the set of items. If it were just removed, a hole would be created which would isolate any following items. If a search routine came upon the hole, it would conclude incorrectly that the item was not found even though it might be in the isolated items. A solution is to tag deleted items with a special code indicating that the position is now available for insertion but should not terminate a search. Morris (17) notes that a deletion handled in this way has no beneficial effect on later searches.

It would thus seem that scatter storage is a reasonable way to go. This may well be the case if the only type of retrieval to be

performed is for a single item. The problem is that no scatter storage technique preserves or creates order in the keys. If a particular application needs to access all the stored entries in key value order, they must first be sorted or some additional mechanism must be attached to the scatter storage technique.

Direct Access to Secondary Storage

For a great many applications the amount of data in a data set may be too large to be contained entirely in main storage. In these cases some secondary storage device must be used as an additional store for data. Programs still may need to retrieve, insert, or delete items from the data set even though it now resides outside of main memory. The notion of scatter storage can be extended to cover these needs. Now instead of developing a main memory address, the hashing transform develops an address of the item on the secondary storage device. This requires the secondary storage device to have direct access capabilities. Thus tape units which must be processed sequentially are not suitable, whereas disk or drum units are.

The hashing or randomizing transform may come in several forms (11). The actual physical device address may be used in certain settings, or a numeric value indicating an offset from the physical beginning of the data set might be appropriate, or a separate cross reference list may be retained which is first consulted and provides the address to be used. In any case, many of the same objectives such as ease and speed of calculation and even distribution of keys stated for scatter storage also apply here. An additional consideration in accessing secondary storage is the time delay resulting

between the issue of a command and the realization of the actual transfer of data. Electronic switching and mechanical movement delays contribute to this additional consideration. Thus secondary storage devices are actually only pseudo-random access devices. Attempts to reduce this time delay may influence the choice of method and placement of overflow records.

Computer manufacturers generally supply at least one direct access retrieval method with their software packages and some have several slightly differing techniques from which the user may choose (4, 11). It is worth emphasizing that direct access techniques, just like scatter storage, do not preserve any logical order to the keys or records and hence are unsuitable if it is important to retrieve more than a single record at a time.

Linked Addresses

The data structures discussed above are sufficient to satisfy the processing needs for a significant percentage of computing applications but are rather rigid. The structures discussed in this section provide the programmer with the option of choosing data structures which are more flexible and hence can be tailored more easily to a particular application. The structures have the common characteristic that the relationships they imply between the data items are not dependent on physical placement of the items but are explicitly stated within the structure itself. This divorcing of the logical relationship from the physical relationship is what contributes to their flexibility. The term "linked list" is used often throughout the literature and is here taken to mean any data structure

which internally contains information describing the data item relationships.

There is a close association between the data structures discussed as linked lists and the notion of a directed graph (2, 8). This association may help to exhibit the similarities and differences of the types of linked lists to be presented.

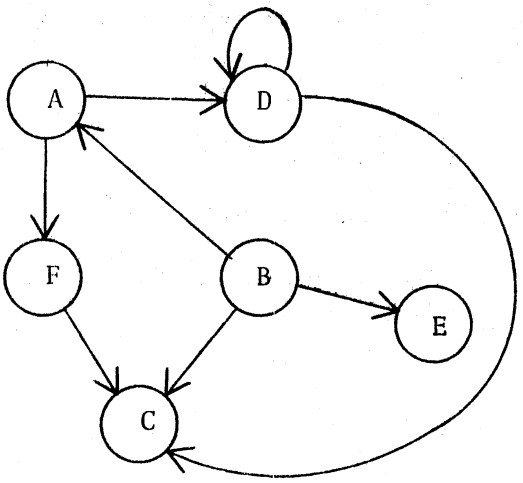
Digraphs

Berztiss (2, p. 103) defines a digraph as follows:

> A directed graph (digraph, oriented graph) is the ordered pair $D=\langle A, R\rangle$, where A is a set of nodes (points, vertices) and R is a relation in A, i.e., R is a set of ordered pairs, which are called arcs (lines, pointers).

Only finite digraphs (A is finite) will be considered in this paper. The relation R can be illustrated with a set of arrows connecting the nodes in a planar representation of a digraph. In Figure 1, the set of nodes is $\{A, B, C, D, E, F\}$ and the relation is illustrated by the directed arrows in the figure. The circles used to represent nodes do not imply the internal composition of the node. In different linked lists a node may itself be a complex digraph. Nodes are points of reference but generally do correspond to the information content of a data structure. Arcs, on the other hand, may be indexes or actual addresses or offsets from a specified address or perhaps some other means of indicating the position at which the corresponding arrow is to terminate.

The second part of Figure 1 displays a connection matrix for for the digraph to its left. Each M (I, J) entry in the connection matrix represents the number of connections (arcs) emanating from



	A	B	C	D	E	F
A	0	0	0	1	0	1
B	1	0	1	0	1	0
C	0	0	0	0	0	0
D	0	0	1	1	0	0
E	0	0	0	0	0	0
F	0	0	1	0	0	0

Figure 1. A Digraph and Corresponding Connection Matrix

node I and terminating at node T. If an arc ⟨X, Y⟩ connects node X to node Y, then X is said to be the initial node and Y is said to be the terminal node. The connection matrix displays several useful pieces of information about the digraph. The sum of the elements in a row tells how many arcs have that node as an initial node and the sum of the elements in a column tells how many arcs have that node as a terminal node. These sums for each node are called the out-degree and indegree of the node, respectively. Thus node A has indegree of one and outdegree of two, and node C has indegree of three and outdegree of zero. A 1 on the diagonal of the matrix indicates that an arc exists from a node to itself (called a loop). The connection matrix and closely related indegree and outdegree

information are referred to often both for consistency and clarity in the development of the data structures.

Some other useful terms relating to digraphs are path, path length, and cycle. A path is said to exist from node X to node Y if a sequence of arcs can be found which connect node X to node Y. The number of arcs in this path is called the path length. Thus there is a path of length one from B to E in Figure 1 and two paths of length two from A to C. A path from a node to itself is termed a cycle. Note that a loop is a cycle of length one. A more detailed discussion of digraphs relating to linked lists can be found in several books including ones by Berztiss (2), Iverson (12), and Knuth (13). The book by Harary, Norman, and Cartwright (9) presents a thorough discussion of directed graphs.

Chains

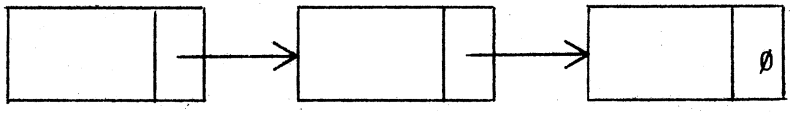
It is not uncommon to find a restricted definition of linked lists to include only chains. Chains can be viewed as vector elements which have been uprooted from their contiguous physical locations and made to reside in locations not necessarily contiguous. The relationship formerly manifested in physical proximity is now represented by links within each data item (node) indicating the location of the next node. These links may take many forms but always indicate the terminal node of the arc represented by the link. This arrangement provides for much easier insertion and deletion of list items. Contrast this with vectors which require much data movement when an item is to be inserted or deleted other than at an end of the vector.

Simple Chains. In the most elementary setting, a node may consist of some information and a single link. The link indicates the next node of the chain. Figure 2 (a.) illustrates a chain containing three nodes. Note that the symbol $\emptyset$ is the link of the last node in the chain. This represents some special signal that no more nodes are to follow.

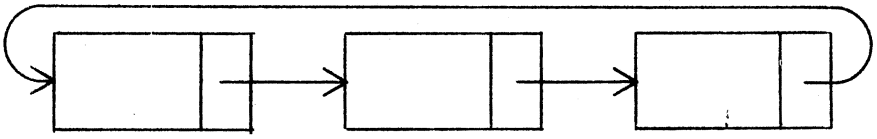
In terms of a connection matrix for a simple chain we can say that the outdegree of every node but one is exactly one and the indegree of every node but one is exactly one. The special nodes are the first and last (head and tail) nodes in the list which have indegree of zero, outdegree of one and indegree of one, outdegree of zero, respectively. The list is thus accessible in one direction, from head to tail passing through each node until the one desired is encountered. This is in contrast with the direct referencing capability vectors.

Stacks, queues, and deques discussed as vectors can be implemented easily as simple chains with each insertion or deletion altering the list length by one and changing either a head or tail pointer. The insertion and deletion operations on simple chains are not as restricted by data movement as those on vectors, however, implying that chains can be more flexible than vectors when insertions and deletions are considered. These operations are illustrated in Figure 3. For an insertion only two links need to be changed, and for a deletion only one link needs to be changed. The algorithms to accomplish insertions and deletions are slightly more complicated when the special cases of an empty list and changing head or tail nodes are

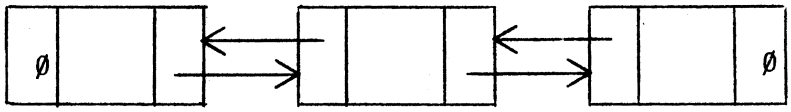
included, but are still quite clear and concise.



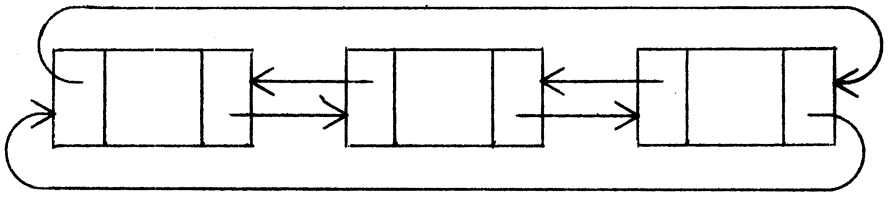
a.) Simple Chain



b.) Circular Chain

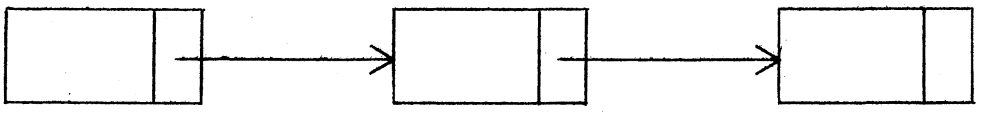


c.) Doubly-Linked Chain

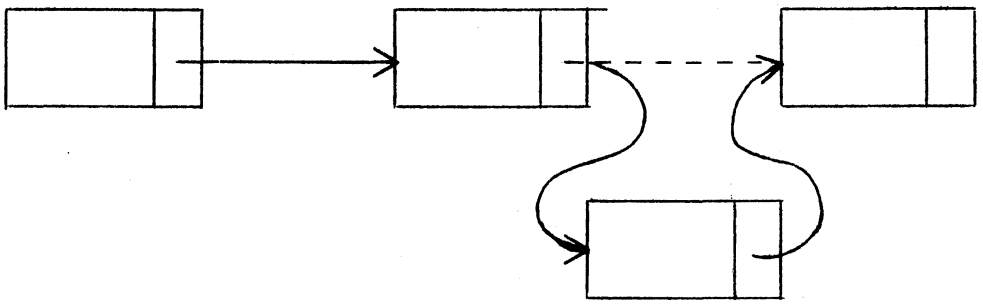


d.) Circular Doubly-Linked Chain

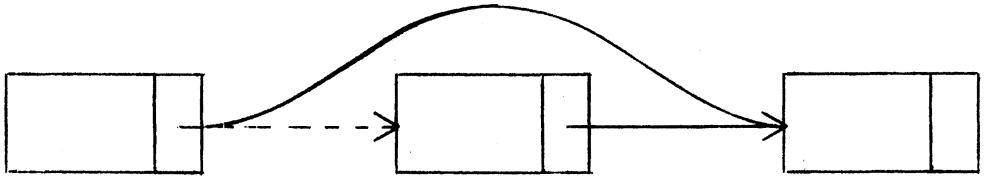
Figure 2. Sample Chains



a.) Original Chain



b.) After an Insertion



c.) After a Deletion

Figure 3. Insertion and Deletion in a Simple Chain

Circular Chains. One difficulty with simple chains is that only those nodes following a given node can be reached. A slight modification is to change the $\emptyset$ link in the tail node to link to the head node. This creates a circular chain, sometimes called a ring. The term ring is ambiguous (Knuth (13)) and is not used in this paper. Figure 2 (b.) illustrates the change needed to create a circular chain.

This modification simplifies the connection matrix relation, causing each node to have indegree and outdegree of one. An interesting characteristic of a circular chain is that the ends of the chain essentially meet and one address can indicate the location of both the head and tail of the chain. This can have beneficial implications in implementing a queue or deque with a circular chain. Additionally, algorithms to insert and delete nodes can be simplified somewhat using circular lists.

Doubly-Linked Chains. For certain applications it is helpful to be able to scan the list of nodes both forward and backward from a specific node. This can be accomplished by installing another set of links in the nodes as illustrated in Figure 2 (c.). Now a node contains information indicating both its successor and predecessor in the list. This is essentially superimposing one simple chain onto a mirror-image simple chain. This view is supported by the connection matrix for a doubly-linked chain which is a symmetric matrix with each node other than the head and tail having indegree and outdegree of two.

The extra linkage is quite beneficial when a deletion is to be performed. With a simple chain the predecessor of the node to be

deleted must be known usually be tracing through the list for the node to be deleted recording the predecessor at each step. With doubly-linked chains, the predecessor is immediately known and no search is required. This can be rather helpful time-wise, if the list is long.

The additional benefits of two-way linking do not come without a cost, however. The extra set of links take storage, either reducing the useful portion of a node or increasing the size of a node. The insertion and deletion algorithms also are complicated to a certain extent due to the additional links which must be set or changed.

Circular Doubly-Linked Chains. Just as circular chains are a logical extension to simple chains, circular doubly-linked chains can be easily constructed from doubly-linked chains. The change required is to set the forward link of the tail node to point to the head node, and the backward link of the head node to point to the tail node. With this arrangement any type of pass through the nodes can be made beginning from any point in the chain. Figure 2 (d.) illustrates the appearance of a circular doubly-linked chain. The connection matrix for a circular doubly-linked chain has each row sum and each column sum equal to two. Note that this is true even if the list only has one element. Both the forward and backward links point to the single node and its indegree and outdegree are two. Figure 4 illustrates this.

Circularly linking a doubly-linked chain can simplify somewhat the insertion and deletion algorithms associated with this type of structure. The requirements of the particular application should be studied before deciding to use double linking and its associated

higher storage requirements.

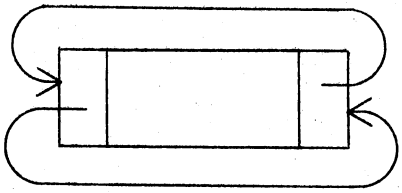


Figure 4. A Circular Doubly-Linked Chain Containing One Node

A subject not yet directly addressed is the manner in which the chain itself is referenced. Two approaches seem to be popular. In one, separate locations are kept which point to the head or tail or some intermediate node in the chain. In the second approach, these separate pointers are incorporated in the chain itself but always kept available. This second approach has the added benefit that the list is never completely empty but always has at least the fixed pointer.

In review, chains are flexible structures, with the ability to accommodate insertions and deletions efficiently. They also are easy to program and understand. The major disadvantage is the inability to reference directly elements contained in the list. To find the nth node, the links must be traced until the nth node is found. This tracing of links may be a strong deterrent when large lists are considered. It may not be important, however, if the nodes need to be referenced sequentially.

Storage Management. It has been assumed in the discussion of chains that when a node is needed for insertion it will be available or when one is deleted it will be reclaimed properly. These features are not automatic but must be provided for by the programmer or by the programming language if it has the power to do so. Several storage management schemes exist for the selective disbursement and reclanation of storage locations. Knuth (13) and Sherman (19) present discussions on this subject.

The subject of chains is treated to some extent by a great many authors. Notable among these are Knuth (13), Harrison (10), and Stone (20).

Trees

A large number of natural physical relationships require structural representations other than chains. For example, a chart illustrating the managerial levels in a large corporation cannot be depicted easily or clearly using one of the chain structures discussed above. The hierarchical nature of the relationships contains connections which cannot be accommodated with even a doubly-linked chain. This type of data arrangement necessitates a new data structure which is called a tree and is specifically designed to accommodate hierarchical data relationships.

General Trees. An example of a tree is illustrated in Figure 5, along with indications of the meanings of certain special terms associated with trees. In terms of the digraph analogy used in des-

cribing other linked lists, the indegree for each node in a tree is exactly one except for one special node called the root which has indegree of zero. Further, all nodes having outdegree of zero are

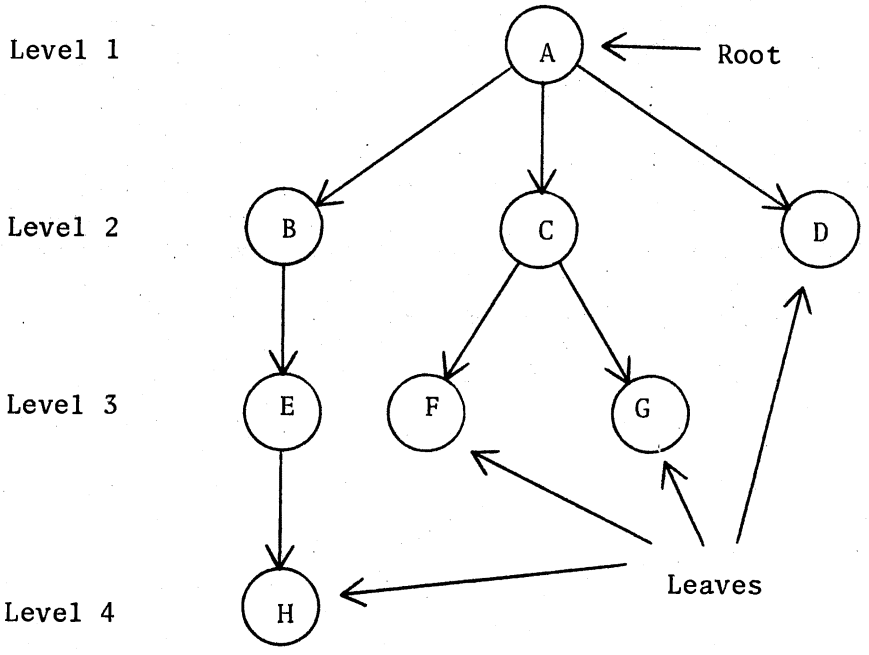


Figure 5. A Tree

called leaves (terminal nodes or twigs by other authors). The outdegree for all non-leaf nodes in a general tree can be any positive integer. The restriction on the indegree of all non-root nodes results in there being a single path from the root to any node. The nodes are divided into levels (1-origin numbering) according to their path length from the root node. Finally, an important point is that in a general tree, no order between the branches from a node is to be

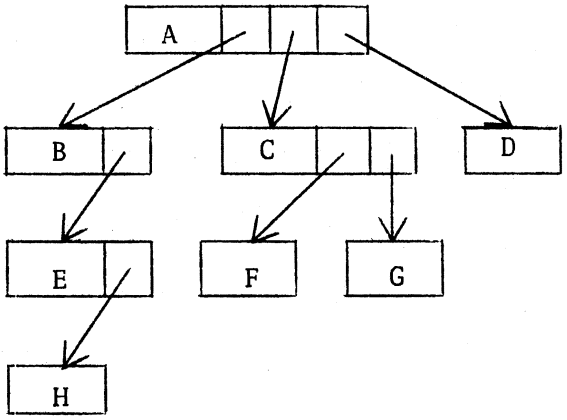
inferred. This last point will be expanded in a later discussion. In Figure 5., node C is said to be a successor of node A; node C is said to be the predecessor of nodes F and G; nodes B and D are said to be siblings of node C; and node B is said to be the root of a subtree consisting of nodes B, E, and H.

There are several methods which can be used to represent general trees within a computer. Three of these are illustrated in Figure 6. The tree represented is that in Figure 5.

The first type is an explicit linking representation. Each node contains the linking information necessary to point to its successors. Only as many links are needed as there are successors. This representation is a logical extension to chains with the one-way linking of a simple chain being expanded to include possibly several successors. The backward linking of doubly-linked chains can be approached in trees using threads (Knuth (13)), but that subject is not discussed here.

A vector representation of a tree is designed essentially for static trees. A node vector in a particular order (Iverson (12)) and an associated degree vector are entirely sufficient to describe a tree. This type of representation is not as easy to update as the explicitly linked, but is more conservative of storage since the links are not required.

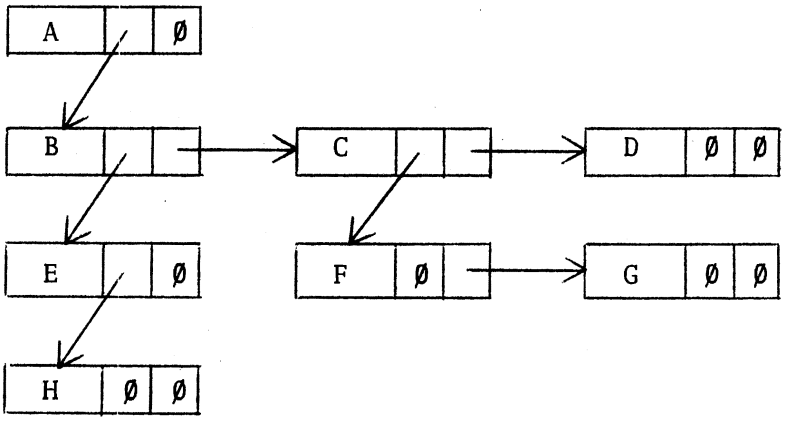
The third type of representation uses a bifurcating node: it is one which contains exactly two links. Through an ingenious mapping, any general tree can be represented by a tree constructed from bifurcating nodes. The links in the nodes of such a tree are designated as left and right. This is a more important premise than might seem obvious. An order is placed on the branches from a node. This pre-



a.) Explicit Linking Representation

Node Vector	Degree Vector
A	3
B	1
E	1
H	0
C	2
F	0
G	0
D	0

b.) Vector Representation



c.) Binary Tree Representation

Figure 6. Representations of General Trees

vents such a tree structure from being represented as a digraph and hence is not a tree in the strict mathematical sense. Rather, it is an ordered tree, and subtrees are referred to as left and right subtrees. The mapping then takes the form of using the left link of a binary tree node (taken from the two link appearance) to point to a successor of the node in question and the right link to point to a sibling. The hierarchical relationship obvious in the original tree. is not quite so obvious in the ordered tree but can be recovered by using the knowledge of how the binary tree representation of the tree was constructed. Note that there are several links which are marked as null (Ø) in the binary tree representation (Figure 6 c.)). A point worth considering is that the right link of the root is always null when representing a single tree. This can be utilized where several trees--a forest of trees--is to be represented by linking this root node to the root node of the next tree in the forest.

The necessity of precisely defining the structure of a tree when represented in a computer makes ordered trees of great importance since they have a very predictable form.

Binary Trees. Although binary trees can be used to represent general trees, their usefulness is not so restricted. For instance, Figure 7 illustrates a binary tree representing the arithmetic expression A * (B + C). This representation obviates the need for parentheses to denote priority of operators since the hierarchial nature of the tree does that automatically.

Another use for binary trees is to represent a set of data such that the operations of searching, insertions, and deletions can be

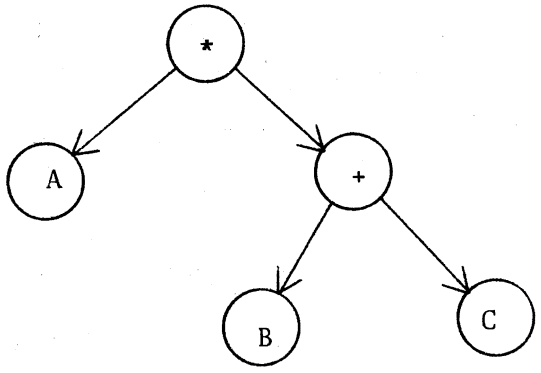


Figure 7. Binary Tree Representing an Arithmetic Expression

executed most efficiently. A binary search on a vector of sorted items has excellent search characteristics but rather poor update performance and linked lists (chains) have the opposite properties. Binary trees can serve to combine the two techniques and produce an all-around efficient data structure which achieves the best properties of each.

Figure 8 illustrates a small binary search tree, as it will be called here. The values in the circles indicate the values of the keys for which a search will be made. The node may contain additional information but only the key is shown. Note that the keys in the left subtree for any node are lexicographically smaller than the key in the node and that the keys in the right subtree for the node are all lexicographically larger than the key in the node. Thus if one is searching for a key value of X, and is at a node with key value Y, the

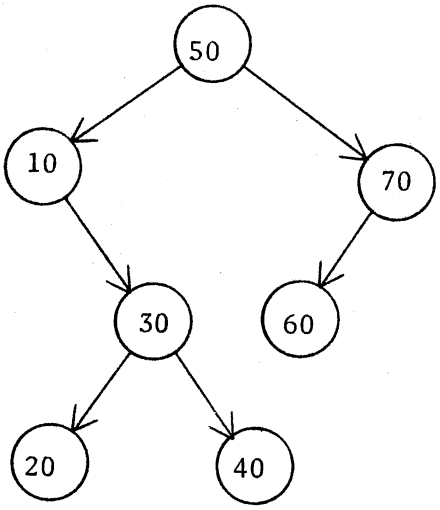


Figure 8. A Binary Search Tree

left subtree should be searched if X < Y and the right subtree should be searched if X > Y. If X = Y, the search is complete. If the tree is perfectly balanced, i.e., the path length from the root to the farthest leaf node is not more than one greater than the path length to the nearest leaf node, then the tree should have search characteristics near those of a binary search (on the order of $\log_2 N$, where the tree has N nodes). The tree also has the advantage of requiring no data movement when an insertion is to occur--only the changing of a few pointers.

A binary search tree can have undesirable search characteristics, however, if the tree is not built randomly. For instance, if the tree is constructed with keys being input already in order, then the tree degenerates to a linked list with order N probes needed to search the

list. This may or may not be a possibility in a certain application but should be realized. Nievergelt (18) presents a comparison of binary search trees with some other common data structures and Knuth (14), presents detailed analyses and observations on the performance of binary search trees.

Traversals. For a particular application it may be necessary to retrieve a part or all of the information contained in a binary search tree. Since the nodes are not necessarily stored in contiguous locations, a sequential pass through memory does not suffice. Some means of systematically tracing the left and right links is needed to effectively recover the information.

A binary tree essentially can be divided into three parts--a left subtree, a root, and a right subtree. Notice that this division applies not just to entire trees but to subtrees as well. Three general approaches to the traversal question are most often applied. They are described and named below: (these definitions correspond to those given in Knuth Volume 1. Second Edition)

Inorder Traversal
    Traverse the left subtree in inorder
    Visit the root
    Traverse the right subtree in inorder

Preorder Traversal
    Visit the root
    Traverse the left subtree in preorder
    Traverse the right subtree in preorder

Postorder Traversal
    Traverse the left subtree in postorder
    Traverse the right subtree in postorder
    Visit the root

Here visit means to accomplish whatever is to be done with the information in a node once it is accessed.

The traversal schemes indicate where to go next when a node is encountered. The definitions apply recursively throughout a tree and hence require a stack or recursion in order to be able to back out of the tree from some internal node. The traversal names used here came from Knuth (13) who changed terminology in the second edition. The reader is warned to check his terminology before reading and comparing traversal schemes in order to avoid confusion. Wilde (23) also presents a readable discussion of traversals, as does Stone (20).

AVL Trees. AVL trees amount to binary search trees which are restricted from becoming out of balance. This guarantees an efficiency of searching which cannot be guaranteed with binary search trees.

Balance tags in each node of an AVL tree are maintained to indicate the degree to which the subtrees of each node are out of balance. If an insertion or deletion causes a node's subtrees to become out of balance by more than one level, the balance tags detect this and signal that corrective action is needed. This corrective action involves "rotating" the tree locally to produce a tree that is in balance. Although these rotations may need to be propagated upward through the tree, Van Doren and Gray (22) have shown empirically that the average number of transformations necessary after an insertion or deletion are approximately 0.5 and 0.23, respectively. The guaranteed upper bound on the number of probes into an AVL tree has been established to be about $1.5 \log_2 N$ (14, 22) but empirical evidence indicates it is lower (22).

According to the algorithms discussed in the paper by Van Doren and Gray (22), if the key 45 were inserted into the tree in Figure 8, the node 10 would be out of balance by more than one level and would require restructuring. After restructuring, the tree would appear as in Figure 9.

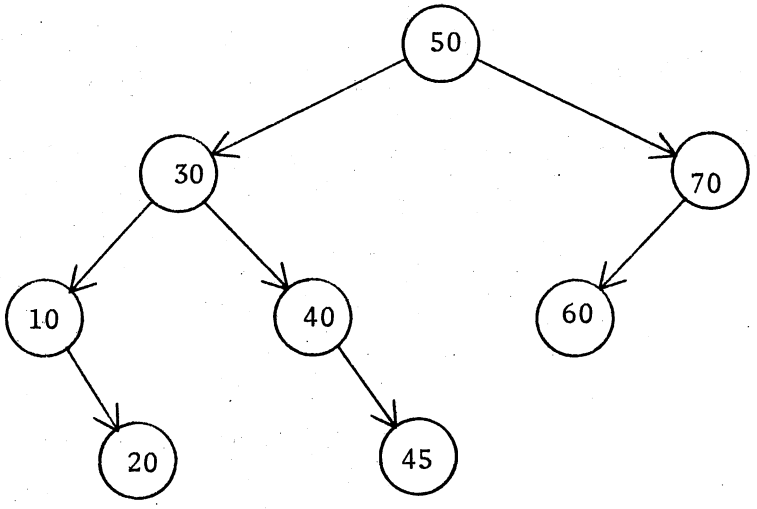


Figure 9. An AVL Tree

AVL trees then, both in theory and by empirical evidence, seem to be useful data structures for searching and for updating. It seems, however, that AVL trees are most useful when used in internal memory. A tree with only 10 levels when stored on secondary storage might require 10 separate secondary storage accesses. Due to the extreme difference in speeds of internal processing and secondary storage retrieval, this would very likely be prohibitive.

If a set of data is to reside on secondary storage but needs to be accessed randomly, the direct accessing method discussed along with hashing would be a likely candidate, but when it is desirable to retain logical ordering, another approach is needed. If the restriction on the number of branches from a node is relaxed, the number of keys and the size of a node can increase and the number of levels in the tree can decrease. The fewer levels there are, the fewer secondary storage accesses are required. This is the rationale behind the next two data structures.

Indexed Sequential. Chapin (4) cites six computer manufacturers as having available indexed sequential software for accessing sets of data stored on secondary storage. An indexed sequential file (a logical file is generally used to refer to a physical data set on secondary storage) usually is a tree structure having three levels. The first two levels are indexes subdividing the file into smaller pieces for searching. The third level contains the actual information to be referenced. In IBM terminology (11), the first level is termed the cylinder index and contains the highest key on each cylinder containing a part of the file and a pointer to that cylinder. Each cylinder, in turn, has a track index which indicates the highest key on each track of that cylinder. Finally the track is searched to retrieve the desired record. Sometimes there are more or fewer than two levels of indexing but two is generally the case, although one is required. The number of cylinders and number of tracks per cylinder are set prior to creating the file so there is an upper limit on the size of each "node". This arrangement makes it possible

to find a record in a number of probes equal to the number of levels of indexing.

The major problem with indexed sequential files concerns the handling of insertions. A static file is most efficient both for direct and sequential processing. Consider the schematic layout of a disk pack containing an indexed sequential file in Figure 10. The prime data area contains the original file. If an attempt is made to insert a record into a full track, the later records on the track are moved up in order to make room for the new record and the one bumped off the track is placed in an overflow area for that cylinder.

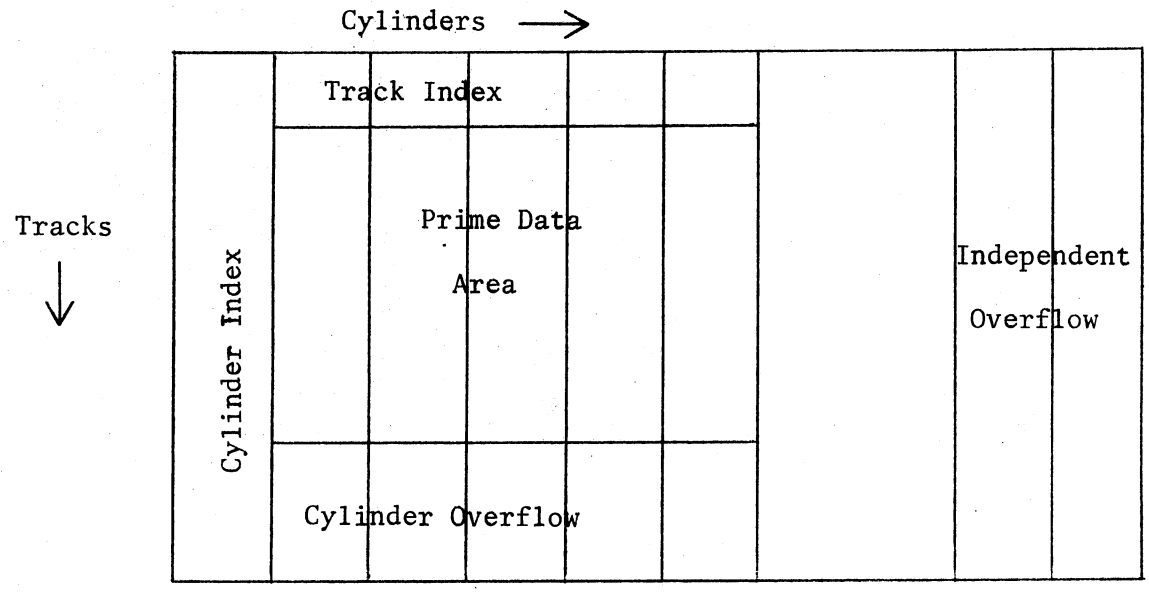


Figure 10. A Schematic Representation of an Indexed Sequential File

Another insertion into that cylinder bumps another record into the overflow area and, in order to retain logical order, the records are linked together. If a cylinder overflow area becomes full, an independent overflow area can be used with all the overflow records being linked together. Then when a search is called for, it may be necessary to trace through a chain of pointers to find the record. This can degrade the performance of a program using an indexed sequential file greatly. This method of overflow insertion can be likened to unbalancing a binary search tree.

Deletions from an indexed sequential file are only marked and the holes created are only filled when the file is restructured. A restructuring involves rebuilding the file, usually a time-consuming process. It is unfortunate but true that the most active files are usually the largest and hence take the most time to restructure. In addition to the references cited above, Flores (8) presents an entire chapter on the subject of indexed sequential access.

B-Trees. The multiway branching structure known as a B-tree (1) can be used to retain sets of data on secondary storage with a guaranteed efficiency on storage utilization and a guaranteed fewer number of accesses than the AVL trees discussed previously. The structure also is quite efficient in handling insertions and deletions with no degradation of search characteristics as with indexed sequential files.

Each B-tree node consists of a set of keys and links, with there being one more link than key. The number of links in any node is guaranteed to be more than about half a predetermined maximum for

the node. This maximum is called the order of the B-tree.

B-trees were first discussed by Bayer and McCreight (1) and are treated in later chapters of this paper.

Multilinked Lists

Lists. Knuth (13) discusses a linked structure he calls a List (the capital is used to distinguish it from the more general term list). The significance of a List is that it allows the full range of connections possible with a digraph. In other words, overlapping lists and even recursive lists are possible. Overlapping lists (trees with common subtrees) can be usefully applied to some applications to reduce the redundancy of data necessary if separate trees with separate subtrees are constructed. Recursive lists, lists involving paths closing on themselves, should be used with great care or endless operations might occur. For instance, if a search operation expecting to find a null link enters a cycle, it will not terminate. Likewise, a copy operation may run into the same problem.

Trees are a restricted case of Lists and can always be represented as Lists. The reverse, however, is not true; not all Lists can be represented as trees. Trees are a proper subset of Lists. Knuth (13) discusses the programming implications of Lists and several approaches to their implementation.

Large and complex data sets are often located on secondary storage, making the time for each secondary storage access important. Although there may be an index to the data set, it is not necessarily a hierarchical one as in indexed sequential files.

Multilist. A multilist file (6, 15) consists of a set of records which are interconnected with several one-way chains. A multilist file with only one chain of links active amounts to a simple chain discussed earlier. As an example, consider records containing an employee name, a job title, and his number of deductions. If it is important to retrieve all records of a specific job title or of a specific number of deductions, links corresponding to these fields can be placed into the records and an index containing the possible common field types can be established. Figure 11 illustrates this type of data structure. In order to find all lawyers, first the title index is searched for lawyer and then the links are followed to locate record numbers 50 and 20 until a Ø link in the title link field is encountered. A similar approach would be used to serve a request for a number of deductions. To find all lawyers with 0 deductions, both chains have to be followed looking for common records. Since the records are probably stored on disk, each record retrieval necessitates a disk access. This is the major disadvantage with multilists. A large file may require many disk accesses--too many to be useful.

The approach does have some desirable features, however. New records can be inserted into or deleted from the file with relative ease due to the linked nature of the relationships. The simplicity of the approach is also a factor in its favor. If the number of disk accesses could be reduced, the data structure would be much more attractive.

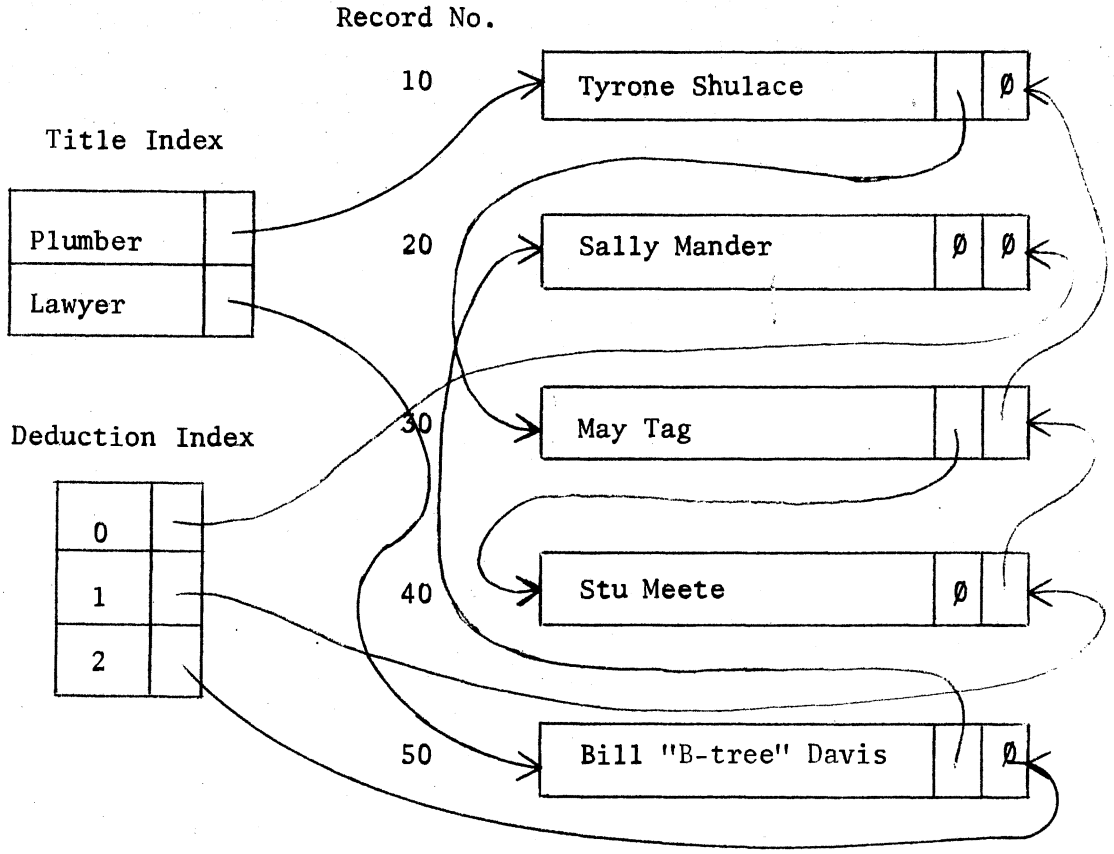


Figure 11. A Multilist File

Inverted List. Using an inverted list (6, 15) approach can help to solve the searching problem encountered with multilists. In an inverted list, the links contained in the records are removed from the records and placed into the indexes. Thus the indexes for the problem in Figure 11. taken as an inverted list would appear as in Figure 12. It is readily apparent from this information that there is

Title Index

Plumber	10 30 40
Lawyer	50 20

Deduction Index

0	20
1	40 30 10
2	50

Figure 12. Inverted List Indexes

a single lawyer with zero deductions. This can be determined solely from the indexes and the only time the record file need be accessed is to retrieve the requested records. In fact, if the entries in the index are kept in a consistent order, only one pass of the indexes need be made.

Although the disk retrieval characteristics have been much improved, this approach leads to other difficulties. The indexes are much more difficult to update. The index entries must be variable in length to contain the variable number of inverted list record references. If the logic to maintain variable blocks is not used but, instead a large block is allocated for each index term, much space will be wasted. A second consideration is that, if the references are to be kept in collating order, they must be inserted in the proper place, magnifying the updating difficulty.

A file is said to be partially inverted if only a portion of the fields in each record are inverted whereas the file is said to be totally inverted if all fields are inverted. The idea of inverted or multilist files can be applied to other uses than simply connecting like attributes. For instance a separate chain or inverted index entry could be used to retain alphabetical order for the names.

Controlled List Length Multilist. A connection between the multilist and inverted list approaches can be achieved through a multilist with controlled list length (15). This essentially means that some upper limit is placed on the number of records that can be contained in each multilist chain. A limit of five would say that a file containing 15 lawyers would have three chains corresponding to lawyer and hence three index entries.

An advantage to this approach is that, under certain circumstances, the accessing of records can be overlapped. In other words, when a record from one chain is coming in, a record from another chain may also be coming in. Unfortunately, the ability to process intersection and union requests in the index itself as is done with

the inverted list, cannot be done with this approach. The approach may, however, be faster than a straight multilist and the ability to vary the upper limit on the list length from index generation to index generation may be an advantage.

The multilist with controlled list length bridges multilists and inverted lists. Note that an upper limit on the list length of one produces an inverted list and no upper limit produces a straight multilist.

Cellular Multilist. Since a secondary storage unit is modular by nature, it is reasonable to take advantage of this modularity. A cellular multilist (15) is based on this premise. The controlling factor on the length of a chain is taken to be the number of records in a given cell of the secondary storage device. Each list contains records wholly contained in the same cell. This promotes the overlapping of accesses only achieved by coincidence with the multilist with controlled list length. A programmer or system designer should be very familiar with not only his programming requirements but the features of the equipment he is to use if he wants to speed up his execution.

CHAPTER III

CHARACTERISTICS OF B-TREES

In 1972 Bayer and McCreight (1) reported development of a new data structure termed a B-tree. The new data structure was designed to serve the user in organizing and maintaining an index to a large dynamically changing random access file. An index is some means of retaining keys to the records in a file. The indexes considered were ones so large that they could not be kept in main storage but had to reside on secondary storage (typically a movable head disk unit). The data structure used reduced effects of disk storage time delays by reducing the number of disk accesses required. Additionally B-trees were found to guarantee a reasonable percentage of storage utilization and to be acceptably easy to update.

A B-tree is a tree structure in which each node can have multiple branches. The maximum number of branches possible from each node is termed the order of the node. It may be possible that the order varies from level to level, but usually is the same for each node in the tree. If the order is the same for each node, it is called the order of the tree, otherwise the order of the tree is not constant and not specified. Thus a B-tree of order 11 has a maximum of 11 branches per node.

A data structure is a B-tree if and only if it satisfies the following conditions:

1). Every node has at most M successors (M is the order of the B-tree).

2). Every node, except the root and the leaves has at least $\lceil M/2 \rceil$ successors ($\lceil x \rceil$ indicates the smallest integer that is greater than or equal to X).

3). The root node has at least two successors unless it is a leaf, in which case it is the only node in the tree.

4). All leaves have null pointers and are on the same level, which in fact is the bottom level of the tree.

5). A non-leaf node with K successors has K - 1 keys. The above conditions will be referred to as properties of a B-tree.

The above properties imply several things about B-trees. Properties 1, 2, and 5 together imply that every node of the tree contains between $M/2 - 1$ and M - 1 keys. This says that each node is at least half full or there is at least 50% storage utilization. It is shown later that the storage utilization is actually much higher than 50%. Property 4 indicates that all leaves are on the same level, the bottom level of the tree. This forces the tree to be in constant balance, guaranteeing searching efficiency. A side implication is that since all leaves appear at the bottom level, the tree must grow upward and not downward as all other tree structures have. This is illustrated in the next chapter.

Figure 13. illustrates a B-tree of order 4. The root node for the B-tree contains the key 50 and there are three levels in the tree. The convention is used here that any link field which is blank should be taken to be null. It is of critical importance when sequential lookup is desired that the keys within a node be in their

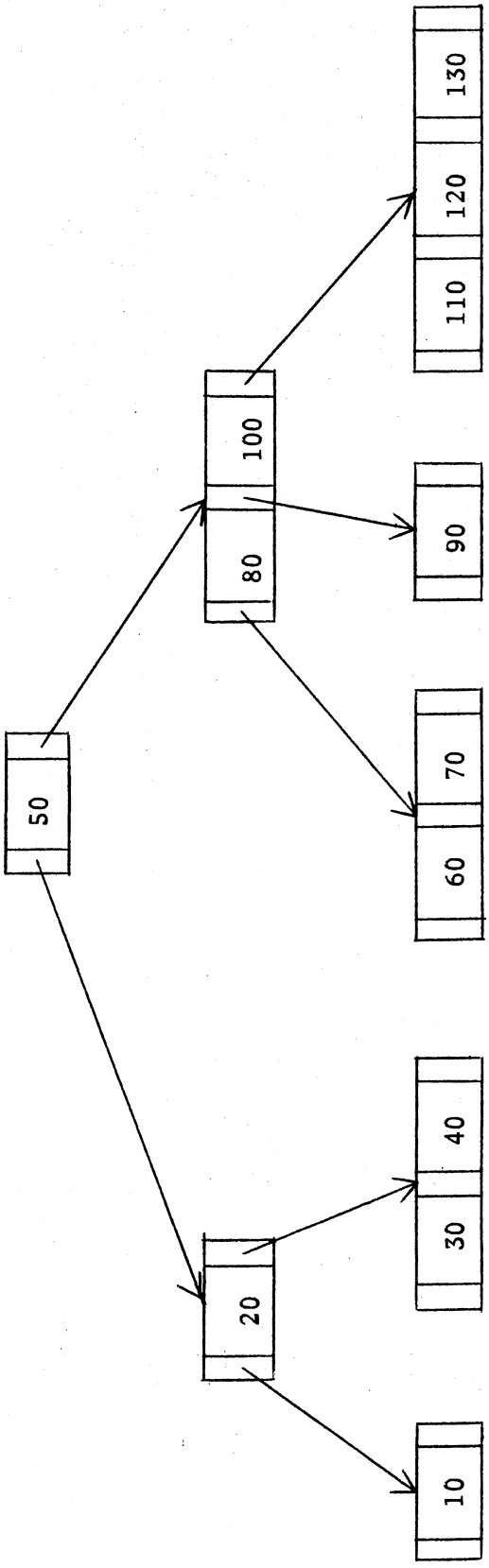


Figure 13. A B-tree of Order 4

proper collating sequence. As a referencing convenience, the links within a node are numbered sequentially beginning with zero and the keys are numbered sequentially beginning at one. Thus key 100 is the second key in that node and the node containing key 90 is pointed to by the link numbered one in its predecessor. Analogous meanings of successor, predecessor, left and right sibling, and subtree used for binary trees are used when referring to B-trees.

The number of levels in a particular B-tree is important because it affects the number of disk accesses necessary to retrieve a record. Bayer and McCreight (1) and Knuth (14) have shown that an upper bound on the number of levels, l, in a B-tree of order M containing N keys is given by:

$$l \leq 1+\log_{\lceil M/2 \rceil}((N+1)/2) \qquad (1)$$

This states that the number of levels is not just a function of N as with other trees, but is also strongly affected by the order of the tree. The order, however, determines how large each node is. Thus there is a trade-off between the size of each node and the number of levels in the tree. For this discussion, it is assumed that the order is held constant throughout the tree. For a large order, the number of levels is small, indicating good access qualities but the search within a node is increased. Since the keys in a node are in sequence, a binary search or even some tree structure could be used to lessen the search time but it still should be considered. For a small order, the node size is small causing a short in-node search but the small order generates many levels causing many disk accesses. It thus seems that the choice of M can be an important one on the

performance of the B-tree. This choice affects: (1) the node occupancy ratio of the number of keys in a node to the maximum number of keys possible per node, (2) the reorganization required within a node, and (3) the reorganization required among nodes (7).

If the tree is stored on disk or drum, a likely choice for a node size is the size of a track. This is due to the nodularity with which the information on a track can be transmitted. Evidence indicates that when secondary storage timing considerations are analyzed, there is a broad minimum ov values of M which will give nearly optional performance (1, 14, 21). Another constraint on the node size arises if the data structure is to be used in a virtual storage environment. In such a case, a single virtual page might be a good choice for a node size. This would prevent excessive paging to occur as might happen if the node were several pages in size and a binary search within the page were implemented. A side note to the virtual storage question, is that one version of the Virtual Storage Access Method (VSAM) by IBM contains a good many of the ideas and terms associated with B-trees (11).

All is not perfect with B-trees, unfortunately. There is a trade-off between the size of each node and the amount of processing it implies. A tree with small order (approaching an AVL tree) has many levels and more frequent maintainence transformations; however, the transformations are relatively simple. Trees with large orders (as exemplified in this paper) require few levels and less frequent maintainence transformations but each transformation is more complex and more time consuming. In a large node, the retention of order within the node is a major factor. An insertion may cause much data

movement and hence be quite expensive.

The problem of data movement in large nodes is a result of two timing considerations; the time needed to move data within main storage and the time needed to transfer information to and from secondary storage.

Paging and gather writing are two approaches to reducing this data movement problem. Paging, as used by Bayer and McCreight (1), involves retaining a number of pages (nodes) of the B-tree within main memory and attempting to do as much processing as possible within main memory. This reduces the number of actual I/O operations to secondary storage since many of the transfers can be entirely within main memory. With this scheme the only time a node is actually read is if it is called for but is not presently in one of the pages in main memory and the only time an actual write is required is when a page is to be actually read and no page area is available for it. In such a case the least recently released (written) node is actually written to secondary storage. Using 10 internal pages and an order of 121, McCreight found that the number of actual reads and writes required when randomly building a B-tree with 5000 keys was only 50% of the total number of reads and writes called for.

A closer inspection of paging reveals that its effectiveness decreases as the number of active nodes becomes larger. For a tree with order greater than 300, if three pages are kept in main storage until the fourth node is activated in the tree, no actual writes or reads (less the three to fill the pages) are required. This means that over 900 keys could be inserted with three actual reads and no actual writes.

Assuming two levels and permanent retention of the root node, if there are 10 leaf nodes, then there are two chances in 10 that the node to be read in a search is already in main storage but if the tree has 100 leaf nodes, then there are only two chances out of 100 that the requested node is already present. This means that percentage-wise more actual data transfers are required as the tree becomes larger. This scheme has a great benefit, however, if the keys to be inserted are already in order since the proper nodes are in main storage more often (1).

Gather writing involves the collecting of data taken from several noncontiguous locations in main storage during a writing operation. This eliminates the need for the data to be moved within memory to a buffer before transfer to the secondary storage device. By undertaking gather writing (through the use of channel programming in IBM terminology), the time and data movements required for each write to secondary storage are reduced.

One wonders whether the benefits of gather writing and paging could be collected in a single implementation. There seems to be a drawback to this, however. When a node is to be written in a paging scheme, it simply replaces the current copy of the node in one of the internal pages. Only when an actual read is needed and no page is available does an actual write take place. In such a case the internal node least recently written is transferred intact to secondary storage. The data to be written comes from only one source and hence does not require or benefit from gather writing. It seems that to a large extent, paging and gather writing are mutually exclusive. Paging attacks the problem by attempting to reduce the number of

actual writes and reads whereas gather writing attempts to reduce the requirements placed on each write operation.

A final topic for this chapter concerns the different ways in which information can be stored in a B-tree. There are three basic classes of B-trees: those that contain information only in the leaf nodes, those that contain information directly in all levels of the tree, and those that contain only pointers to the records which are stored in another file.

The first of these classes is similar to the indexed sequential organization discussed earlier. However, B-trees possess much better insertion and deletion characteristics since they do not degrade to a linear search as can happen with indexed sequential files. Since there is duplication of some keys, the tree may have more levels. An inorder traversal of the tree is probably faster since fewer nodes containing information are retrieved, however, the approach causes slightly more complex programming problems since the order of the tree would vary at the bottom level.

The second class is probably more straightforward but not necessarily more efficient than the first. Since the information is contained in each node, the order for each node is reduced, increasing the number of levels in the tree.

The third class removes the information from each node, placing it in a separate file and planting a link to it in the B-tree node. This allows the order to increase and the number of levels to decrease. Note that separating the information from the keys as in this class is most beneficial if there is a likelihood of multiple keys per piece of information. The other two methods allow redundancy of

data. This approach is somewhat like causing the B-tree to be the index to a multilist file.

In any of these classes, there are a great many links in the leaf nodes which are of no value. For certain applications, it might be advantageous to cause the leaf nodes to have a different structure and remove the unnecessary links and pack more information into the leaf nodes.

CHAPTER IV

OPERATIONS ON B-TREES

This chapter is intended to provide the reader with an introduction to the searching, inserting, and deleting functions as they apply to B-trees. Not all possible variations are covered. For a further discussion, refer to the in depth report by Davis (5).

Searching

As with any tree structure, a search for a random key should begin at the root node if no special information is a priori known about the tree. For this reason, it would be advisable, if possible, always to retain the root node in main storage to reduce disk accesses. A search in a binary tree involves tracing through perhaps several links until the key is found or a null link is encountered. A search in a B-tree is a bit more complicated. Since each node may contain several keys, an additional search within the node is required. A node itself may be structured as a vector, allowing a linear or binary search or as a tree providing for a tree search. In any case, there is a link on either side of each key and the subtree pointed to by this link contains keys less than or greater than the key depending on whether the link is to the left or right of the key.

A search proceeds from level to next lower level attempting to locate the search key in each node. If the key is in a node, the

search terminates; otherwise the link between the two keys less than and greater than the search key is followed to the next lower level. Thus two pieces of information about each node encountered in the search are of prime importance--the identification of the node and the position in the node at which the key is or should be located.

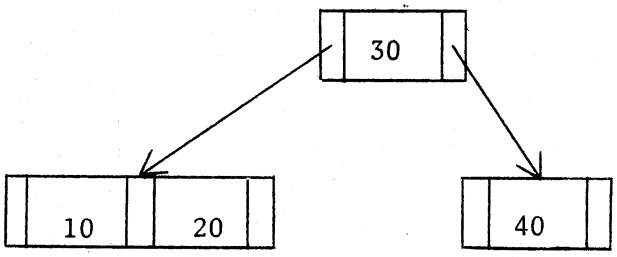
Consider the order four B-tree in Figure 13. If key 80 is to be found, the root node is examined and the search key is found to be greater than the largest key in the node so the rightmost link is followed. In the next node to be searched, the search key is found at key position 1 so the search terminates. Now consider a search for key 85. The link designated as being in link position 1 of the root is followed since 85 is greater than 50. The search key of 85 falls between 80 and 100 in the node at level two so the second link is followed to the next level. The smallest key in this node is 90 so link zero should be followed to the next level, except that this link is null; thus the search terminates without finding the key but with an indication that the key should be in position 1 of the particular leaf node.

Two observations are appropriate at this point. In order to determine that a key is not in the tree, it is necessary to search through the entire height of the tree. Also, when a new key is to be placed into the tree, it will be placed into a leaf node, but when a key is to be deleted it may come from some node other than a leaf node.

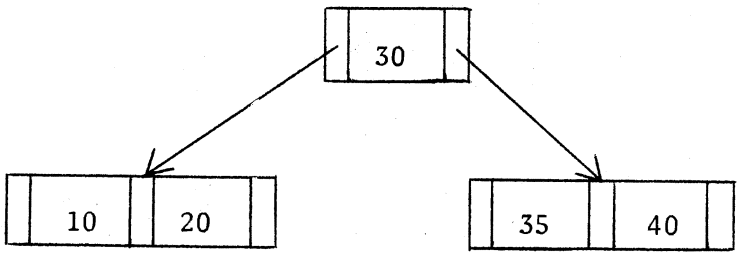
Insertion

Basic

As noted above, when a key is to be inserted into a B-tree, it will be placed into a leaf node. This is in contrast to a binary tree in which an insertion always causes the creation of a new node and possibly a new level. Figure 14. illustrates the basic insertion



a.) Before Inserting 35



b.) After Inserting 35

Figure 14. An Order Three B-tree Illustrating the Basic Insertion Process

process. Note that as the key 35 is inserted, key 40 had to be moved to provide room for the new key. This problem of data movement can be rather troublesome if done often and this should be kept in mind when the structure of the B-tree is being designed.

Two Way Split

If a basic insertion causes the node to become overfull, i.e., there are more keys than are allowed for the particular order, some method of processing the overfull node must be found such that the properties of a B-tree are not violated. One such method is called a two way split (called simply a split). In a split, the overfull node is broken into three parts, the middle key of the node and the two resulting sets of keys. An additional node is obtained and one of the two resulting strings is placed into it with the other string remaining in the original node. The node is thus split. To finish the process, the middle key and a pointer to the new node are propagated (inserted) into the predecessor of the original node. This process adds one node to the tree, and causes the original and new nodes to each be approximately half full. Since the process caused another insertion to occur in the next level up, the entire tree does not stabilize until the propagated key and link fit into a node without causing it to become overfull. A split is caused if key 25 is inserted into the tree of Figure 14 a.). The resulting tree is shown in Figure 15. In this case, key 20 is the propagated key and the node containing key 25 is the newly created node.

If the splitting propagates to the root node, and the root node is overfull, not only is a new node created to contain half of the

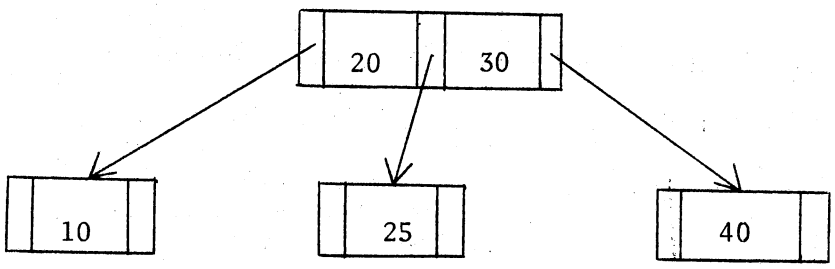


Figure 15. An Order Three B-tre Illustrating a Split

split node but an additional node is created to contain the propagated key. Thus a new root has been formed and the tree is one level higher. The veracity of the statement that a B-tree grows upward is thus demonstrated.

Storage utilization was mentioned earlier as a general term. It is now planned to specify exactly what is meant by utilization. The ratio of keys in a tree to the possible number of positions available in the nodes currently active in a tree is taken to be the utilization of the tree. Since, by definition, each node but the root is at least half full, one would expect the utilization to be at least 50%; it turns out that it is actually much larger. Van Doren (21) has shown that the asymptotic storage utilization of a B-tree with a large degree of branching will be $\log_e 2$ or about 69.3%. This assumes splitting for insertions.

Overflow

Another possible way to handle the problem of overfull nodes is called overflow. Overflow involves a redistribution of keys between the overfull node, its left or right sibling and the intervening key in the predecessor node for the two. The redistribution essentially requires one or more keys from the overfull node to be moved through the predecessor key slot into the sibling. The number of keys that are moved is a function of the programmers intuition since no empirical or theoretical work gives a sound base for a decision. Figure 16 shows the results if overflow is performed when key 25 is inserted into the tree in Figure 14 (a.). In this example

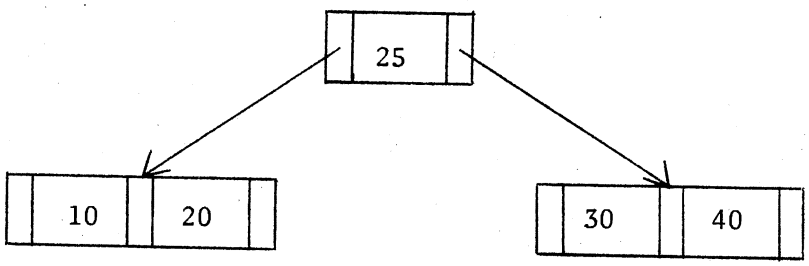


Figure 16. An Order Three B-tree Illustrating Overflow

the rightmost key is the overfull node, key 25 moved up to the predecessor node forcing key 30 to the sibling node. An overflow can only be accomplished if the sibling node is not full, however, both siblings can be checked before a split must be performed. In the

last two examples the same key was inserted into the same tree but a split caused a new node to be used whereas an overflow did not.

Overflows do not propagate. Once an overflow is performed, no more revision to the tree is necessary. Empirical evidence by Davis (5) indicates that overflow greatly increases the utilization that can be expected. On trees of order 48 a storage utilization of 85% was achieved.

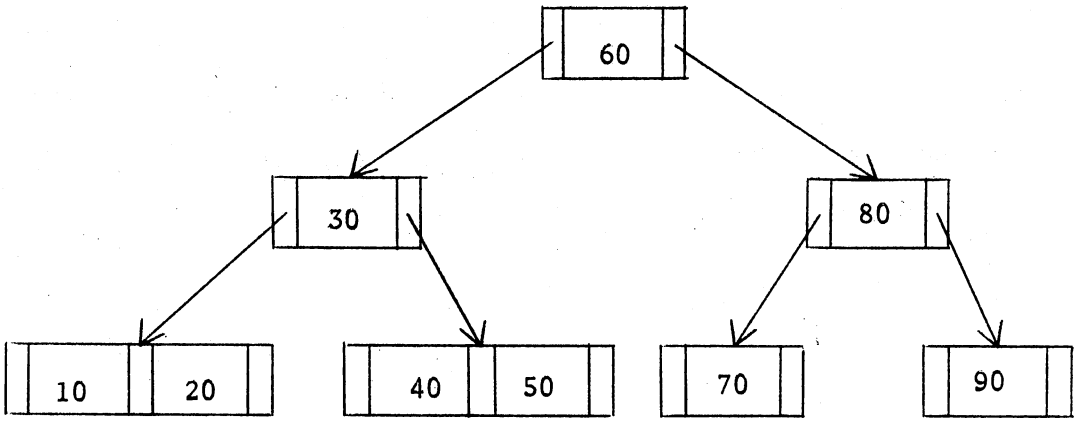
Overflow is a supplement to splitting. Overflow alone cannot be used to preserve the properties of a B-tree, but splitting can. Overflow is not necessary but since splits propagate and overflows do not and storage utilization is appreciably increased, overflow is recommended (1, 5, 14).

Deletion

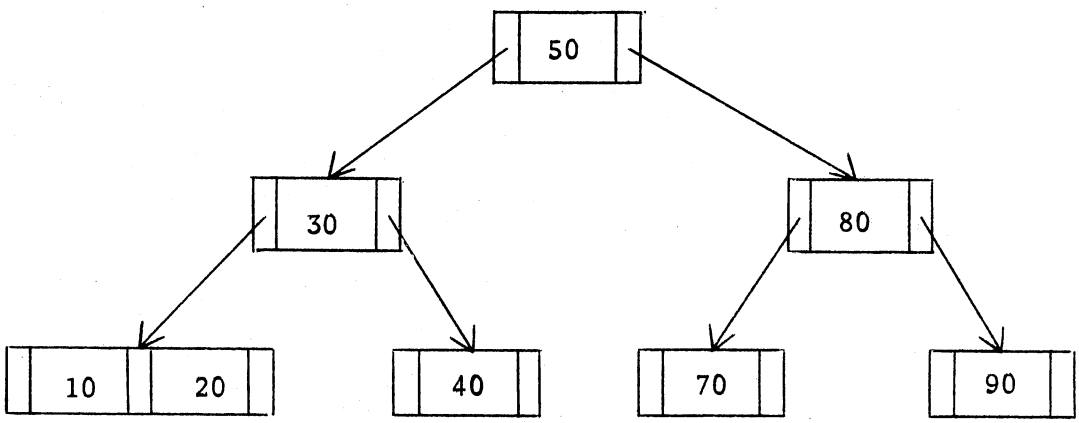
Basic

Deletion from a B-tree involves removing a key and necessarily one link from some node of the tree. Note that a deletion may come from some non-leaf node although for trees of large order most of the keys and hence most of the deletions will be from a leaf node. If the key is in a leaf, the normal deletion process is followed. If, however, the key is in a non-leaf node, deleting the key and a link (a link must be deleted or there will be two more links than keys in in the node) will also delete a subtree from the tree. Since this subtree may contain valuable information, this should be avoided. A solution is to exchange the key to be deleted with the next larger or next smaller key in the tree. This lexicographically larger or smaller key would come from a leaf node found by following the

leftmost or rightmost links, respectively, of the subtrees to the right and left of the key to be deleted. It really does not matter which key is chosen; the point is that after the exchange is made the tree is still in order and a deletion from a leaf node can be made. For an illustration of this, refer to Figure 17.



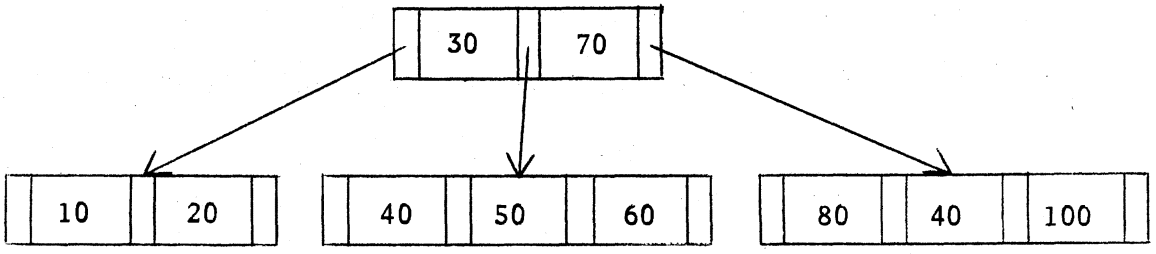
a.) Before Deletion of Key 60



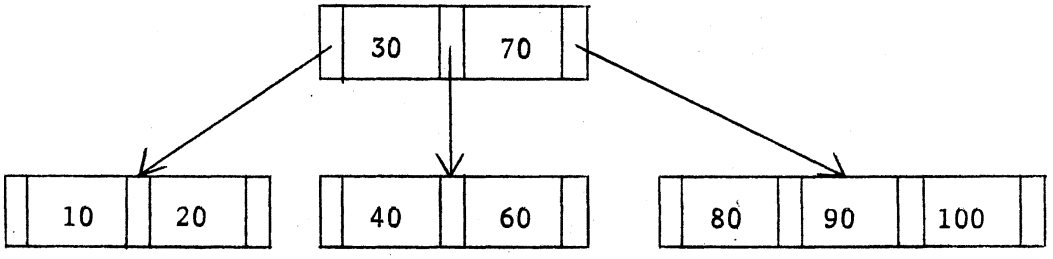
b.) After Deletion of Key 60

Figure 17. An Order Three B-Tree Illustrating Deletion from a Non-Leaf Node

The basic deletion process involves removing a key and link from a leaf node and squeezing out the hole created by the deletion. Figure 18 illustrates the basic deletion process. Note that key 60 had to be moved over in order to close the ranks in that node.



a.) Before Deleting Key 50



b.) After Deleting Key 50

Figure 18. An Order Five B-Tree Illustrating the Basic Deletion Process

Catenation

If, when a deletion is to be completed, the node becomes underfull, i.e., it contains fewer than the minimum number of keys allowed per node, special actions must be taken to again make the tree follow the guidelines for B-trees. One such action is called catenation. A catenation is essentially the reverse of a split. In a catenation, the underfull node and a sibling and the intervening key from the predecessor node are combined into one node. This reduces the number of nodes in the tree. Figure 19 illustrates the results when the key 20 is deleted from the B-tree in Figure 18(b.). Note that in

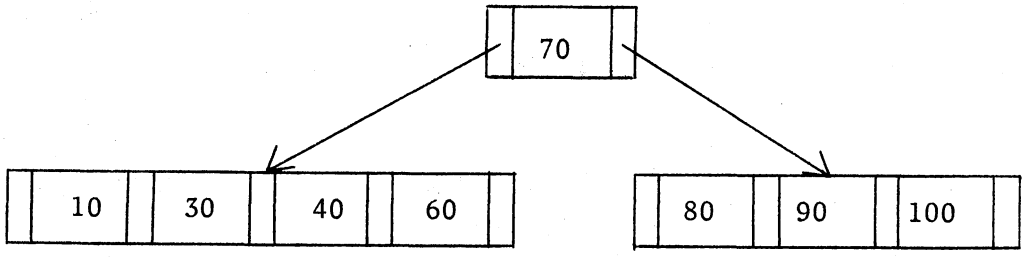


Figure 19. An Order Five B-tree Illustrating Catenation

order for catenation to be possible, the sum of the number of keys in the underfull node, the number of keys in the sibling, and the single key from the predecessor node must be strictly less than or equal to the maximum number of keys allowed per node. If this is not the case, then an overfull node results.

Just as with splits, catenations can propagate upward through the tree. Since one key from the predecessor node is removed for the catenation, the predecessor node may become underfull and require attention. However, unlike splitting, a choice can be made with catenations as to which sibling to catenate with. In the cases in which only one sibling exists or when only one sibling satisfies the number of key requirements there is no choice.

If the catenation process reaches all the way to the root node and the root only has one element, the catenation will cause a new root to be determined and two nodes to be returned to the available unused pool of nodes. When this happens, the number of levels in the tree is reduced by one. Again, this shows that the tree grows and shrinks at the root level.

Underflow

What can be done if the number of keys in the siblings of an underfull node are all too large to allow a catenation? In such a case, another action called underflow takes place. Underflow in practice if not in theory, is completely symmetric with overflow. The keys in the underfull node, the keys in the sibling, and the intervening key from the predecessor node are redistributed to produce an arrangement consistent with the definition of a B-tree. The analogy is so complete in fact that a single equalizing routine can be constructed to accomplish the redistribution in both cases. Figure 20 indicates the resulting tree configuration if key 10 is deleted from the tree in Figure 18 (a.). Note that key 30 has moved from the predecessor node to the underfull node and that key 40 has

moved up to the predecessor node. Underflow or overflow can be viewed as a step-by-step movement of single keys through the predecessor node until the desired distribution is reached. One would be ill-advised to actually implement the method as a stepwise approach, however.

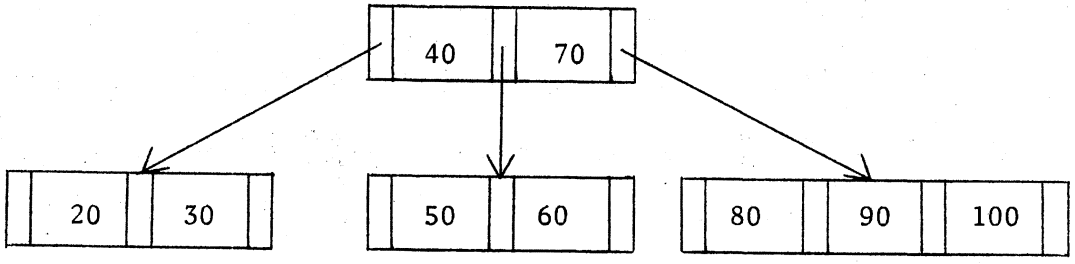


Figure 20. An Order Five B-tree Illustrating Underflow

In comparing the different updating techniques, one can readily see that catenation and underflow must both be implemented if the properties of B-trees are to be maintained. They are mutually exclusive operations with the number of keys in the siblings determining which method must be used. It is not quite the same story with splitting and overflow, however. Any overfull node can be properly handled with splitting whereas only certain situations allow overflow. When inserting, if there is a choice as to method of handling an overfull node, overflow should be used since it does not add to the number of nodes and will not be propagated. With deletions on the other hand, catenation is beneficial because it

reduces the number of nodes in the tree and can be propagated whereas underflow cannot be propagated. The programmer thus has some minor freedom in choosing combinations of techniques.

The methods presented here do not exhaust the possible ways to maintain B-trees. Three and four-way splitting, overflowing and underflowing to non-sibling peer nodes, and the use of variable length keys are some other possible factors to be considered when designing a B-tree scheme.

CHAPTER V

DESIGNING AN ISRS

System Objectives

This chapter presents the analysis and design considerations involved in one information storage and retrieval system. A great many of the data structures previously presented are contained in this system either directly as discussed or indirectly in hybrid data structures. It is hoped that the reader's understanding of some of the concepts discussed in the preceding chapters will be solidified by his following the example presented in this chapter.

Today's world is saturated with data. In a scientific field, so many new and valuable articles and books are published each year that an individual would be sorely pressed to keep up with new developments in his field and still have time for any productive work. The need to be aware of the state-of-the-art is, however, an important one. There must be some way to relieve the person from being required to use his own time and effort in researching information relevant to his interests. The speed and accuracy of a computer should be of great value in this effort. It is precisely this problem that the information storage and retrieval system presented here addresses.

There are thus two objectives for the information storage and retrieval system (abbreviated ISRS) in this chapter. It should serve

as an example of many types of data structures and it should be of some practical use in serving as an automated way of referencing desired pieces of information.

This ISRS pertains to articles taken from journals, magazines, etc. The articles have the common attributes of author, title, and journal. Journal is here taken to mean the source of the article, i.e., publication, volume, date, etc. There may be some articles which have several authors or perhaps an unknown author. Together the attributes serve to provide both an indication of the contents of the article and a guideline to locating the article. Although in many cases it would be helpful, an abstract of the content of an article is not considered here.

If put to general use this ISRS could contain many thousands of articles, too many for an individual to manually scan. The objective of the system is to structure the information such that particular subsets can be retrieved easily. The data structures used in the implementation of the system will serve as a major determining factor in the success of the system.

A reasonable idea as to the content of articles can often be deduced from the title of the articles. Although titles rarely give a complete view of the material contained within the article, they do serve to identify its major thrusts. This ISRS uses the keywords in the title of an article as the means of semantically differentiating between articles. Keyword means a word that is more intrinsically descriptive than widely used adjectives, prepositions, or nouns. Thus one matter the system must attend to is determining what words in the titles are indeed keywords. This can be a complex

problem (15) when one considers plurals, synonyms, and multiple occurrences of keywords. The approach taken here is to have a list of nonkeywords against which each word in a title is compared. If no match is found, the word is a keyword. Another point about this system is that as keywords are extracted from the titles, their position in the title is retained so that on later analysis the context of the keyword is available. This is termed KWIC (Keyword In Context) indexing and is opposed to KWOC (Keyword Out of Context) indexing.

One possible way to structure the data to allow for retrieval is to simply keep each article in a sequential file and search through the file completely for each request. Thus in order to locate all articles with the term "hashing" in the title would require a linear search of the entire file. This is entirely unreasonable if the file is large. It thus seems that some more complex, yet more efficient scheme should be devised.

The scheme chosen for this ISRS involves essentially two sets of interrelated data. One set, called an article file, contains the articles contained in the system. The second set of data, called a key file, serves as an index or directory to the article file. The key file does not have the extra information from the articles clogging up the data set so a more efficient search for particular values can be achieved. These two files are central to the ISRS and are discussed at length below.

Files

The article file serves as a repository for the articles in the

system. Certain properties of the data and the ISRS cause the file to be structured in a certain fashion. It is hoped that the following discussion gives both a description and rationale for the design chosen.

The article file is large. The sample set of data used to test the system contains about 7000 articles with provisions to allow for considerably more articles. Although the sample data contains articles primarily covering the computer science field, the system is not limited to such articles. The large number of articles and hence large storage requirement indicates that the file should reside on secondary storage. A further consideration is that the secondary storage should have direct access capabilities. This is a result of desiring to access any article in the file directly. Thus the file must be on secondary storage having direct access capabilities, in other words a disk.

Since this system is written in PL/1, the most useful file organization available is Regional (1). This allows for direct access based on a numeric relative record number which the system translates into an actual physical device address. A problem arises, however, in that Regional (1) data sets must have fixed length records, but the attributes of the articles are highly variable in length.

A possible solution is to store each attribute in a different record with the record size being large enough to contain the largest attribute in the whole file. This would encourage much wasted space and many records. An alternative is to catenate the attributes into one record and store it in a record with the record size being large enough to contain the largest catenated article. This has some

beneficial effects in lessening the variability in length since a long title may be matched with a short author attribute. There will still be much wasted space if the maximum size record is provided. The solution chosen is to provide a record size less than the maximum and if the catenated attributes cannot be stored in a single record to store the excess in another record and link it to the original. In this way any length article can be handled with less wasted space.

The size record chosen should be a compromise between requiring overflow records and the attendant extra disk accesses, and wasting space by making the records longer. Based on an observed mean of about 100 characters and standard deviation of around 25 characters, the percentage of overflow records can be calculated. An additional consideration is the optimal use of secondary storage taking into account interrecord gaps. Based on the above considerations, a record size of 139 characters (5 of which are used for an identifier, a tag, and a link) was chosen which corresponds to a 6% overflow rate. Figure 21 displays a view of the article file records. ID represents the identifier of the particular record. Regional (1) data sets contain records numbered sequentially from zero. LINK is the link to an overflow record (also used for storage management). TAG identifies the record as being the first record in a chain of records or as being an overflow record. INFO contains the catenated attributes stored in this record. The values in parentheses indicate the size in bytes (characters) of the particular field.

The second major file in this ISRS is termed the key file. It is actually a secondary key directory (the primary key is the iden-

tifier for the article in the article file). It is to this file that particular search requests are first directed. After it is determined what, if any, articles satisfy the search request, then the article file is consulted.

ID (2)	LINK (2)	TAG (1)	INFO (134)

Figure 21. Article File Record Layout

Several factors enter into the design considerations for the key file. First, the file is large (approximately 25000 keyword references). This implies that secondary storage should be used. Second, there is a high likelihood of multiple keywords per article. In fact, experience shows that there are between three and four keywords in each title. There is also a great likelihood of many nonunique keywords. Lastly, in order to allow for the most efficient processing of intersection and union keyword requests, the keyword references associated with each distinct keyword must be kept in order by reference number (the identifier of the containing article record). If the references are not in order, several passes may be required to process a request; otherwise a simple match or merge can handle the request. These constraints present two approaches:

1) Retain unique keywords in some tree structure and have each entry point to an entry in another file. This extra file is an

inverted list for each keyword.

2) Retain all keyword references in a B-tree.

In the first approach, the tree structure is necessary to allow for dynamic maintainence and still have reasonable search characteristics. The tree should be a B-tree because even retaining unique keys results in a tree too large to contain in main memory and any binary tree form (regular binary tree, AVL tree, etc.) would degrade if placed on secondary storage. This approach necessitates another file to contain the inverted lists. This presents additional problems in dealing with insertions and deletions from the inverted lists in addition to the inconvenience of an additional file. One alternative to this would be to cause the B-tree to have variable length nodes. This would involve more programming effort but might be profitable.

The second approach has the drawback of having multiple occurrences of some keywords. There are advantages, however. If the key for the B-tree is taken to be a keyword catenated with the article reference number, then an insertion automatically provides for the retention of proper order for intersection and union requests. The B-tree should have excellent search characteristics and will perform well in handling insertions and deletions. Another advantage is that no other file is needed. For these reasons approach 2 was selected. Figure 22 illustrates the layout for a node (record) of the key file.

The values in parentheses are the sizes in bytes of the fields. The maximum length for any keyword was chosen after examining many keywords and determining a length which would promote a high percentage of unique words. The other major choice is in the order of the

tree. One track on a 2314 can contain 7294 bytes of information. Therefore when allowing for a keyword of 18 characters, the maximum number of links (and hence order) would be 304. It turns out that the insertion and deletion algorithms are benefited if one link and key position are left unused. This means that the true usable order is 303 with 302 keys per node.

Node ID (2)	Avail Link (2)	Current Length (2)	B-tree Link (2)	Keyword (18)	Article Ref. No. (2)	Position in Title (2)
			304 times		303 times	

Where:

Node ID	- Identifier for the record (node)
Avail Link	- Link used in storage management
Current Length	- The number of keys currently in this node
Keyword	- A keyword extracted from a title
Article Ref. No.	- The article from which the keyword came
Position in Title	- The position in the title of the keyword

Figure 22. Key File Record Layout

The type of B-tree used in this system is one in which the keys contain pointers to another file. Note that since there are multiple keywords per title, storing the article in the B-tree would create much redundancy of data and that the order would necessarily decrease.

The values illustrated in the record layouts are the values used in the actual testing of the system. The length of the information portion of the article records and the keyword length and order

of the key file are set when the files are created and can be changed from system to system as discussed in the User's Guide in Appendix A.

Two additional permanent files are needed by the system. They are a count file and a nonkeyword file. The nonkeyword file contains all the words determined to be nonkeywords and against which all prospective keywords are compared. The count file contains the parameters necessary to allow the software to begin processing a new set of data at the point where the previous execution terminated. The count file contains the order of the tree, keyword length, etc., storage management information, and statistical values for the article file and keyword file.

Software

In order for the file structures to be effectively used there must be software to manipulate the files properly. There are four major programs in the software associated with this ISRS. They pertain to creation of files, editing of the articles, updating the article and key files, and reporting results of various retrieval requests.

There are actually two separate file creation routines. One makes use of the operating system sort and file creation utility to create the nonkeyword file. The result of this routine is a sequential file of the nonkeywords in alphabetical order. This file is used by the updating program to select keywords. The other file creation routine develops the frameworks for the article and key files and sets several values in the count file.

The edit program accepts card images describing the articles to be entered into the system and checks them for completeness and order. Articles passing the editing conditions are written onto a tape file for later use by the updating program. Articles not passing the editing conditions can be punched into cards for later correction and resubmittal to the editing program.

The update program performs four functions. The program can delete specific keywords, delete entire articles, insert entire articles, or insert specific keywords. In cases two and three, dealing with entire articles, the article file is altered and in all four the key file is altered. The B-tree maintainence algorithms utilize two-way splitting and left and right overflow for insertion and catenation and underflow, both checking left and right siblings, for deletions. As stated earlier the B-tree file is kept in order by keyword by article reference number.

The report program can produce reports of four forms:

1) A complete listing by article reference number of the contents of the article file.

2) A complete listing in alphabetical order of the keywords in the key file showing the frequency of occurrence of each keyword.

3) A complete listing in alphabetical order by keyword showing the permuted titles of the articles in the article file.

4) Listings of subsets of the article file which satisfy intersection and/or union keywords requests. The requests can be of an arbitrary number of keywords separated by and's and or's of arbitrary order.

Appendix A specifies details for communicating with the several

programs in this ISRS. For the user who desires a more detailed illustration of the logic of the programs, Appendix B is included, displaying program flowcharts. Appendix C, which contains sample outputs from the report program, can be consulted for examples of what to expect from the system.

CHAPTER VI

SUMMARY AND RECOMMENDATIONS

This project was undertaken with two major objectives in mind: one objective was to implement successfully and effectively an information storage and retrieval system which would provide access to relevant articles on particular subjects; the second objective was to investigate the use of B-trees in such a system.

The first objective has been accomplished through the file structures and software described in the previous chapter. The ISRS implemented provides the user both with the means to satisfy inquiries relatively easily and with a degree of control. There are, however, several improvements and additions to the system which would make the system have greater value or wider applicability.

One improvement to the existing system would change the method by which keywords are identified. With the present system, a prospective keyword is found to be a keyword only if it does not match any existing nonkeyword. This means that a copy of each form of a word regarded as a nonkeyword must be retained in the nonkeyword file. Much room in the nonkeyword file, and more importantly, much more effective keyword searches could be expected if basic stems of words were used. Thus "multilist", "multilists", and "multilisted" would be classified as having the same stem. Another improvement to the existing system would be to include a command which could

delete all occurrences of a keyword rather than require each occurrence to be deleted individually.

Another addition of possible value would be to retain counts of the occurrences of each nonkeyword. This facility in conjunction with the Frequency of Keyword Occurrences listing might point out beneficial changes of classification for certain words.

One problem which does not exist now but would need to be accounted for in a system of constantly and rapidly growing type is the expandability of files. The storage provided in the present system is sufficient for the foreseeable future in its current environment, but other implementations may not be so predictable. Protective features should be included to insure the integrity of data; to prevent the overrunning of current allocations, and to preserve data in the case of machine malfunctions.

The present systems provides for a secondary key directory based exclusively on the keywords extracted from the titles of the articles in the system. A system which would retain secondary key directories based on author or journal information would promote much greater freedom in locating specific information. An additional attribute which might be considered in a system of this type would be the physical location of a copy of the article. Thus one would know to look in Room 103 on shelf A4 for a specific article rather than having to search for the article. Other candidates for a secondary key directory would be references to abstracts or selected keywords and phrases contained in the text of articles. Keywords contained in titles are a convenient means of semantically defining the intent of an article but are not always all-inclusive.

It is anticipated that requests for listings of subsets of the total set of articles will be rather limited in complexity. Based on this assumption, the handling of Boolean intersection and union keyword requests on a lef-to-right priority basis should prove more than sufficient. In a more extensive system, possibly including several types of key directories, proper handling of more complex search requests would be imperative.

A final recommendation is that the ISRS reporting system would be very profitably implemented in an on-line environment. The easier access and quicker response of an on-line system should greatly increase the attractiveness of the system to a prospective user. The author would suggest the on-line implementation as a next step in creating an information storage and retrieval system which would be widely used.

The second objective, to investigate the use of B-trees in an ISRS, answered some old questions and posed some new ones.

The report by Davis (5) indicates that insertion using overflow into a B-tree of order 49 resulted in a utilization of approximately 85%. In this ISRS using a B-tree of order 303, a utilization of 86.9% for all nodes and of 87.4% for leaf nodes was obtained. This far exceeds the guaranteed utilization of 69.3% deserved by Van Doren (21) for large order B-trees using only two-way splitting. High utilization of active space is not the only advantage of B-trees, however. The tree produced by the test data contains over 27,000 key entries but requires only two levels and hence only two disk accesses as a maximum in order to reference any element. This should be contrasted with the same number of keys stored in an AVL tree which

would require a minimum of 15 levels and a corresponding number of disk accesses. An indexed sequential file with two levels of indexing on the other hand would require a minimum of three disk accesses per reference with additional accesses required for overflow records. Furthermore, the number of local transformations in the tree needed to maintain proper B-tree properties is relatively small in comparison with the number necessary for AVL trees. During the insertion of 27,299 keys only 91 two-way splits and 1632 overflows were required, an average of 0.063 transformations per insertion. An AVL tree will require an average of 0.45 transformations per insertion for a tree with 6400 keys (22). An indexed sequential file will require a transformation whenever a record is written in any overflow area.

The scheme to reduce data movement in maintaining a B-tree which was chosen for this ISRS is to retain permanently the root node of the tree. Since the tree only has two levels, transfers to or from secondary storage would involve only leaf nodes. In this implementation, the number of actual reads was reduced by almost 50% (from 59, 167 to 29,683) and the number of actual writes by over 2000 (from 31,073 to 28,885). By using gather writing exclusively, the number of actual reads is not reduced whereas in this simple scheme, the number of actual reads was halved.

The study of B-trees is quite open for investigation into the benefits of these techniques. It may be possible to strike effectively a compromise between paging and gather writing which would be of greater value than either alone. The present system could be used as a test vehicle to this end since the UPDATE program captures all calls for reads and writes and the appropriate routines could be written

and substituted directly into the UPDATE program. Additionally, a framework is established in the present system in which gather writing could be implemented by simply substituting a routine for the WRITE_NODE routine.

Even though it is necessary to manipulate an article by stripping the keywords out before insertions can occur, the system is still able to perform 3.25 keyword insertions per second on the average. The principal time effectiveness of the system shows up, however, in the retrieving of subsets of articles. Keyword retrieval requests are satisfied, including system overhead time, in an average of less than 0.5 seconds. This is certainly adequate for a batch system and would quite likely serve well in an on-line system.

As a helpful warning to other programmers using IBM's PL/1 compiler, the author would like to mention several restrictions and unimplemented features which were found to be troublesome during the implementation of this system. These conditions are listed below:

1) A READ operation cannot have an array element as its destination field.

2) Pointer qualifiers cannot be elements of a based structure.

3) Assignments of cross sections of arrays are disallowed.

4) A cross section of an array of structures cannot be passed as an argument to a subprogram.

5) In an array of structures, bound information is not available except when references are made to elementary items.

6) Based structures do not have sufficient facilities for handling variably dimensioned arrays.

The list, of course, does not contain all difficulties in PL/1 but

the ones listed were encountered and proved to be troublesome at best. The sixth restriction is by far the most difficult to surmount. Based structures are quite useful, but the inability to have them change in size depending on the environment, as non-based structures can, places a severe limitation on their usefulness and generality.

The information storage and retrieval system developed as a portion of this project can be of much value if it is utilized. The data in the present system describes articles almost exclusively oriented to the computer science field. This is not at all a system restriction for the system could handle equally well data from any discipline. The reader is therefore encouraged to make use of the system and to possibly add to it. Only if the system is utilized can it be said that is it truly implemented.

SELECTED BIBLIOGRAPHY

(1) Bayer, R. and E. McCreight. "Organization and Maintainence of Large Ordered Indexes." Acta Informatica, 1 (1972), 173-189.

(2) Berztiss, A. T. Data Structures Theory and Practice. New York: Academic Press, 1971.

(3) Bobeck, Andrew H. and H. E. D. Scovil. "Magnetic Bubbles." Scientific American, Vol. 224 (June, 1973), 78-90.

(4) Chapin, Ned. Data Structures for Better Programming and Faster Operations. New York: Association for Computing Machinery, 1971.

(5) Davis, William S. "Empirical Behavior of B-Trees." (unpub. Masters thesis, Oklahoma State University, 1974.)

(6) Dodd, George G. "Elements of Data Management." Computing Surveys, 1, 2 (June, 1969), 117-133.

(7) Fisher, D. D. Data Structures and Programming Languages (unpublished class notes). Stillwater, Oklahoma: Oklahoma State University, 1974.

(8) Flores, Ivan. Data Structure and Management. Englewood Cliffs: Prentice-Hall, 1970.

(9) Harary, Frank, Robert Z. Norman and Darwin Cartwright. Structural Models: An Introduction to the Theory of Directed Graphs. New York: John Wiley and Sons, 1965.

(10) Harrison, Malcolm C. Data-Structures and Programming. Glenview: Scott, Foresman and Company, 1973.

(11) Introduction to IBM Direct-Access Storage Devices and Organization Methods, (GC20-1649-6). New York: International Business Machines Corporation, 1974.

(12) Iverson, Kenneth E. A Programming Language. New York: John Wiley and Sons, 1962.

(13) Knuth, D. E. The Art of Computer Programming, Vol. 3. Reading: Addison-Wesley, 1973.

(14) Knuth, D. E. The Art of Computer Programming, Vol. 3. Reading: Addison-Wesley, 1973.

(15) Lefkovitz, David. File Structures for On-Line Systems. New York: Spartan Books, 1969.

(16) Maurer, W. D. and T. G. Lewis. "Hash Table Methods." Computing Surveys, 7, 1 (March, 1975), 5-19.

(17) Morris R. "Scatter Storage Techniques." Comm. ACM, 11, 1 (Jan., 1968), 38-43.

(18) Nievergelt, J. "Binary Search Trees and File Organization." Computing Surveys, 6, 3 (September, 1974), 195-207.

(19) Sherman, Philip M. Techniques in Computer Programming. Englewood Cliffs: Prentice-Hall, 1970.

(20) Stone, Harold S. Introduction to Computer Organization and Data Structures. New York: McGraw-Hill, 1972.

(21) Van Doren, J. R. Data and Storage Structures (unpublished class notes). Stillwater, Oklahoma: Oklahoma State University, 1975.

(22) Van Doren, J. R. and J. L. Gray. "An Algorithm for Maintaining Dynamic AVL Trees." Information Systems, Vol. 4. New York: Plenum Press, 1974, 161-180.

(23) Wilde, Daniel U. An Introduction to Computing. Englewood Cliffs: Prentice-Hall, 1973.

APPENDIX A

USER'S GUIDE

APPENDIX A

USER'S GUIDE

This is intended to provide the user with guidelines and specifics in how to use this ISRS. Throughout the discussions of the programs it may prove helpful to refer to the Input/Output Schematic Diagrams for the programs, Figure 24, which illustrate the logical relations of the files associated with each program. The symbols chosen in Figure 24 to depict the type of file are not absolute. In other environments the NONKEY file might be on tape, for instance. Figure 25 is also included to help describe the files.

File Creation

This ISRS requires four files to be available during execution of the file updating and report generation programs. One of these files is created using operating system utility programs and the others are created by the PL/1 program CREATE.

The first file, NONKEY, consists of the nonkeywords with which each prospective keyword is to be compared. Input to this sorting and file creation utility package is a set of card images containing a single nonkeyword per card beginning in column 1. The programming presently allows for words of length 18 characters.

The program CREATE generates three files which are of major importance to the system. These are the count file, COUNT, the key

directory file, KEY, and the article file, ARTICLE. A single parameter card is used to specify the particular implementation attributes for the KEY and ARTICLE files. The parameter card has the following fields:

Columns 1 - 10 The B-tree order

Columns 11 - 20 Number of records in KEY file

Columns 21 - 30 The maximum keyword length

Columns 31 - 40 The length of article information portion of each ARTICLE file record

Columns 41 - 50 Number of records in ARTICLE file

Parameters 1 and 3 determine the size of each record in the KEY file and parameter 2 determines the number of records in the KEY file. Parameters 4 and 5 determine the size and number of records in the ARTICLE file.

The size of each KEY file record can be determined by the following relation:

$$\text{Size of KEY record} = (KL + 6)\ ORDER + 8 \qquad (2)$$

where KL is parameter 3 and order is parameter 1. For efficiency it is suggested that this value be as close to one track in size as possible. KL, in order to provide for proper boundary alignment, should be even. The value for ORDER is the order of the B-tree that is actually used by the algorithms. The program automatically creates one additional key and link field in each node for working area but this is not to be included in the parameter value.

The size of each ARTICLE file record can be determined by the following relation:

$$\text{Size of ARTICLE record} = LAI + 5 \qquad (3)$$

where LAI is parameter 4. This size should be chosen to maximize the utilization of secondary storage space and to minimize wasted space in considering overflow records (articles whose information content is to large to be stored in one record).

The CREATE program sets all values in the COUNT file to begin processing immediately and creates the framewords of the KEY and ARTICLE files. No actual article information is placed into either the KEY or ARTICLE files by the CREATE program.

Article Editing

The program, EDIT, accepts card images describing articles and checks them for completeness, rejecting incomplete ones and passing complete ones. The card images to be edited can come from either or both of two source files, ARTCRDS and ERRCRDS. Both files have 80 character records with fields as below:

Col 1 — Attribute identifier (A-Author, T-Title, J-Journal)

Col 2 — Sequence number within attribute

Col 3 - 13 — Article identifier consists of

- Author's last name - 4 characters
- Author's first and middle initials - 2 characters
- First letter of each of first three words in title - 3 characters
- Year of publication - 2 digits

Col 14 - 80 — Article descripts field

These cards are used as input to this system as they were prepared for another system, and hence the format is as it was specified for that

previous system.

The ARTCRDS file is designed to be primary input to the edit program with ERRCRDS being corrected cards not passing the edit conditions in a previous run and being recycled.

A single parameter card from the file PARM is used to specify whether all article cards not passing the edit conditions are to be punched into cards. This option is chosen by punching 'YES' in columns 8 - 10. Whether the option is chosen or not, a listing is produced showing all articles found to be incomplete.

The file OKARTS represents all articles which did meet all conditions for acceptance. It is shown as a tape file however any medium having variable length record capabilities would suffice. The information describing the articles is taken from the card images and catenated together to constitute an OKARTS file record. This file is used as input to the file updating program when articles are to be added to the system.

File Updating

The program, UPDATE, performs maintainence functions on the KEY and ARTICLE files. Changes in these files also cause changes in values of the file, COUNT. The NONKEY file is used by several of the maintainence functions but is not altered by the UPDATE program. The NONKEY file can only be changed by changing the source data and rerunning the create NONKEY file utility routine. The UPDATE program can perform any number of any or all of the four functions described below. The order of acceptance of the functions by the program is as below.

The first two functions provide for the user to delete specific keyword references or entire articles from the system. A keyword reference is taken to mean a single entry in the KEY file.

To delete a specific keyword reference the user should include in the KEYWDDL file data set a card containing the following:

Columns 1 - 40 Keyword (left-justified)

Columns 41 - 50 Article Reference Number

Any number of commands will be accepted with unmatched requests being ignored. The Article Reference Number can most easily be found for an article by locating it in the Articles File by Reference Number listing illustrated in Appendix C and described in the next section of this appendix.

Entire articles can be deleted by specifying the Article Reference Number of the article to be deleted. This number should be punched in columns 1 - 10 of a data card in the ARTDL file. Again any number of articles can be deleted and any not found requests are ignored. This function deletes the article from the ARTICLE file and all references to it from the KEY file.

Articles can be added to the system through the ARTIN file. This file should be the most used one for a growing system and is usually the output from a run of the program, EDIT. This functions inserts the article into the ARTICLE file and also inserts all references to it into the KEY file. Input through this file should be in variable length records with the author attribute followed by the title attribute followed by the journal attribute with each attribute having a terminating '$'.

The last function available to the user is to insert specific keyword references into the KEY file. A request of this type is as follows:

Columns	1 - 40	Keyword (left-justified)
Columns	41 - 50	Article Reference Number
Columns	51 - 60	Position of the start of the keyword in the title
Column	61	Force insertion code

The third value indicates the position in the title of the article of the specific keyword to be inserted. This is used in the KWIC report generations. Any nonblank character in column 61 will cause the keyword reference to be inserted regardless of whether the keyword is in fact a keyword or not otherwise a keyword detected to be a nonkeyword will be rejected.

Figure 23 illustrates typical requests for functions 1, 2, and 4. Input for function 3 is generated by the program EDIT and is not shown.

Report Generation

The program, REPORT, can furnish any or all of four possible reports of the contents of the ISRS. These reports are titled as follows:

1) Article File by Reference Number
2) Titles in Article File Permuted by Keyword
3) Frequency of Keyword Occurrences
4) Article File Interest Subset Selection

```
         11111111112222222222333333333344444444455555555556666
12345678901234567890123456789012345678901234567890123456789012345678901234567890123
```

IMPLEMENT	15
PRIMITIVE	91

a.) Keyword Delete Requests

9
23

b.) Article Delete Requests

MINICOMPUTER	12	19
DIGRAPH	30	5F

c.) Keyword Insert Requests

Figure 23. Sample File Updating Requests

Requests for any or all of the listings come from data cards in the REQUEST file. The first card in the REQUEST file has the following fields:

Columns 8 - 10 Report 1) option

Columns 18 - 20 Report 2) option

Columns 28 - 30 Report 3) option

If 'YES' is specified for any report, that report is generated. All other cards in the REQUEST file define requests to selectively report subsets of the ARTICLE file.

The Article File by Reference Number reports lists the entire contents of the ARTICLE file. The Reference Number is the identifier of the record in the ARTICLE file containing the first portion of the article and thus due to overflow records, there may be some gaps in the sequential listing of reference numbers.

The Titles in the Article File Permuted by Keyword report presents a KWIC (Keyword in Context) view of each article in the system. There is one entry in this listing for each entry in the KEY file.

The Frequency of Keyword Occurrences report lists the keywords contained in the KEY file in alphabetical order showing the number of occurrences of each keyword.

The fourth report illustrates the real usefulness of the system. Using this facility an individual can specify his interest by selecting a set of keywords and let the system find the articles satisfying that interest. Each request (the program will accommodate multiple requests) can consist of an arbitrary number of keywords with each keyword being separated by an 'AND' specifying intersection of the two adjacent keyword subsets or 'OR' specifying union of the

two adjacent keyword subsets. Each request should be terminated by a '$' but may extend over any number of card boundaries. It is important that each request begin on a new card, however. As an example, suppose one wishes to locate all articles containing 'BUSINESS' and 'STATISTICS' in their titles. The following request would be appropriate:

```
BUSINESS AND STATISTICS $
```

Intervening blanks are ignored between debiniters. Likewise, if desired to locate all articles containing 'STRUCTURES' or 'MACHINES' in their titles, the following request would be appropriate:

```
STRUCTURES OR MACHINES $
```

More complex requests can be established, however, they are evaluated exactly as less complex ones, i.e., the subsets satisfying the previous left to right subrequest is either merged or matched with the subset satisfying the keyword specified. Note that no parentheses are allowed and hence logically complex requests may need rewriting in order to be handled properly.

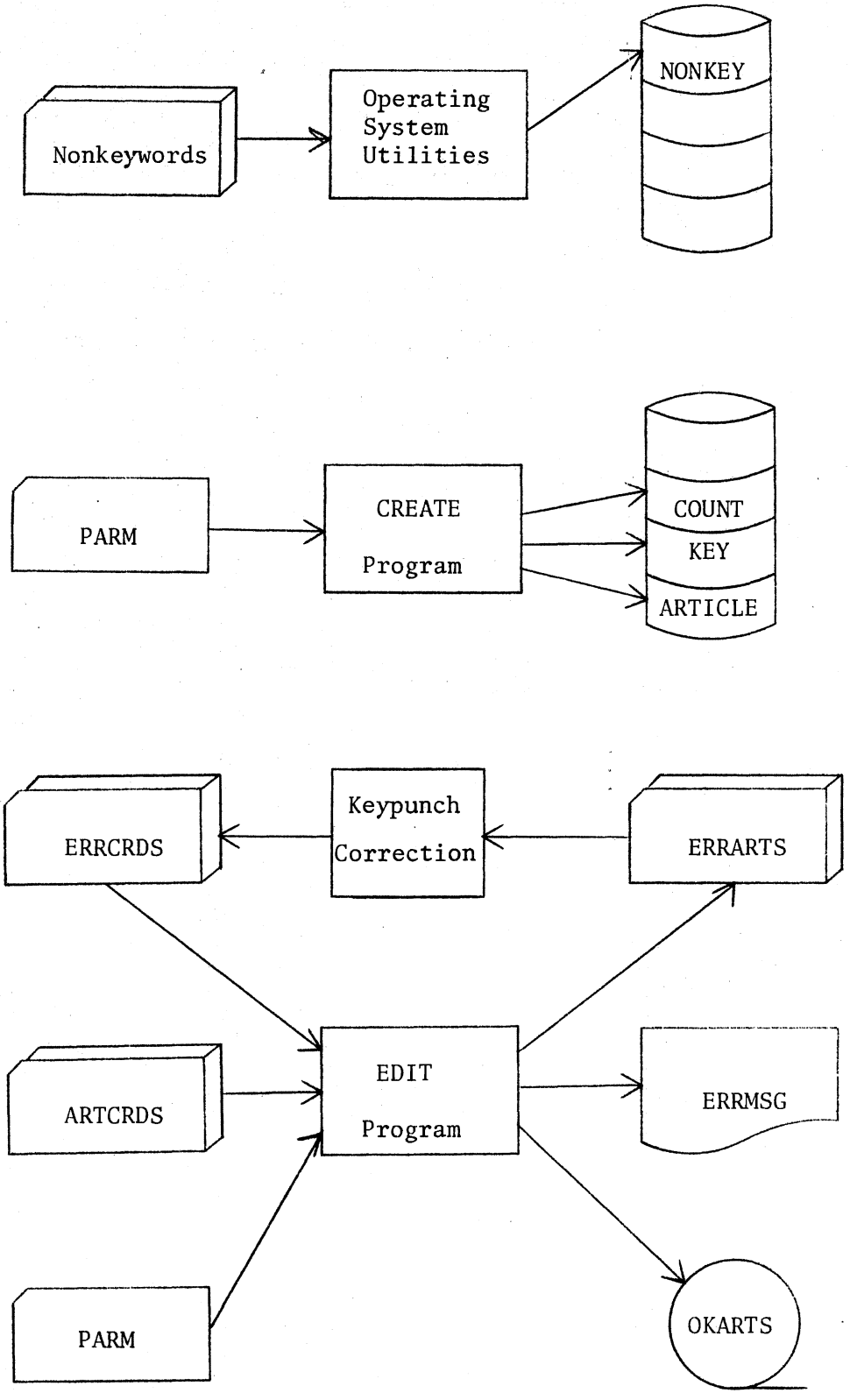


Figure 24. Input/Output Schematic Diagrams

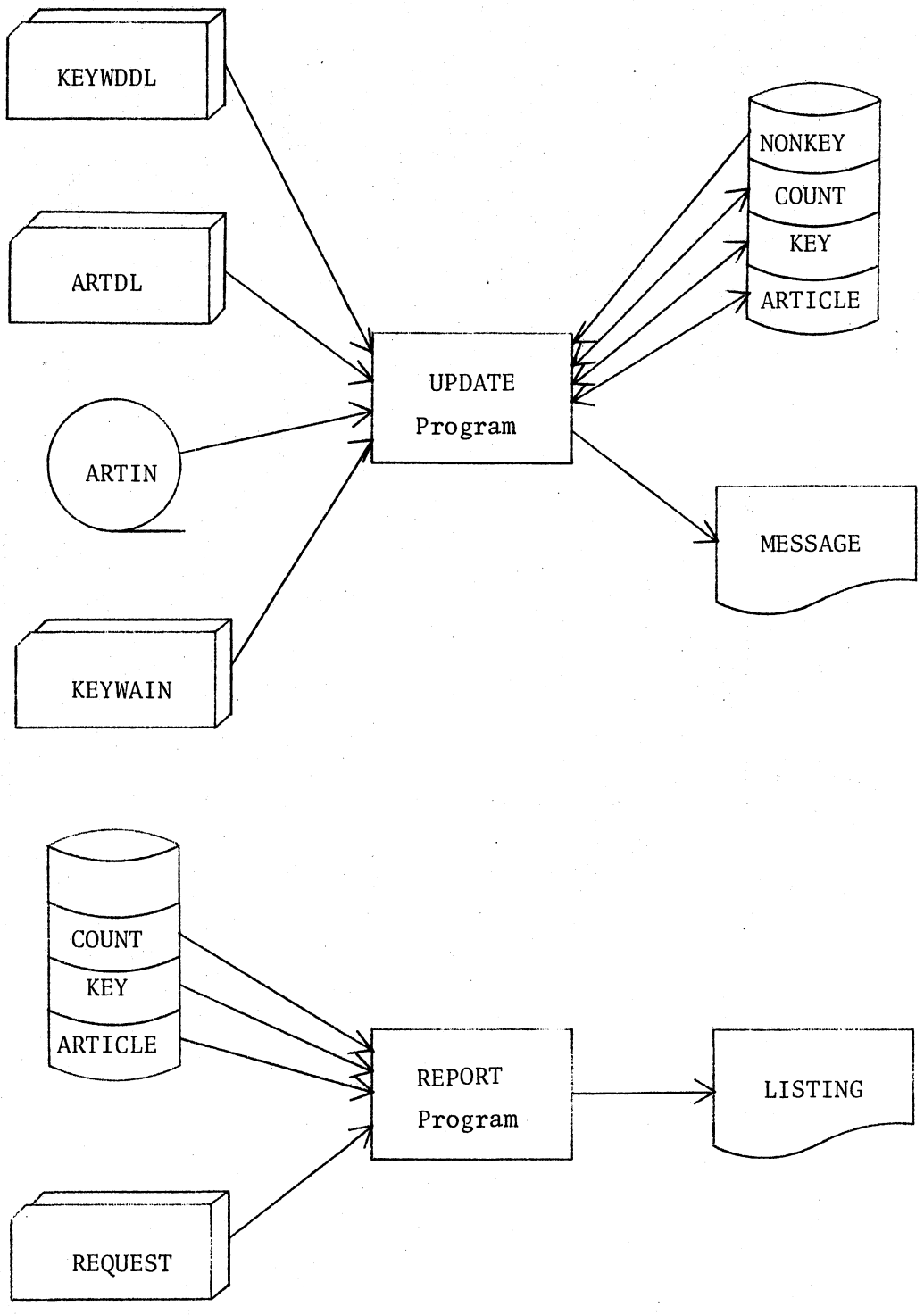


Figure 24. Continued

File	Description
Nonkeywords	Collection of nonkeywords to be included in the NONKEY file (entered through the SORTIN file of the sort utility)
NONKEY	File of nonkeywords in alphabetical order
PARM	For CREATE program, specifies parameters describing size and number of records in KEY and ARTICLE files
COUNT	Set of descriptors identifying the present status of the KEY and ARTICLE files
KEY	Index by keyword to the titles of articles in the ARTICLE file (organized as a B-tree)
ARTICLE	Set of articles available in the system
ERRCRDS	Set of corrected article descriptor cards rejected by a previous execution of the EDIT program
ARTCRDS	Set of article descriptor cards to be edited
PARM	For EDIT program, specifies whether rejected card images are to be punched
ERRARTS	Set of rejected article descriptor cards to be corrected and resubmitted through the ERRCRDS file
ERRMSG	Listing of rejected articles and causes for rejection and post editing statistics
OKARTS	Set of articles passed by the EDIT program (used as the ARTIN file for the UPDATE program)
KEYWDDL	Set of keyword deletion requests
ARTDL	Set of article deletion requests
ARTIN	Set of articles to be inserted
KEYWDIN	Set of keyword insertion requests

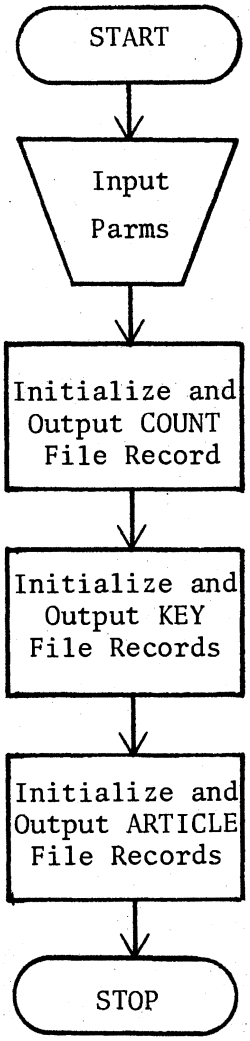
MESSAGE	Listing of post updating statistics and/or error messages
REQUEST	Set of requests for particular reports
LISTING	File containing all reports produced by the REPORT program

Figure 25. Descriptions of Files

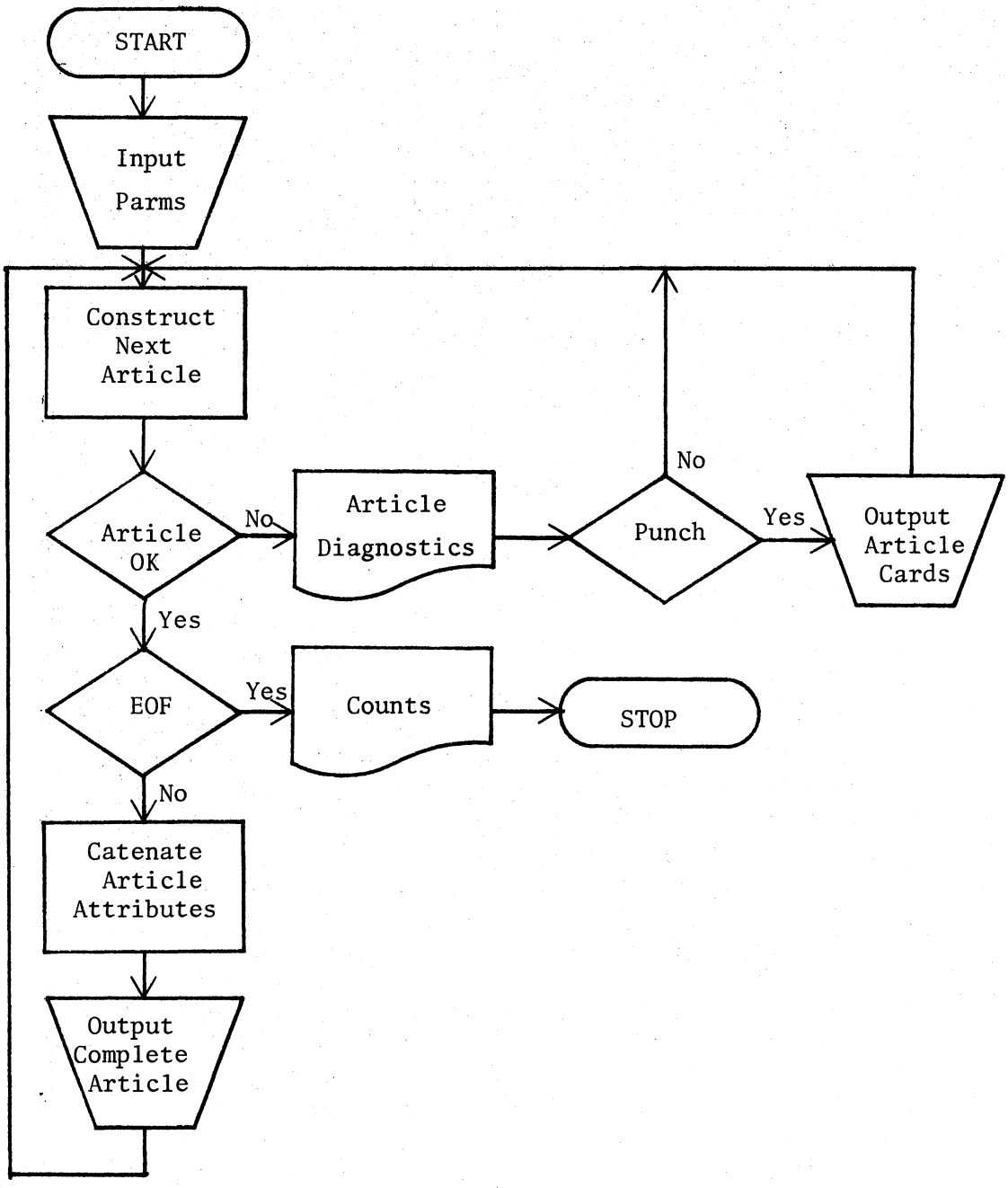
APPENDIX B

PROGRAM LOGIC FLOWCHARTS

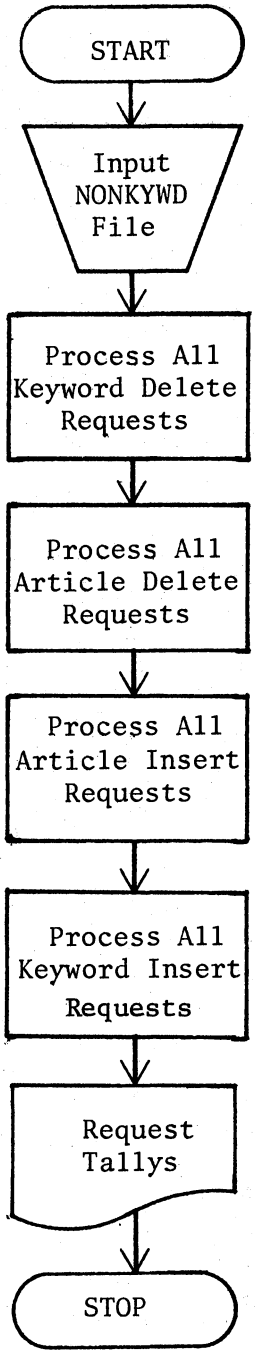
CREATE Program



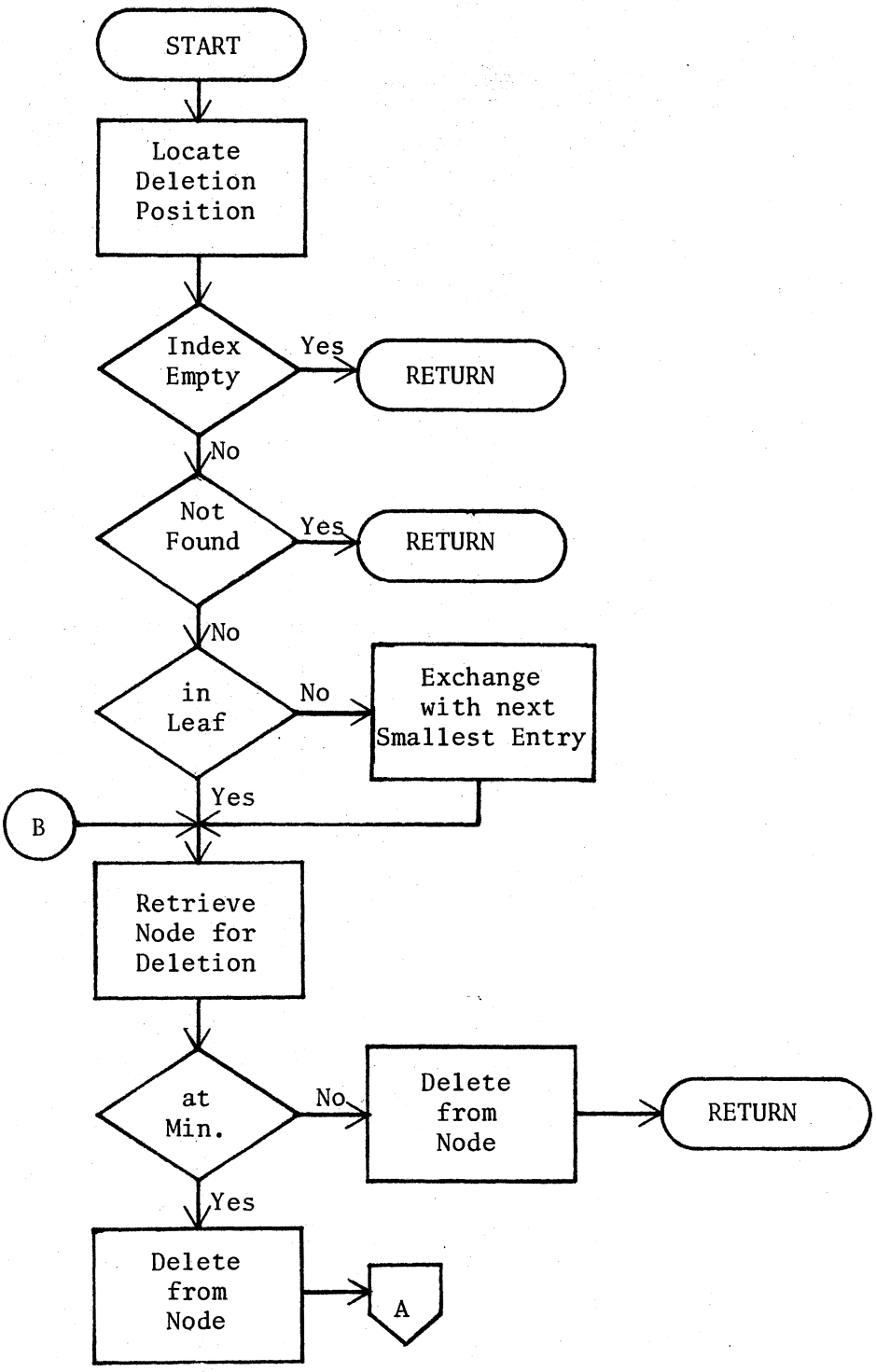
EDIT Program

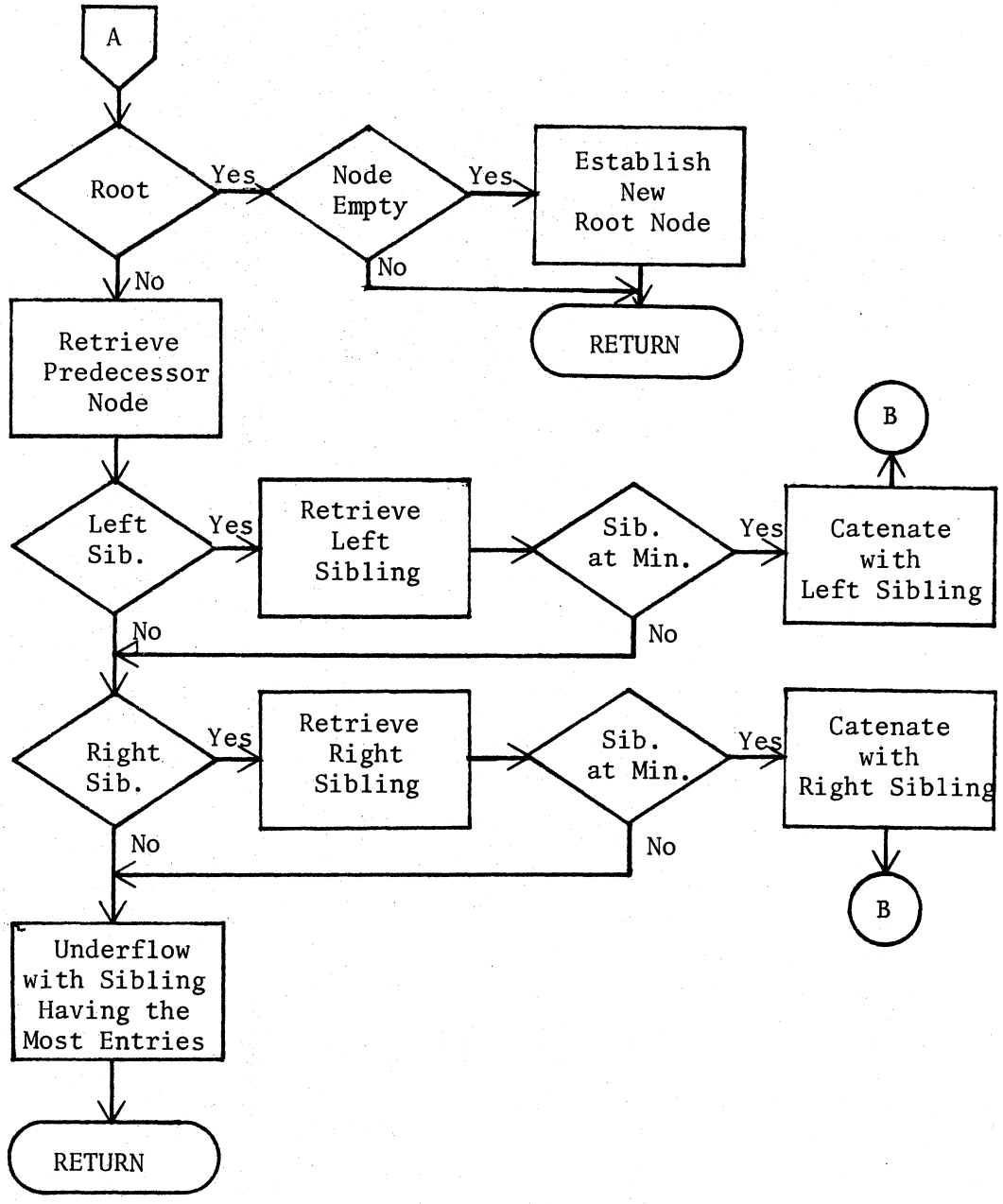


Update Program

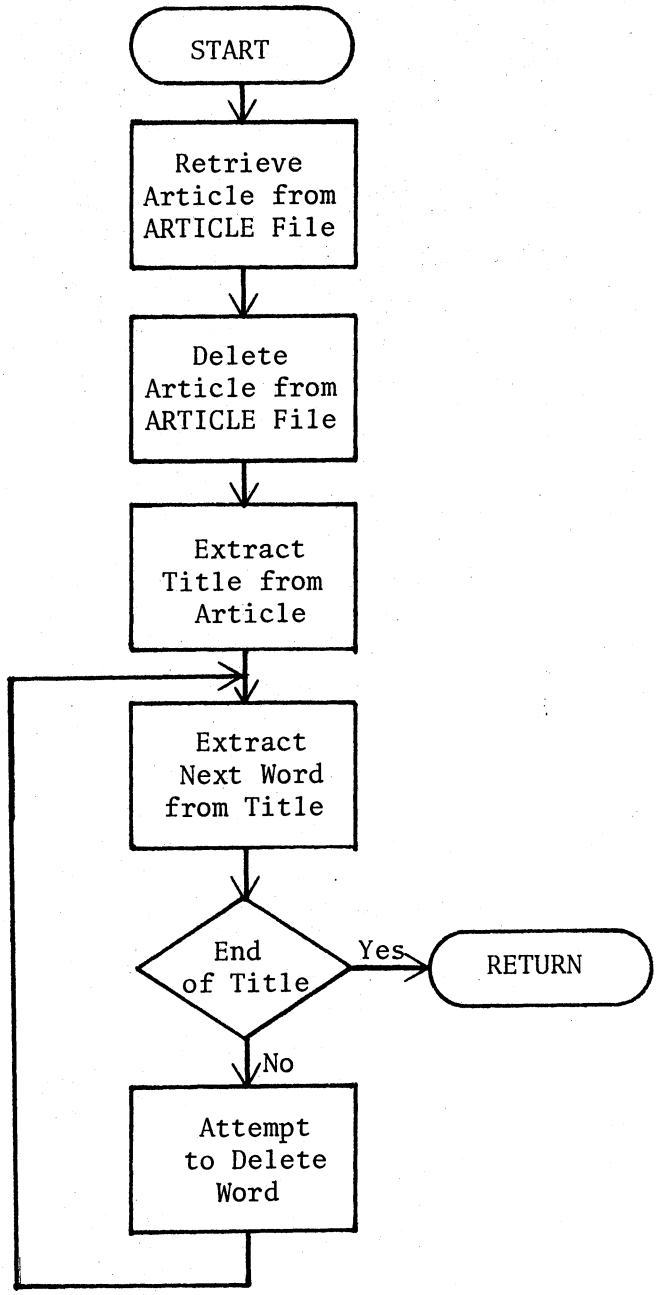


Keyword Deletion

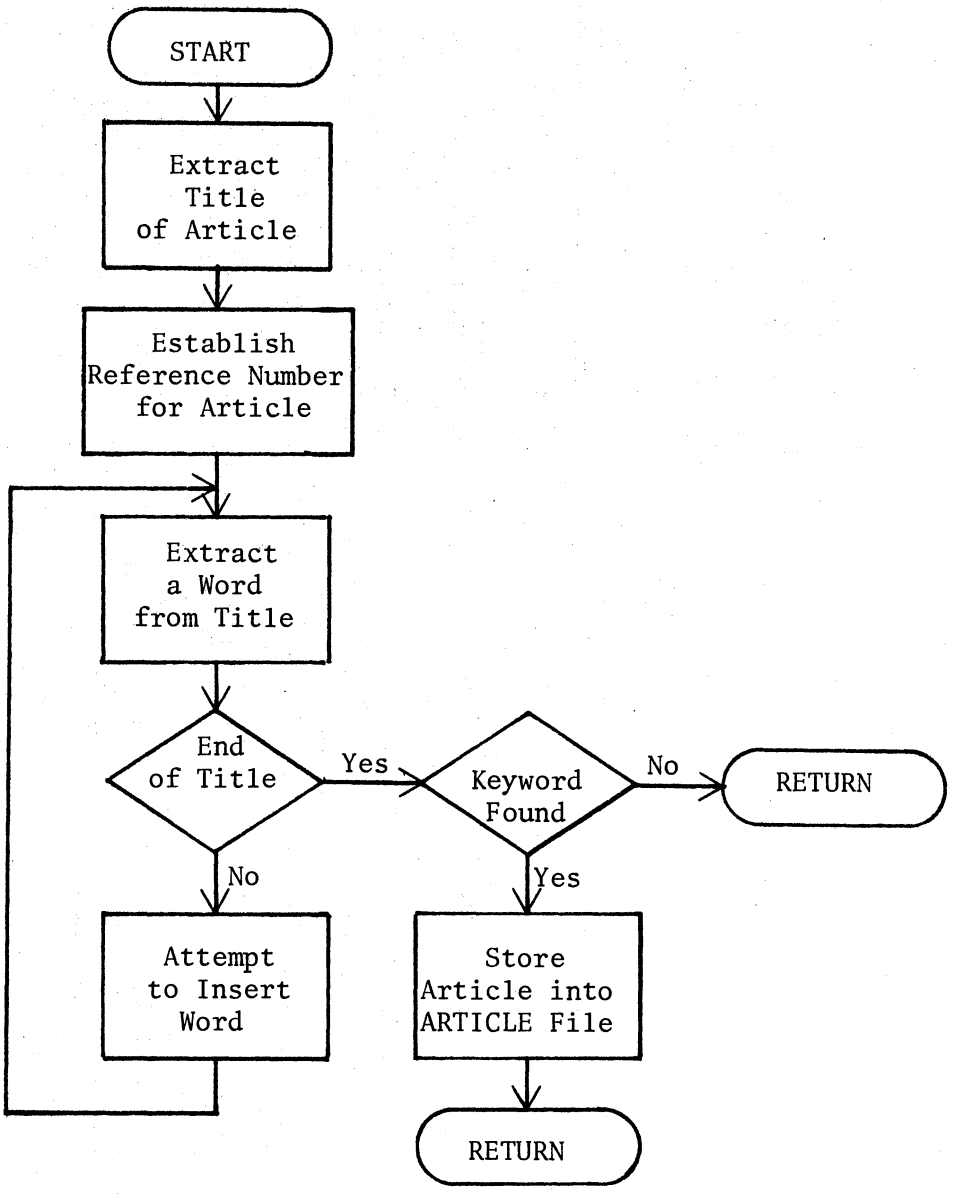



A
Root
Yes
Node Empty
Yes
Establish New Root Node
No
RETURN
No
Retrieve Predecessor Node
B
Left Sib.
Yes
Retrieve Left Sibling
Sib. at Min.
Yes
Catenate with Left Sibling
No
No
Right Sib.
Yes
Retrieve Right Sibling
Sib. at Min.
Yes
Catenate with Right Sibling
No
No
B
Underflow with Sibling Having the Most Entries
RETURN

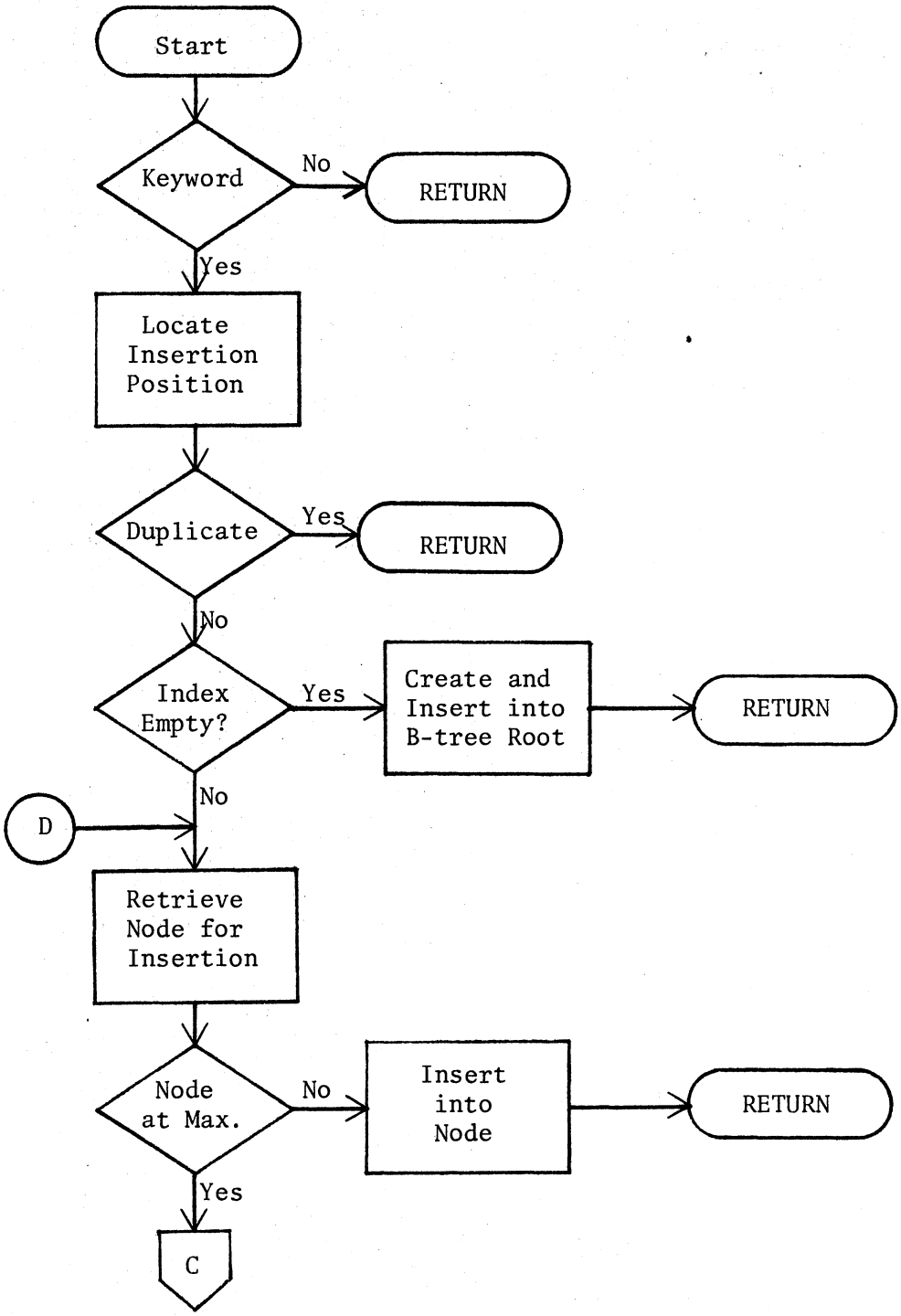
Article Deletion

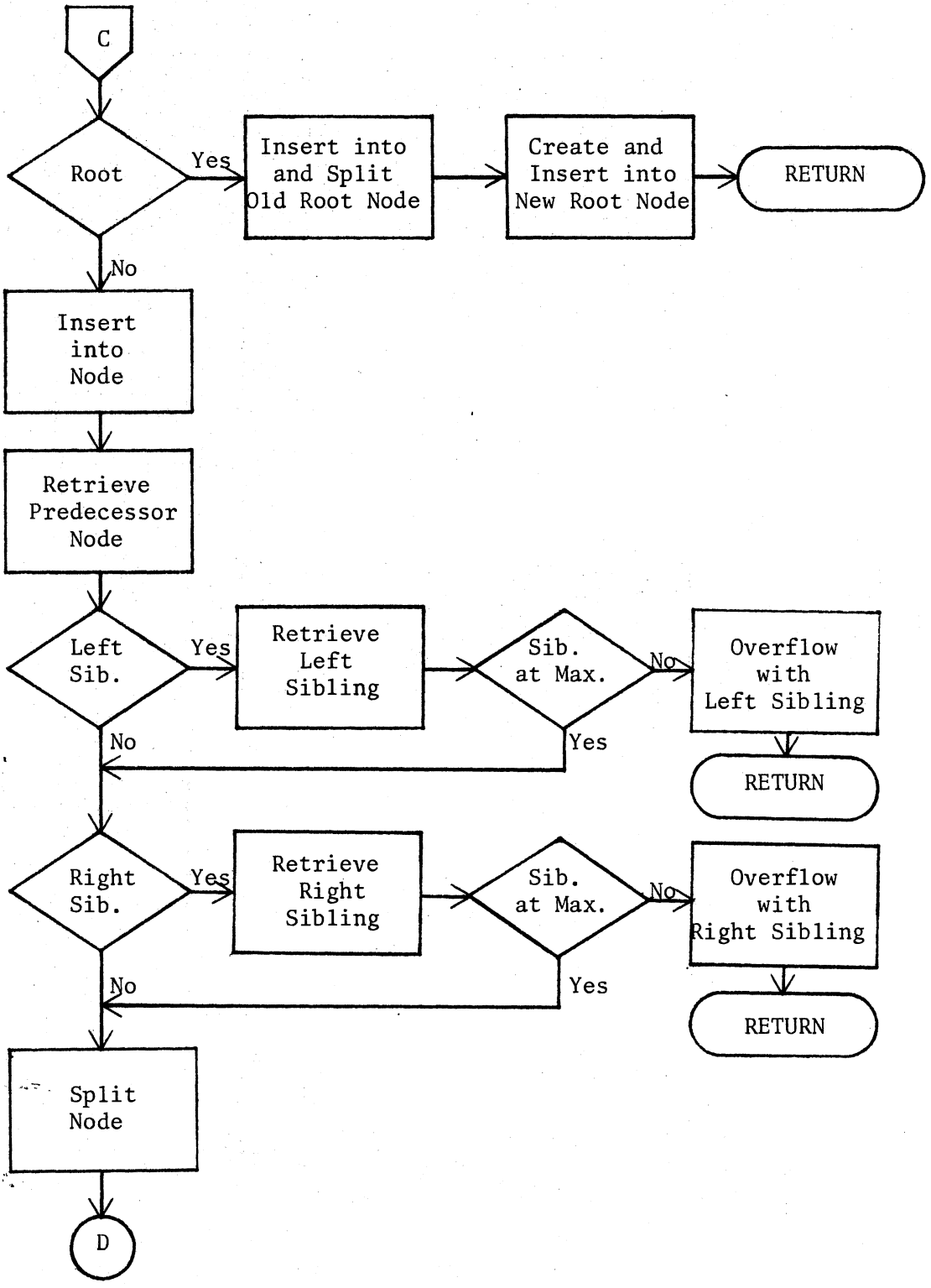


Article Insertion

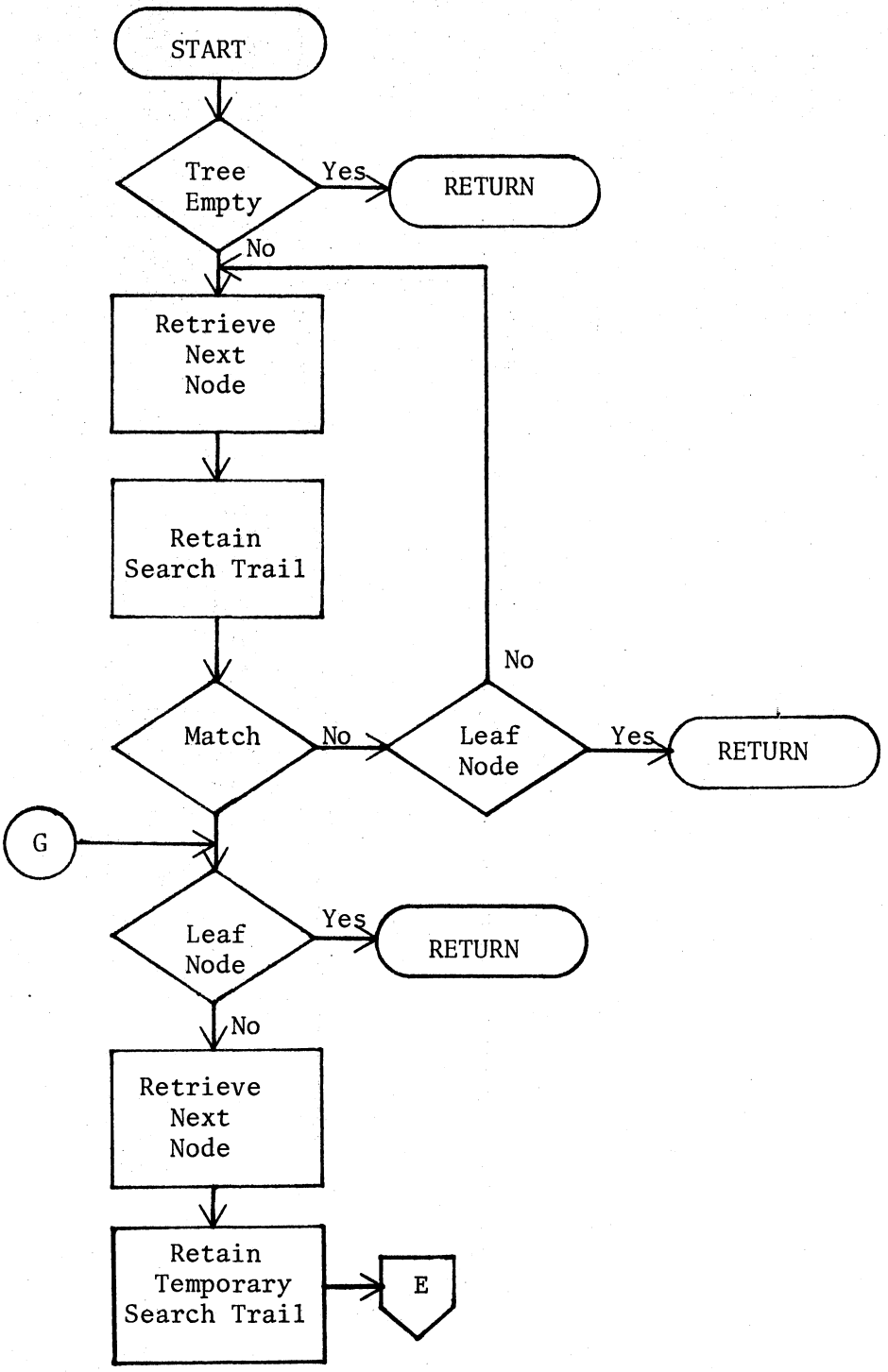


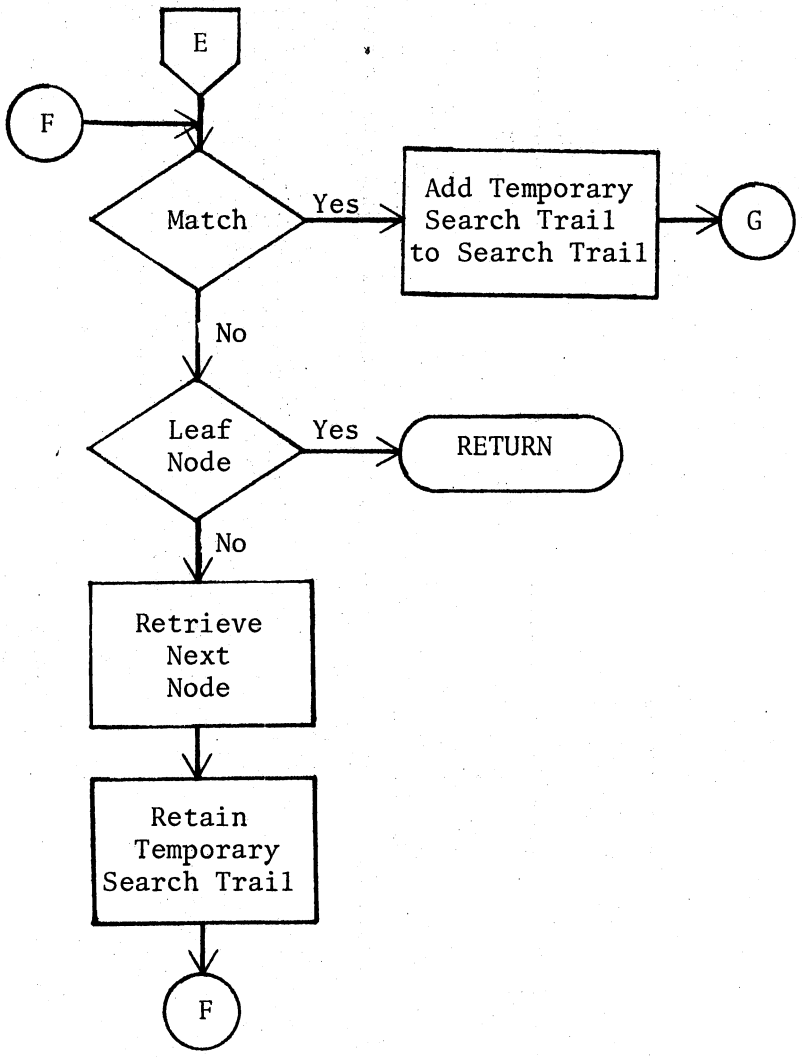
Keyword Insertion



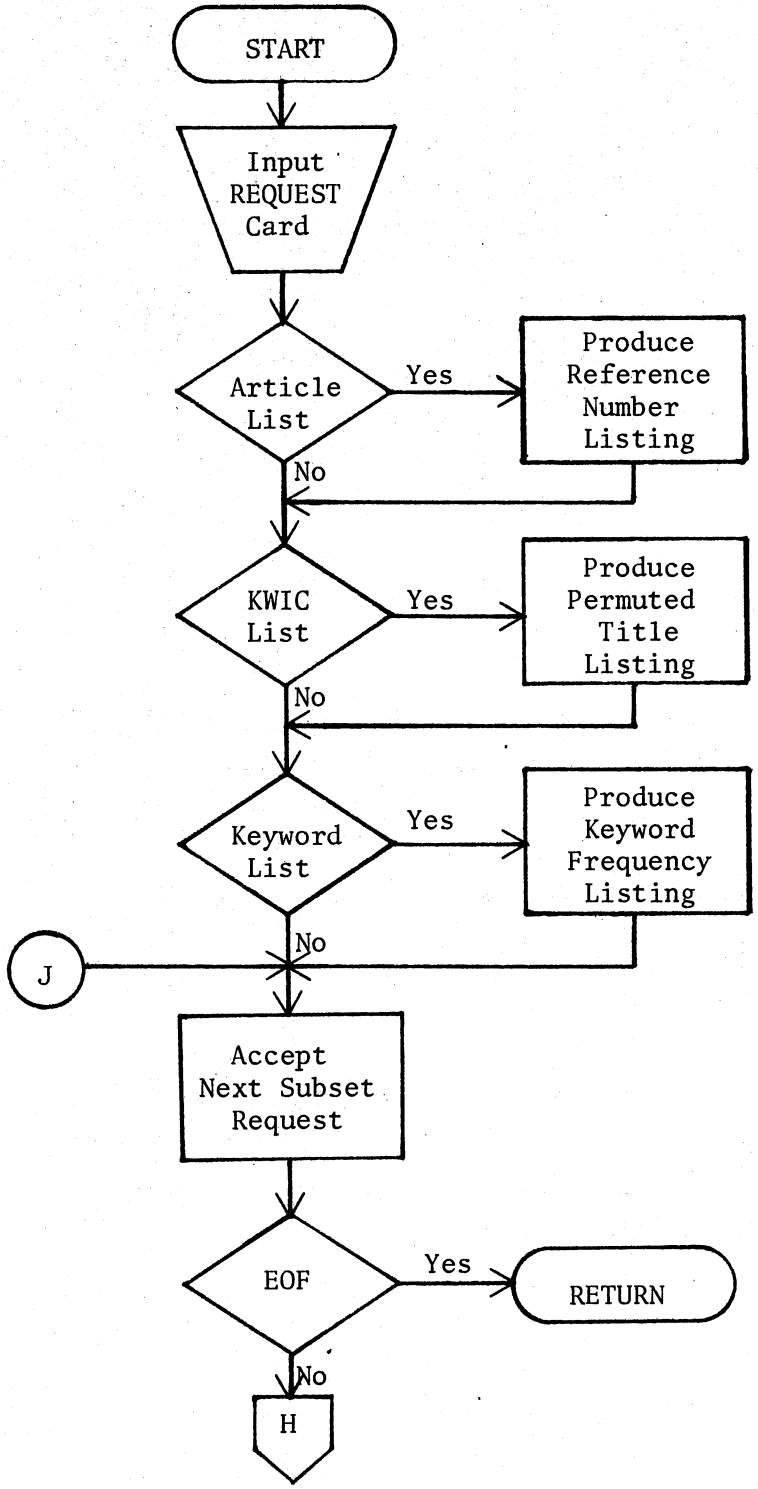

C
Root
Yes
Insert into and Split Old Root Node
Create and Insert into New Root Node
RETURN
No
Insert into Node
Retrieve Predecessor Node
Left Sib.
Yes
Retrieve Left Sibling
Sib. at Max.
No
Overflow with Left Sibling
Yes
No
RETURN
Right Sib.
Yes
Retrieve Right Sibling
Sib. at Max.
No
Overflow with Right Sibling
Yes
No
RETURN
Split Node
D

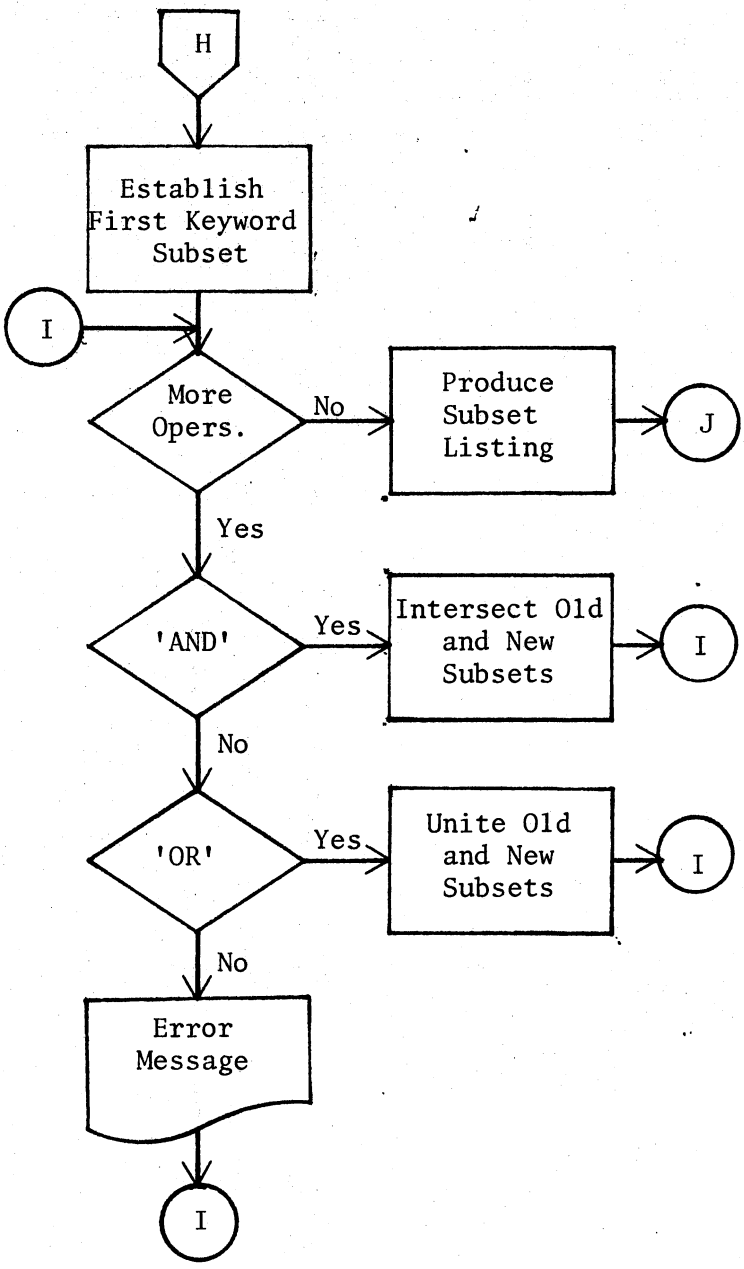
Search


START
Tree Empty
Yes
RETURN
No
Retrieve Next Node
Retain Search Trail
No
Match
No
Leaf Node
Yes
RETURN
G
Leaf Node
Yes
RETURN
No
Retrieve Next Node
Retain Temporary Search Trail
E


E
F
Match
Yes
Add Temporary Search Trail to Search Trail
G
No
Leaf Node
Yes
RETURN
No
Retrieve Next Node
Retain Temporary Search Trail
F

REPORT Program




H
Establish First Keyword Subset
I
More Opers.
No
Produce Subset Listing
J
Yes
'AND'
Yes
Intersect Old and New Subsets
I
No
'OR'
Yes
Unite Old and New Subsets
I
No
Error Message
I

APPENDIX C

SAMPLE REPORT PROGRAM OUTPUTS

```
                                ARTICLE FILE BY REFERENCE NUMBER

REFERENCE  AUTHOR---->  EACH ATTRIBUTE IS TERMINATED BY A $.
 NUMBER    TITLE---->   IF ANY ATTRIBUTE CANNOT BE PRINTED ON A SINGLE LINE, IT IS CONTINUED ON THE NEXT LINE
           JOURNAL-->         AFTER AN INDENTATION OF FIVE SPACES.

     0     LARMOUTH,J.$
           SCHEDULING FOR A SHARE OF THE MACHINE$
           SOFTWARE 5 NO.1(1975)P.29$

     1     ECKLUND,E.F.*EGGLETON,R.B.$
           PRIME FACTORS OF CONSECUTIVE INTEGERS$
           AM. MATH. MONTHLY 79 NO.10(1972)P.1082$

     2     DE LUCENA,C.J.P.*DE ALMEIDA CUNHA,L.F.$
           A MODELLING TECHNIQUE IN PROGRAMMING$
           PUC, CENTRO TECNICO CIENTIFICO SEPTEMBER NO.9/71(1971)$

     3     $
           THE RIGHT OF EQUAL ACCESS TO GOVERNMENT INFORMATION$
           COMPUTERS AND AUTOMATION 20 NO.4(1971)P.32$

     4     ANDERSON,J.W.*ATKINSON,M.P.*COLIN,A.J.T.*HAINSWORTH,D.J.* LISTER,A.M.$
           THE EVOLUTION OF AN OPERATING SYSTEM$
           COMPUTER BULLETIN 15 NO.6(1971)P.212$

     6     UTGOFF,V.A.*KASHYAP,R.L.$
           ON BEHAVIOR STRATEGY SOLUTIONS IN TWO-PERSON ZERO-SUM FINITE EXTENDED GAMES WITH IMPERFECT INFORMATION. PART
                I: A METHOD FOR DETERMINATION OF MINIMALLY COMPLEX BEHAVIOR STRATEGY SOLUTIONS$
           SIAM J. APPL. MATH., 22 NO.4(1972)P.648$

     8     $
           POCKET CALCULATOR$
           COMPUTER DECISIONS 4 NO.3(1972)P.42$

     9     GYLLSTROM,H.C.$
           A SYNTAX-DIRECTED TRANSLATING SYSTEM$
           TECHNICAL REPORT, IOWA UNIV. AUGUST NO.01 (1969)$

    10     VYSSOTSKY,V.A.$
           COMMON SENSE IN DESIGNING TESTABLE SOFTWARE$
           TECHNICAL REPORT, BELL LABORATORIES (1972)$

    11     SAYRE,D.$
           IS AUTOMATIC "FOLDING" OF PROGRAMS EFFICIENT ENOUGH TO DISPLACE MANUAL?$
           COMM. ACM, 12 NO.12(1969)P.656$

    12     GAUTSCHI,W.:KLEIN,B.J.$
           RECURSIVE COMPUTATION OF CERTAIN DERIVATIVES-A STUDY OF ERROR PROPAGATION$
           COMM. ACM, 13 NO.1(1970)P.7$

    13     CHENEY,C.J.$
           A NONRECURSIVE LIST COMPACTING ALGORITHM$
           COMM. ACM 13 NO.11(1970)P.677$

    14     FENG,T.Y.$
           INFORMATION SYSTEMS SEARCH ALGORITHMS FOR ASSOCIATIVE MEMORIES$
           DEPT OF ELECTRICAL ENGINEERING, PRINCETON, UNIV MARCH (1970)$

    15     MILLER,W.$
           ALGORITHMS COMPUTING ZEROS OF C$
           DEPT OF ELECTRICAL ENGINEERING, PRINCETON, UNIV MARCH (1970)$

    16     BENNETT,W.S.$
           SWITCHING THEORY ON OBTAINING BOOLEAN FUNCTIONS FOR APPROXIMATING BASE TWO ALGORITHMS$
           DEPT OF ELECTRICAL ENGINEERING, PRINCETON, UNIV MARCH (1970)$
    18     HSIAO,M.Y.:SELLERS,F.F.:CHIA,D.K.$
           SWITCHING THEORY BOOLEAN DIFFERENCE FOR TEST PATTERN GENERATION$
           DEPT OF ELECTRICAL ENGINEERING, PRINCETON, UNIV MARCH (1970)$

    20     MCLANE,P.J.$
           STOCHASTIC CONTROL AND ESTIMATION A LINEAR OPTIMAL ESTIMATION-CONTROL ALGORITHM FOR LINEAR SYSTEMS WITH
                STATEDEPENDENT DISTURBANCES$
           DEPT OF ELECTRICAL ENGINEERING, PRINCETON, UNIV MARCH (1970)$

    22     FENICHEL,R.R.$
           A NEW LIST-TRACING ALGORITHM$
           TM-19 OCTOBER(1970) AD-714-522$

THE ARTICLE FILE CONTAINS    18 ARTICLES.
```

TITLES IN ARTICLE FILE PERMUTED BY KEYWORD

REFERENCE NUMBER	TITLE----> THE END OF THE TITLE IS DENOTED BY A $.
3	ACCESS TO GOVERNMENT INFORMATION$THE RIGHT OF EQUAL
16	APPROXIMATING BASE TWO ALGORITHMS$SWITCHING THEORY ON OBTAINING BOOLEAN FUNCTIONS FOR
14	ASSOCIATIVE MEMORIES$INFORMATION SYSTEMS SEARCH ALGORITHMS FOR
16	BASE TWO ALGORITHMS$SWITCHING THEORY ON OBTAINING BOOLEAN FUNCTIONS FOR APPROXIMATING
16	BOOLEAN FUNCTIONS FOR APPROXIMATING BASE TWO ALGORITHMS$SWITCHING THEORY ON OBTAINING
18	BOOLEAN DIFFERENCE FOR TEST PATTERN GENERATION$SWITCHING THEORY
15	C$ALGORITHMS COMPUTING ZEROS OF
8	CALCULATOR$POCKET
10	COMMON SENSE IN DESIGNING TESTABLE SOFTWARE$
13	COMPACTING ALGORITHM$A NONRECURSIVE LIST
6	COMPLEX BEHAVIOR STRATEGY SOLUTIONS$ON BEHAVIOR STRATEGY SOLUTIONS IN TWO-PERSON ZERO-SUM FINITE EXTENDED GAMES WITH IMPERFECT INFORMATION. PART I: A METHOD FOR DETERMINATION OF MINIMALLY
1	CONSECUTIVE INTEGERS$PRIME FACTORS OF
12	DERIVATIVES-A STUDY OF ERROR PROPAGATION$RECURSIVE COMPUTATION OF CERTAIN
18	DIFFERENCE FOR TEST PATTERN GENERATION$SWITCHING THEORY BOOLEAN
9	DIRECTED TRANSLATING SYSTEM$A SYNTAX-
11	DISPLACE MANUAL?$IS AUTOMATIC "FOLDING" OF PROGRAMS EFFICIENT ENOUGH TO
20	DISTURBANCES$STOCHASTIC CONTROL AND ESTIMATION A LINEAR OPTIMAL ESTIMATION-CONTROL ALGORITHM FOR LINEAR SYSTEMS WITH STATEDEPENDENT
11	EFFICIENT ENOUGH TO DISPLACE MANUAL?$IS AUTOMATIC "FOLDING" OF PROGRAMS
3	EQUAL ACCESS TO GOVERNMENT INFORMATION$THE RIGHT OF
12	ERROR PROPAGATION$RECURSIVE COMPUTATION OF CERTAIN DERIVATIVES-A STUDY OF
20	ESTIMATION A LINEAR OPTIMAL ESTIMATION-CONTROL ALGORITHM FOR LINEAR SYSTEMS WITH STATEDEPENDENT DISTURBANCES$STOCHASTIC CONTROL AND
4	EVOLUTION OF AN OPERATING SYSTEM$THE
6	EXTENDED GAMES WITH IMPERFECT INFORMATION. PART I: A METHOD FOR DETERMINATION OF MINIMALLY COMPLEX BEHAVIOR STRATEGY SOLUTIONS$ON BEHAVIOR STRATEGY SOLUTIONS IN TWO-PERSON ZERO-SUM FINITE
1	FACTORS OF CONSECUTIVE INTEGERS$PRIME
6	FINITE EXTENDED GAMES WITH IMPERFECT INFORMATION. PART I: A METHOD FOR DETERMINATION OF MINIMALLY COMPLEX BEHAVIOR STRATEGY SOLUTIONS$ON BEHAVIOR STRATEGY SOLUTIONS IN TWO-PERSON ZERO-SUM
11	FOLDING" OF PROGRAMS EFFICIENT ENOUGH TO DISPLACE MANUAL?$IS AUTOMATIC "
16	FUNCTIONS FOR APPROXIMATING BASE TWO ALGORITHMS$SWITCHING THEORY ON OBTAINING BOOLEAN
6	GAMES WITH IMPERFECT INFORMATION. PART I: A METHOD FOR DETERMINATION OF MINIMALLY COMPLEX BEHAVIOR STRATEGY SOLUTIONS$ON BEHAVIOR STRATEGY SOLUTIONS IN TWO-PERSON ZERO-SUM FINITE EXTENDED
18	GENERATION$SWITCHING THEORY BOOLEAN DIFFERENCE FOR TEST PATTERN
3	GOVERNMENT INFORMATION$THE RIGHT OF EQUAL ACCESS TO
6	IMPERFECT INFORMATION. PART I: A METHOD FOR DETERMINATION OF MINIMALLY COMPLEX BEHAVIOR STRATEGY SOLUTIONS$ON BEHAVIOR STRATEGY SOLUTIONS IN TWO-PERSON ZERO-SUM FINITE EXTENDED GAMES WITH
1	INTEGERS$PRIME FACTORS OF CONSECUTIVE
20	LINEAR OPTIMAL ESTIMATION-CONTROL ALGORITHM FOR LINEAR SYSTEMS WITH STATEDEPENDENT DISTURBANCES$STOCHASTIC CONTROL AND ESTIMATION A
11	MANUAL?$IS AUTOMATIC "FOLDING" OF PROGRAMS EFFICIENT ENOUGH TO DISPLACE
14	MEMORIES$INFORMATION SYSTEMS SEARCH ALGORITHMS FOR ASSOCIATIVE
6	METHOD FOR DETERMINATION OF MINIMALLY COMPLEX BEHAVIOR STRATEGY SOLUTIONS$ON BEHAVIOR STRATEGY SOLUTIONS IN TWO-PERSON ZERO-SUM FINITE EXTENDED GAMES WITH IMPERFECT INFORMATION. PART I: A
6	MINIMALLY COMPLEX BEHAVIOR STRATEGY SOLUTIONS$ON BEHAVIOR STRATEGY SOLUTIONS IN TWO-PERSON ZERO-SUM FINITE EXTENDED GAMES WITH IMPERFECT INFORMATION. PART I: A METHOD FOR DETERMINATION OF
2	MODELLING TECHNIQUE IN PROGRAMMING$A
15	NONRECURSIVE LIST COMPACTING ALGORITHM$A
16	OBTAINING BOOLEAN FUNCTIONS FOR APPROXIMATING BASE TWO ALGORITHMS$SWITCHING THEORY ON
4	OPERATING SYSTEM$THE EVOLUTION OF AN
20	OPTIMAL ESTIMATION-CONTROL ALGORITHM FOR LINEAR SYSTEMS WITH STATEDEPENDENT DISTURBANCES$STOCHASTIC CONTROL AND ESTIMATION A LINEAR
6	PART I: A METHOD FOR DETERMINATION OF MINIMALLY COMPLEX BEHAVIOR STRATEGY SOLUTIONS$ON BEHAVIOR STRATEGY SOLUTIONS IN TWO-PERSON ZERO-SUM FINITE EXTENDED GAMES WITH IMPERFECT INFORMATION.
18	PATTERN GENERATION$SWITCHING THEORY BOOLEAN DIFFERENCE FOR TEST
8	POCKET CALCULATOR$
1	PRIME FACTORS OF CONSECUTIVE INTEGERS$
12	PROPAGATION$RECURSIVE COMPUTATION OF CERTAIN DERIVATIVES-A STUDY OF ERROR
12	RECURSIVE COMPUTATION OF CERTAIN DERIVATIVES-A STUDY OF ERROR PROPAGATION$
3	RIGHT OF EQUAL ACCESS TO GOVERNMENT INFORMATION$THE
0	SCHEDULING FOR A SHARE OF THE MACHINE$
14	SEARCH ALGORITHMS FOR ASSOCIATIVE MEMORIES$INFORMATION SYSTEMS
10	SENSE IN DESIGNING TESTABLE SOFTWARE$COMMON
0	SHARE OF THE MACHINE$SCHEDULING FOR A
10	SOFTWARE$COMMON SENSE IN DESIGNING TESTABLE
6	SOLUTIONS IN TWO-PERSON ZERO-SUM FINITE EXTENDED GAMES WITH IMPERFECT INFORMATION. PART I: A METHOD FOR DETERMINATION OF MINIMALLY COMPLEX BEHAVIOR STRATEGY SOLUTIONS$ON BEHAVIOR STRATEGY
20	STATEDEPENDENT DISTURBANCES$STOCHASTIC CONTROL AND ESTIMATION A LINEAR OPTIMAL ESTIMATION-CONTROL ALGORITHM FOR LINEAR SYSTEMS WITH
20	STOCHASTIC CONTROL AND ESTIMATION A LINEAR OPTIMAL ESTIMATION-CONTROL ALGORITHM FOR LINEAR SYSTEMS WITH STATEDEPENDENT DISTURBANCES$
6	STRATEGY SOLUTIONS IN TWO-PERSON ZERO-SUM FINITE EXTENDED GAMES WITH IMPERFECT INFORMATION. PART I: A METHOD FOR DETERMINATION OF MINIMALLY COMPLEX BEHAVIOR STRATEGY SOLUTIONS$ON BEHAVIOR
12	STUDY OF ERROR PROPAGATION$RECURSIVE COMPUTATION OF CERTAIN DERIVATIVES-A
6	SUM FINITE EXTENDED GAMES WITH IMPERFECT INFORMATION. PART I: A METHOD FOR DETERMINATION OF MINIMALLY COMPLEX BEHAVIOR STRATEGY SOLUTIONS$ON BEHAVIOR STRATEGY SOLUTIONS IN TWO-PERSON ZERO-
16	SWITCHING THEORY ON OBTAINING BOOLEAN FUNCTIONS FOR APPROXIMATING BASE TWO ALGORITHMS$
18	SWITCHING THEORY BOOLEAN DIFFERENCE FOR TEST PATTERN GENERATION$
9	SYNTAX-DIRECTED TRANSLATING SYSTEM$A
2	TECHNIQUE IN PROGRAMMING$A MODELLING
18	TEST PATTERN GENERATION$SWITCHING THEORY BOOLEAN DIFFERENCE FOR
10	TESTABLE SOFTWARE$COMMON SENSE IN DESIGNING
22	TRACING ALGORITHM$A NEW LIST-
9	TRANSLATING SYSTEM$A SYNTAX-DIRECTED
6	ZERO-SUM FINITE EXTENDED GAMES WITH IMPERFECT INFORMATION. PART I: A METHOD FOR DETERMINATION OF MINIMALLY COMPLEX BEHAVIOR STRATEGY SOLUTIONS$ON BEHAVIOR STRATEGY SOLUTIONS IN TWO-PERSON
15	ZEROS OF C$ALGORITHMS COMPUTING

THIS LISTING CONTAINS 70 PERMUTED TITLES.

FREQUENCY OF KEYWORD OCCURRENCES

THE LISTING IS ALPHABETICAL WHEN READ FROM LEFT TO RIGHT ACROSS EACH ROW.

FREQUENCY	KEYWORD	FREQUENCY	KEYWORD	FREQUENCY	KEYWORD	FREQUENCY	KEYWORD
1	ACCESS	1	APPROXIMATING	1	ASSOCIATIVE	1	BASE
2	BOOLEAN	1	C	1	CALCULATOR	1	COMMON
1	COMPACTING	1	COMPLEX	1	CONSECUTIVE	1	DERIVATIVES
1	DIFFERENCE	1	DIRECTED	1	DISPLACE	1	DISTURBANCES
1	EFFICIENT	1	EQUAL	1	ERROR	1	ESTIMATION
1	EVOLUTION	1	EXTENDED	1	FACTORS	1	FINITE
1	FOLDING	1	FUNCTIONS	1	GAMES	1	GENERATION
1	GOVERNMENT	1	IMPERFECT	1	INTEGERS	1	LINEAR
1	MANUAL	1	MEMORIES	1	METHOD	1	MINIMALLY
1	MODELLING	1	NONRECURSIVE	1	OBTAINING	1	OPERATING
1	OPTIMAL	1	PART	1	PATTERN	1	POCKET
1	PRIME	1	PROPAGATION	1	RECURSIVE	1	RIGHT
1	SCHEDULING	1	SEARCH	1	SENSE	1	SHARE
1	SOFTWARE	1	SOLUTIONS	1	STATEDPPENDENT	1	STOCHASTIC
1	STRATEGY	1	STUDY	1	SUM	2	SWITCHING
1	SYNTAX	1	TECHNIQUE	1	TEST	1	TESTABLE
1	TRACING	1	TRANSLATING	1	ZERO	1	ZEROS

THE KEY DIRECTORY CONTAINS 70 ENTRIES, INCLUDING 68 UNIQUE KEYWORDS.

ARTICLES SATISFYING THE FOLLOWING SEARCH REQUEST:

*** SOFTWARE ***

REFERENCE NUMBER AUTHOR---> EACH ATTRIBUTE IS TERMINATED BY A $.
TITLE----> IF ANY ATTRIBUTE CANNOT BE PRINTED ON A SINGLE LINE, IT IS CONTINUED ON THE NEXT LINE
JOURNAL--> AFTER AN INDENTATION OF FIVE SPACES.

10 VYSSOTSKY,V.A.$
COMMON SENSE IN DESIGNING TESTABLE SOFTWARE$
TECHNICAL REPORT, BELL LABORATORIES (1972)$

THERE ARE 1 ARTICLES SATISFYING THIS REQUEST.

ARTICLES SATISFYING THE FOLLOWING SEARCH REQUEST:

*** PATTERN AND GENERATION ***

REFERENCE NUMBER AUTHOR---> EACH ATTRIBUTE IS TERMINATED BY A $.
TITLE----> IF ANY ATTRIBUTE CANNOT BE PRINTED ON A SINGLE LINE, IT IS CONTINUED ON THE NEXT LINE
JOURNAL--> AFTER AN INDENTATION OF FIVE SPACES.

18 HSIAO,M.Y.:SELLERS,F.F.:CHIA,D.K.$
SWITCHING THEORY BOOLEAN DIFFERENCE FOR TEST PATTERN GENERATIONS$
DEPT OF ELECTRICAL ENGINEERING, PRINCETON, UNIV MARCH (1970)$

THERE ARE 1 ARTICLES SATISFYING THIS REQUEST.

ARTICLES SATISFYING THE FOLLOWING SEARCH REQUEST:

*** SWITCHING OR MODELLING ***

REFERENCE NUMBER AUTHOR---> EACH ATTRIBUTE IS TERMINATED BY A $.
TITLE----> IF ANY ATTRIBUTE CANNOT BE PRINTED ON A SINGLE LINE, IT IS CONTINUED ON THE NEXT LINE
JOURNAL--> AFTER AN INDENTATION OF FIVE SPACES.

2 DE LUCENA,C.J.P.*DE ALMEIDA CUNHA,L.F.$
A MODELLING TECHNIQUE IN PROGRAMMING$
PUC, CENTRO TECNICO CIENTIFICO SEPTEMBER NO.9/71(1971)$

16 BENNETT,W.S.$
SWITCHING THEORY ON OBTAINING BOOLEAN FUNCTIONS FOR APPROXIMATING BASE TWO ALGORITHMS$
DEPT OF ELECTRICAL ENGINEERING, PRINCETON, UNIV MARCH (1970)$

18 HSIAO,M.Y.:SELLERS,F.F.:CHIA,D.K.$
SWITCHING THEORY BOOLEAN DIFFERENCE FOR TEST PATTERN GENERATIONS$
DEPT OF ELECTRICAL ENGINEERING, PRINCETON, UNIV MARCH (1970)$

THERE ARE 3 ARTICLES SATISFYING THIS REQUEST.

ARTICLES SATISFYING THE FOLLOWING SEARCH REQUEST:

*** BOOLEAN AND APPROXIMATING OR SCHEDULING ***

REFERENCE NUMBER AUTHOR---> EACH ATTRIBUTE IS TERMINATED BY A $.
TITLE----> IF ANY ATTRIBUTE CANNOT BE PRINTED ON A SINGLE LINE, IT IS CONTINUED ON THE NEXT LINE
JOURNAL--> AFTER AN INDENTATION OF FIVE SPACES.

0 LARMOUTH,J.$
SCHEDULING FOR A SHARE OF THE MACHINE$
SOFTWARE 5 NO.1(1975)P.29$

16 BENNETT,W.S.$
SWITCHING THEORY ON OBTAINING BOOLEAN FUNCTIONS FOR APPROXIMATING BASE TWO ALGORITHMS$
DEPT OF ELECTRICAL ENGINEERING, PRINCETON, UNIV MARCH (1970)$

THERE ARE 2 ARTICLES SATISFYING THIS REQUEST.

APPENDIX D

SAMPLE JCL LISTINGS

CREATION OF NONKEYWORD FILE

```
//STEP1     EXEC PGM=SORT
//SYSOUT    DD SYSOUT=A
//SORTLIB   DD DSN=SYS1.SORTLIB,DISP=SHR
//SORTWK01  DD UNIT=SYSDA,SPACE=(TRK,(30),,CONTIG)
//SORTWK02  DD UNIT=SYSDA,SPACE=(TRK,(30),,CONTIG)
//SORTWK03  DD UNIT=SYSDA,SPACE=(TRK,(30),,CONTIG)
//SORTIN    DD *
/*
//SORTOUT   DD DSN=COMSC.SEQ.CRCTZER.NONKYWD,UNIT=2314,
//          VOL=(PRIVATE,SER=DISK28),SPACE=(TRK,(2,2)),
//          DCB=(RECFM=FB,BLKSIZE=1800,LRECL=18),DISP=(OLD,PASS)
//SYSIN     DD *
 SORT FIELDS=(1,18,CH,A)
 END
/*
//STEP2     EXEC PGM=IEBGENER
//SYSIN     DD DUMMY
//SYSUT1    DD DSN=*.STEP1.SORTOUT,DISP=(OLD,KEEP)
//SYSUT2    DD SYSOUT=A,DCB=(RECFM=F,BLKSIZE=18,LRECL=18)
//SYSPRINT  DD SYSOUT=A
//
```

CREATE PROGRAM

```
//CREATE   EXEC PGM=CREATE,REGION=126K
//STEPLIB  DD DSN=COMSC.PROG.CROTZER,UNIT=2314,
//         VOL=(PRIVATE,SER=DISK28),DISP=SHR
//SYSPRINT DD SYSOUT=A
//SYSIN    DD DUMMY,DCB=BLKSIZE=80
//PARM     DD *
      303        20        18        134       200
/*
//COUNT    DD DSN=COMSC.SEQ.CROTZER.COUNT.SAMPLE,UNIT=2314,
//         VOL=(PRIVATE,SER=DISK28),SPACE=(TRK,1),
//         DCB=(RECFM=F,BLKSIZE=32,LRECL=32),DISP=(NEW,KEEP)
//KEY      DD DSN=COMSC.REG.CROTZER.KEY.SAMPLE,UNIT=2314,
//         VOL=(PRIVATE,SER=DISK28),SPACE=(TRK,(10,5)),
//         DCB=(RECFM=F,BLKSIZE=7280,LRECL=7280),DISP=(NEW,KEEP)
//ARTICLE  DD DSN=COMSC.REG.CROTZER.ARTICLE.SAMPLE,UNIT=2314,
//         VOL=(PRIVATE,SER=DISK28),SPACE=(TRK,(10,10)),
//         DCB=(RECFM=F,BLKSIZE=139,LRECL=139),DISP=(NEW,KEEP)
//
```

EDIT PROGRAM

```
//EDIT      EXEC PGM=EDIT,REGION=126K
//STEPLIB   DD DSN=COMSC.PROG.CROTZER,UNIT=2314,
//          VOL=(PRIVATE,SER=DISK28),DISP=SHR
//SYSPRINT  DD SYSOUT=A
//SYSIN     DD DUMMY,DCB=BLKSIZE=80
//ARTCRDS   DD *
/*
//ERRCRDS   DD *
/*
//CKARTS    DD DSN=COMSC.SEQ.CRCTZER.SAMPEDTD,UNIT=TAPE,
//          VOL=SER=T9092,LABEL=(4,SL),DISP=(NEW,KEEP),
//          DCB=(RECFM=VB,BLKSIZE=2000,LRECL=750)
//ERRARTS   DD SYSOUT=B,DCB=BLKSIZE=80
//PARM      DD *
        YES
/*
//ERRMSG    DD SYSOUT=A
//
```

UPDATE PROGRAM

```
//UPDATE   EXEC PGM=UPDATE,REGION=198K
//STEPLIB  DD DSN=COMSC.PROG.CROTZER,UNIT=2314,
//         VOL=(PRIVATE,SER=DISK28),DISP=SHR
//SYSPRINT DD SYSOUT=A
//SYSIN    DD DUMMY,DCB=BLKSIZE=80
//CCUNT    DD DSN=COMSC.SEQ.CROTZER.CCUNT.SAMPLE,UNIT=2314,
//         VOL=(PRIVATE,SER=DISK28),SPACE=(TRK,1),
//         DCB=(RECFM=F,BLKSIZE=32,LRECL=32),DISP=(OLD,KEEP)
//NCNKEY   DD DSN=COMSC.SEQ.CROTZER.NCNKYWD,UNIT=2314,
//         VOL=(PRIVATE,SER=DISK28),SPACE=(TRK,(2,2)),
//         DCB=(RECFM=FB,BLKSIZE=1800,LRECL=18),DISP=(OLD,KEEP)
//KEYWDDL  DD *
/*
//ARTDL    DD *
/*
//ARTIN    DD DSN=COMSC.SEQ.CROTZER.SAMPEDTD,UNIT=TAPE,
//         VOL=SER=T9092,LABEL=(4,SL),DISP=(OLD,KEEP),
//         DCB=(RECFM=VB,BLKSIZE=2000,LRECL=750)
//KEYWDIN  DD *
/*
//KEY      DD DSN=COMSC.REG.CROTZER.KEY.SAMPLE,UNIT=2314,
//         VOL=(PRIVATE,SER=DISK28),SPACE=(TRK,(10,5)),
//         DCB=(RECFM=F,BLKSIZE=7280,LRECL=7280),DISP=(OLD,KEEP)
//ARTICLE  DD DSN=COMSC.REG.CROTZER.ARTICLE.SAMPLE,UNIT=2314,
//         VOL=(PRIVATE,SER=DISK28),SPACE=(TRK,(10,10)),
//         DCB=(RECFM=F,BLKSIZE=139,LRECL=139),DISP=(OLD,KEEP)
//MESSAGE  DD SYSOUT=A
//
```

REPORT PROGRAM

```
//REPORT   EXEC PGM=REPORT,REGION=126K
//STEPLIB  DD DSN=COMSC.PROG.CROTZER,UNIT=2314,
//         VOL=(PRIVATE,SER=DISK28),DISP=SHR
//SYSPRINT DD SYSOUT=A
//SYSIN    DD DUMMY,DCB=BLKSIZE=80
//COUNT    DD DSN=COMSC.SEQ.CROTZER.COUNT.SAMPLE,UNIT=2314,
//         VOL=(PRIVATE,SER=DISK28),SPACE=(TRK,1),
//         DCB=(RECFM=F,BLKSIZE=32,LRECL=32),DISP=(OLD,KEEP)
//KEY      DD DSN=COMSC.REG.CROTZER.KEY.SAMPLE,UNIT=2314,
//         VOL=(PRIVATE,SER=DISK28),SPACE=(TRK,(10,5)),
//         DCB=(RECFM=F,BLKSIZE=7280,LRECL=7280),DISP=(OLD,KEEP)
//ARTICLE  DD DSN=COMSC.REG.CROTZER.ARTICLE.SAMPLE,UNIT=2314,
//         VOL=(PRIVATE,SER=DISK28),SPACE=(TRK,(10,10)),
//         DCB=(RECFM=F,BLKSIZE=139,LRECL=139),DISP=(OLD,KEEP)
//LISTING  DD SYSOUT=A,DCB=BLKSIZE=133
//REQUEST  DD *
      YES        YES        YES
SOFTWARE $
   PATTERN AND GENERATION$
                    SWITCHING          OR
 MODELLING   $
BOOLEAN AND APPROXIMATING OF SCHEDULING $
/*
//
```

VITA

Arthur Douglas Crotzer

Candidate for the Degree of

Master of Science

Thesis: EFFICACY OF B-TREES IN AN INFORMATION STORAGE AND RETRIEVAL ENVIRONMENT

Major Field: Computing and Information Sciences

Biographical:

Personal Data: Born in Clarksville, Tennessee, February 25, 1951, the son of Mr. and Mrs. A. E. Crotzer.

Education: Graduated from Clarksville High School, Clarksville, Tennessee, in June, 1969; received Bachelor of Science degree from Austin Peay State University, Clarksville, Tennessee, in June, 1973; with majors in Mathematics and Physics; completed requirements for the Master of Science degree at Oklahoma State University, Stillwater, Oklahoma, in July, 1975.

Professional Experience: Application Programmer, Austin Peay State University Computer Center, Clarksville, Tennessee, Summer 1971, 72, 73; Graduate Assistant, Oklahoma State University, Computing and Information Sciences Department, Stillwater, Oklahoma, August, 1973, to May, 1975.