A STUDY OF TWO COMPETING INDEX MECHANISMS: PREFIX B^+-TREE AND TRIE STRUCTURES

BY

AN-LEE ANNE FENG

Bachelor of Science in Agriculture

National Taiwan University

Taipei, Taiwan

Republic of China

1972

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the Degree of
MASTER OF SCIENCE
December, 1982

Thesis
1982
F332s
Cop. 2

A STUDY OF TWO COMPETING INDEX
MECHANISMS: PREFIX B^+-TREE
AND TRIE STRUCTURES

Thesis Approved:

James R. Van Doren
Thesis Adviser

Donald D. Fisher

Sharilyn A. Thoreson

Norman N. Durham
Dean of Graduate College

1143210

PREFACE

This thesis deals with two competing index mechanisms, namely, prefix B^+-trees and trie structures, which are useful for handling varying size keys in document retrieval systems. Refinements and variants of these two indexing methods are studied. Tradeoffs of storage requirements and retrieval time or performance benefits and maintainance difficulties for various refining approaches are examined.

I would like to express my sincere appreciation and gratitude to Dr. James R. Van Doren, my major professor, for his patience, guidance, encouragement and understanding throughout my graduate study. Thankes are also extended to Dr. D. D. Fisher and Dr. S. A. Thoreson for serving on my graduate committee.

I extend a very special and sincere thanks to Dr. D. D. Fisher and Dr. J. R. Phillips for unfailing confidence and support which they have given me over the last three years.

Finally, a debt of gratitude which can never be adequately expressed is due my parants, Mr. and Mrs. Ko-Chun Feng and their favorite daughter, An-Chun, for their love, understanding, and sacrifices throughout my studies in the States. For this and their unfailing support I am forever indebted.

TABLE OF CONTENTS

Chapter Page

I. INTRODUCTION 1

II. BRIEF DESCRIPTION OF ON-LINE DOCUMENT RETRIEVAL SYSTEMS 3

Introduction 3
How It Works 4
Inverted File Techniques 6
General Operations 10
Index Decoding 11

III. PREFIX B^+-TREE INDEXING 14

Basic B-Trees 15
Organization of B-Trees 15
Advantages of B-Tree Based Indexing . 17
B^+-Trees 19
Motivation of B^+-Trees 19
Characteristics of B^+-Trees 20
Prefix B^+-Trees 22
Simple Prefix B^+-Trees 26
Prefix B^+-Trees 28
Search, Insertion, and Deletion . . . 33
Evaluation of Prefix B^+-Tree Indexing . . 41

IV. TRIE STRUCTURE INDEXING 44

Digital Search Trees 45
Basic Trie Structure 48
Refinements and Variants of Tries 55
Pruning a Trie 56
Reordering Attribute Testing 57
Entering Multiple Keys 62
O-Tries (Order-Containing Tries) . . 62
Linked List Implementation 66
C-Tries (Compressed Tries) 68
Evaluation of Trie Index 71

V. SUMMARY, CONCLUSION AND SUGGESTION FOR FURTHER RESEARCH 75

Summary of Prefix B^+-Tree Indexing 75
Summary of Trie Indexing 78

Chapter | Page

Conclusion and Suggested Further Work . . . 81

BIBLIOGRAPHY 83

LIST OF TABLES

Table — Page

I. Common Prefix in Simple Prefix B^+-tree of Figure 8 31

II. Knuth's MIX Character Code and Bit String Representation for each Alphabetic Character 47

LIST OF FIGURES

Figure		Page
1.	Inverted File Structure with Separate Accession List	8
2.	Inverted File Structure with Accession Lists Included in the Index	9
3.	Existing Index Decoding Techniques	12
4.	Page Organization of B-Trees	17
5.	A B-Tree with Index and Sequence Set	21
6.	(a) A Leaf Node of a B^+-Tree (b) Results of Inserting the Key 'SEPARATOR' into the Leaf Node of (a)	24
7.	Two Possible Node Organizations for Prefix B^+-Trees	26
8.	A Simple Prefix B^+-Tree	27
9.	Partial Index Structure of a Simple Prefix B^+-Tree	30
10.	Prefix B^+-Tree Derived from the Simple Prefix B^+-Tree of Figure 8	32
11.	Result of Inserting the Key 'CONSTRUCT' into the Prefix B^+-Tree of Figure 11	36
12.	Result of Deleting the Key 'CONTROL' from the Prefix B^+-Tree of Figure 11, Merging Scheme Is Used	39
13.	Result of Deleting the Key 'CONTROL' from the Prefix B^+-Tree of Figure 11, Redistribution Scheme Is Used	40
14.	A Digital Search Tree for 10 Common Programming Languages and Software Packages, Inserted in Increasing Lexical Order	47

Figure | Page

15. Trie Constructed for Keys of Figure 15, Sampling One Character at a Time, Left to Right 50

16. Result of Inserting Keys 'PLANS' and 'APPLY' into the Trie of Figure 16 53

17. Result of Deleting Keys 'FORTRAN' and 'PLI' from the Trie of Figure 16 54

18. (a) A Full Trie, and (b) A Pruned Trie for Keys of Figure 16, Sampling One Character at a Time, Left to Right 58

19. (a) A Leaf Chain and (b) an Internal Chain . . . 59

20. Trie Constructed for Keys of Figure 15, Sampling One at a Time, Right to Left 61

21. Trie Obtained for Keys of Figure 16 When Number of Levels is Limited to 3, Key Has Been Sampled Left to Right, One at a Time 64

22. An O-Trie for the Set of Keys of Figure 8(# of Levels = 3, # of Branch Nodes = 5) 65

23. An Optimum Pruned Trie for the Set of Keys in Figure 8 (# of Levels = 5, # of Branch Node = 8) 66

24. The Linked List Implementation of the Trie Shown in Figure 21 67

25. Structure of (a) an Internal Branch Node and (b) a Leaf Node in a C-trie 70

CHAPTER I

INTRODUCTION

A major use of digital computers is to manage, correlate and retrieve large collections of data, either in the form of formatted text or text with minimal formatting. Retrieval of information from large data files stored on secondary storage, such as magnetic disk and drum cannot be performed efficiently or rapidly without an efficient method for external searching. Standard forms of information retrieval systems consist of master files, inverted files and an index of keywords. To retrieve an item, the index is searched for the keyword and the corresponding entry in the inverted file extracted, giving the address in the master file of all the records satisfying the request. The most time consuming part of this retrieval process is the search of the index and several methods have been devised to minimize this.

The intention of this thesis will center on two techniques for constructing an index: prefix B^+-tree and trie structure. Both of them are tree structured indices with keys of variable length and can be used in textual databases or document retrieval systems to speed up information retrieval.

Chapter II presents a brief description of on-line document retrieval systems. Index techniques, inverted file techniques, general operations and several index schemes used in document retrieval systems are all addressed.

Chapter III contains a discussion of the development of B-trees, B^+-trees, simple prefix B^+-trees and prefix B^+-trees. Motivation, refinements, and tradeoffs at each evolutionary step of B-tree development are illustrated by examples.

Chapter IV discusses a particular type of digital search tree which is called a trie structure. The primary concern about it in this thesis will be placed on illustrating how to minimize storage requirements. A primary difficulty with a trie structure is also discussed. The variants of tries, such as pruned tries, O-tries, linked list implementation of tries and C-tries, are examined by examples.

The final chapter summarizes what has been presented, illustrates the comparisons between prefix B^+-trees and trie structures and makes suggestions for further study and research.

CHAPTER II

BRIEF DESCRIPTION OF ON-LINE DOCUMENT RETRIEVAL SYSTEMS

Introduction

The great importance of the role played by document retrieval systems or textual databases is to achieve better access to all types of stored information from the different areas in science, so that people can make use of existing knowledge and information to solve various problems such as scientific, political, technical, economic and social problems.

Document retrieval systems or textual databases consist of a large collection of documents, with some scheme to delimit and access the individual documents within a database. By convenience, the term document will refer to the individual books, journal articles, court decision cases, etc. (26). Retrieval from textual databases may be based on contexts and retrieval keys consisting of arbitrarily chosen words or portions of words. Unlike formatted databases, which are concerned with fields and key values of known position and format, the contents of textual databases are order dependent and very little formatting is necessary. The order dependency here means that contents

retrieved from a textual database do not reside in some known positions within a record which one can specify, but are exactly in the order in which the contents are kept in a database. Many textual databases contain a large number of documents and grow fairly rapidly. For example, a textual database containing all court decisions would take around 25 billion characters, while large formatted databases may generally contain 10 to 100 million characters (13).

How It Works

In the past, many of the computer-based retrieval systems relied on manually assigned keywords or index terms for the identification of documents, even though a search operation, for the most part, is carried out automatically. The typical document retrieval system in the past can be considered in four parts:

1. A classification scheme is devised for the document collection.
2. Index terms are assigned to a document so that it can be entered into the classification.
3. A query is formulated using terms from the classification scheme.
4. A search is made to find documents relevant to the query (17).

A variety of classification schemes are used. For example, the Dewey or Universal Decimal System used in libraries assigns a text number to each document, with which the position of this document relative to others in a hierarchical system is shown. An alternative method is to

assign index terms or keywords which indicate the subject matter of the document. For instance, a particular article might have as index terms the phrases, 'Information retrieval system', 'File organization', and 'Search algorithm' (26).

Nevertheless, the idea that manual systems and procedures should be replaced by suitably chosen automatic methods has became more widespread since the 1960s, as the amount and complexity of the available information has continued to grow. However, cost and storage capacity for automatic full text analysis has been a serious limitation. Recent improvements in microelectronics and peripheral storage technology have eliminated many of the cost barriers to such approaches. Improvements in indexing organization schemes have also contributed to solving performance limitations. Today a growing number of textual databases dealing with bibliographic reference allow online access to a large body of information in a given field, e.g. computer assisted legal research (CALR) systems in law. Lexis, Westlaw and JURIS are the three major online CALR systems which provide free text accesses to the full text of source documents (1). In this approach, the full text of documents is stored in its original form and then is lexically analyzed on a word basis: significant or nontrivial words are selected to build an index to enable retrieval to stored documents, while predetermined noise or noncontent words (e.g., the, to, and, this, that etc.) are ignored. Each

significant word can serve as a key term from which all the documents containing it can be obtained by a search scheme, no subject indexing is used. This is called free text access which means the access to any word in the entire text, excluding a list of noise words. This full text/free text combination permits the searcher to look for almost any combination of words or phrases, any place in the text of document. Since a particular document can be searched by specifying a great number of significant words rather than a few index terms in the classification schemes, the search is more precise and also saves the manual work of full text searching. For instance, if the user asks for the co-occurrence of two terms in the same sentence or paragraph, it can be done by just merging the accession lists corresponding to the specified co-occurrence terms (see the next section), and then retrieving documents according to the accession numbers on the resulting accession list. However, a large amount of computer effort may be required to implement lexical analysis on full text during database preparation. General descriptions and evaluations of online full text document retrieval systems can be found in Appenzeller (1) and Benson (4).

Inverted File Techniques

After the complete set of allowable key terms is obtained, no matter whether they are index terms assigned to documents or significant words selected from documents, a

'roadmap' providing the search path to the documents is required to be constructed. A common approach to search such document collections rapidly is to use an inverted file technique. According to Knuth's (16) definition, an inverted file means that the roles of records and attributes are reversed. That is to say, instead of listing the attributes of a given record, the records having a given attribute are listed. Here, his definition can be extended as: from an inverted file, a list can be obtained which contains all the accession numbers of the documents in a database in which the given term is found.

Figure 1 and Figure 2 illustrate two possible schemes for implementing an inverted file structure. Although it is convenient to view this structure as having two parts, the inversion and document files, more details can be obtained if the inversion file is further divided into two logical parts, index and accession lists. The index consists of all the unique key terms, and all of them might reside in the bottom level with some of them being duplicated in the upper levels. In Figure 1, each unique key term in the bottom level is stored with a pointer to the corresponding accession list, while in Figure 2, each key term in the bottom level is immediately followed by the accession list corresponding to it. In the latter case, the organization of the bottom level is different from that of upper levels. For large databases, both the index and the accession lists are separate, or the accession lists are included in the

lowest level of the index, index as well as accession lists are stored in a secondary storage.

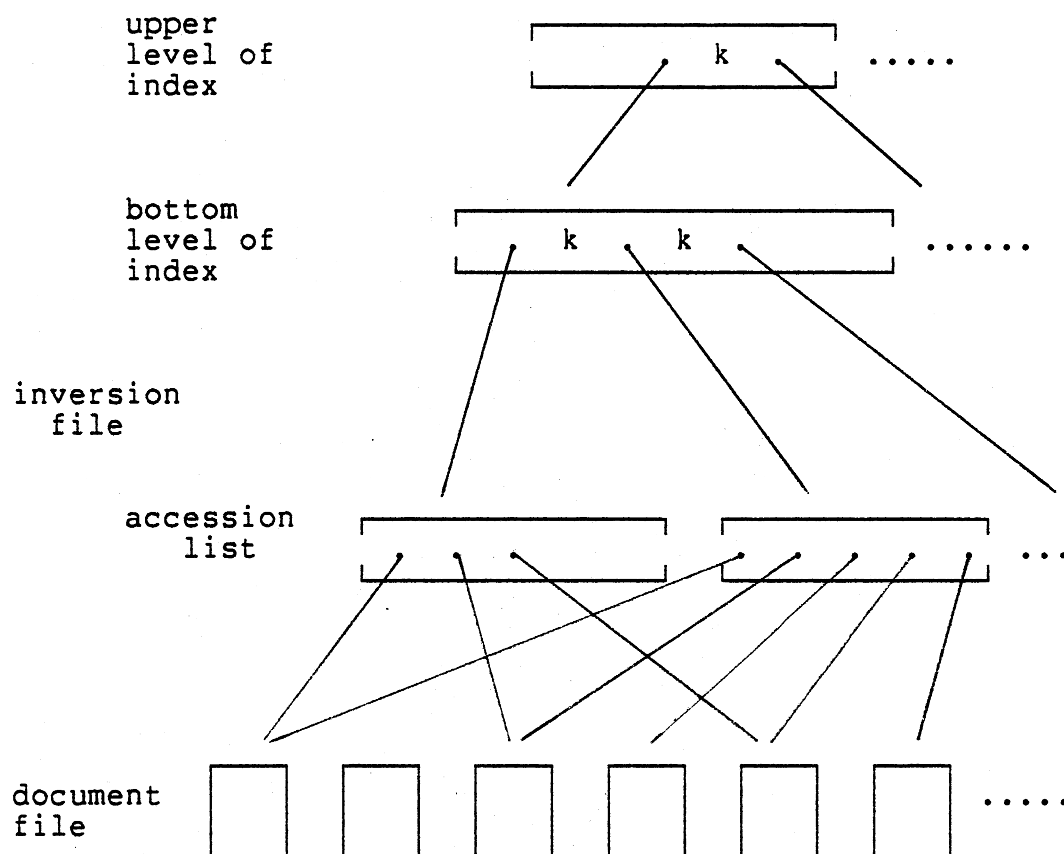


Figure 1. Inverted File Structure with Separate Accession Lists

In using an inverted file technique, a query is answered by locating the accession lists of the key terms in the query, followed by processing ('or-ing' and 'and-ing') these lists to determine the correct documents, and finally

by retrieving the documents. The principal advantages of this organization are that all query logic can be completed without accessing the database until the resulting subset of the database is formed, and then the qualified documents can be searched (13).

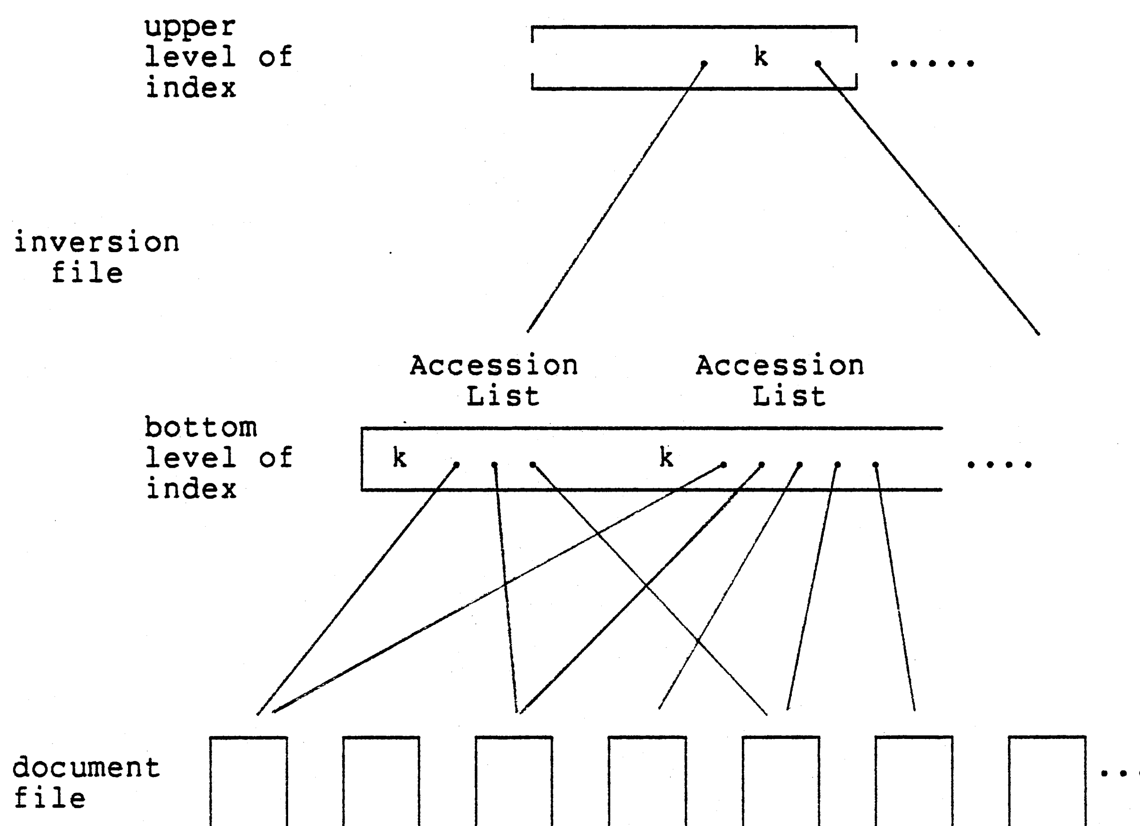


Figure 2. Inverted File Structure with Accession Lists Included in the Index

General Operations

Normal operations on an interactive document retrieval system consist of forming progressively smaller subsets of the database until the number of the documents is small enough to be examined by the user. This is done by the specification of search patterns consisting of co-occurrences, alternatives, and exclusions. Searches for co-occurrences locate two or more terms within a specified context, either unordered or ordered. The specification of an ordered co-occurrence can either require that the terms be contiguous or be separated within a specified number of words. Searches for alternatives locate contents which contain at least one of a group of given terms. Exclusion searches locate contexts which do not contain two selected terms simultaneously (13).

It is frequently desirable in on-line retrieval systems that there be some 'dialogues' which transmit the intermediate results from the system to the user, and based upon these results, the user can specify the action he wants the system to take via this dialogue facility. For example, the presearch statistics such as the number of documents required to be retrieved can be transmitted to the user after the index decoding process (see next section) is done. The user at the terminal then decides, based on the statistics, whether to proceed with the search in the document file, to modify the query, or terminate it. This facility can help to avoid wasteful and wrong retrieval.

Index Decoding

Basically, on-line document retrieval can be viewed as a two step process: step 1 involves index decoding which translates the query language key term into an address or series of addresses to every document in the document file that satisfies the key conditions. The information required to perform this decoding is called the key index. Step 2 consists of the random access search in the document file based upon the list addresses obtained from step 1 (17). The most time-consuming part of this retrieval process is the index decoding. The critical parameter is the number of accesses to the secondary storage. One procedure is to narrow the search down to a group (known as a block or bucket or page) of keys, which can be searched rapidly in primary memory. The size of these blocks is selected to be the same as the size of the unit of transfer between primary and secondary storage. The search time can be further reduced by selecting an appropriate index structure.

Figure 3 classifies the existing techniques that are used to perform index decoding. These divide into two general classes, one called key to address transformation or hashing, the other called tree or table look-up decoding (17). The first level distinguishes the hashing approach from the tree approach. In general hashing requires less search time than the tree approach. However, the range and distribution of the values of keys may effect the efficiency

of the hashing scheme to a great degree (6). This makes it difficult to use such a scheme in a general document retrieval system in which the properties of keys are not known in advance. The tree approach, on the other hand, has no such difficulty.

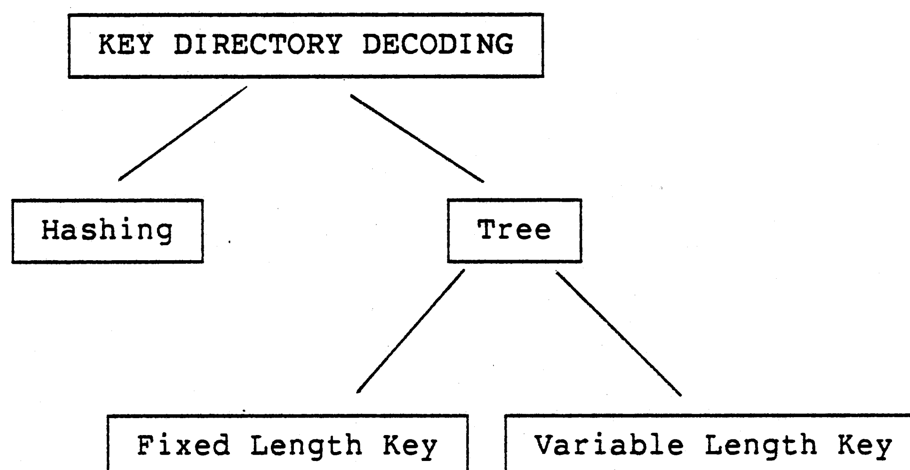


Figure 3. Existing Index Decoding Techniques

At the second level, the tree method branches into fixed versus variable length keys. The trade-off here is based entirely upon ambiguous decoding. Since in a general language a complete key is generally variable in length, if any transformation is made on this key that converts it to a fixed length, then some ambiguous decoding may be introduced. On the other hand, the tree with variable length keys is guaranteed not to produce an ambiguous

decoding, but the price is increased programming complexity.

The attention of this thesis is focused narrowly and specifically upon the data structure of the index using tree structure with variable length keys. Two techniques are to be examined, namely, prefix B^+-trees and trie structures.

CHAPTER III

PREFIX B^+-TREE INDEXING

External searching is critical to retrieve information from databases such as document retrieval systems appearing on secondary storage. The index which speeds retrieval by directing the search path to the document file is kept in the secondary storage as well as the document file itself because the set of all keys may not fit in primary memory. A tree organized index is efficient for external searching, if an appropriate way to represent the tree is chosen (16).

The starting section of this chapter presents a brief discussion of the basic B-tree as proposed by Bayer and McCreight (2), and illustrates why B-trees are considered the standard organization for indexes in a database system. Section 2 shows a superior variant of B-tree, the B^+-tree, which has an independent B^+-index and the order set of leaves, the sequence set or B^+-file (3). The remainder of this chapter is focused on the prefix B^+-tree, in which, the B^+-index in a B^+-tree is further improved by using key prefix compression and "shortest" separator keys in order to reduce the number of levels and the space requirements of the B^+-index. Simple prefix B^+-trees and prefix B^+-trees are illustrated. In section 3, the algorithms for

constructing and maintaining prefix B^+-trees are reviewed. The final section, section 4, contains an evaluation of prefix B^+-tree indexing.

Basic B-trees

With the fact that an index resides on discs or drums, searching it should be done by accessing secondary storage. The time required to access secondary storage is the main component of the total time required to retrieve information from databases (11). Minimizing the number of accesses to secondary storage is highly desirable.

A new approach to external searching by means of multi-way branching was proposed in 1970 by Bayer and McCreight (2). They called this new kind of data structure a B-tree. Based upon Bayer and McCreight's definition, the index consists of a number of entries which are triples (k(i), a(i), p(i)) of fixed size data items, namely a key k(i), some associated information a(i), and a pointer p(i). The key k(i) identifies an unique element in the index, the associated information field a(i) is typically a pointer to a record or a collection of records identified by k(i), and the pointer p(i) is a disc address at which the root of the subtree containing all the keys which satisfy the branching condition is located.

Organization of B-trees

The index is broken into pages of fixed size. A page

is a block of information transferred between primary memory and secondary storage, and also corresponds to a node in a B-tree index. Each page need only be partially filled. Figure 4 depicts the organization of a page (node) P with j keys, j associated information fields, j+1 pointers and some unused space. k(i), a(i) and p(i) represents key, associated information and pointer to the ith successor of P respectively. Within each page (node) P, the keys are sequential in increasing order, that is, k(i) < k(i+1) for 0 < i < j. p(0) is a pointer to a subtree which contains keys less than k(1) and p(j) is a pointer to a subtree which contains keys greater than k(j). Other pointers p(i), for 0 < i < j, point to subtrees which contain keys greater than k(i) but less than k(i+1). If the node P is a leaf node, then all pointers of it are undefined (2), or they should be eliminated (16). since a leaf node is a terminal node which carries no branching information in the indexing sense.

A B-tree of order m is a tree which has the node organization mentioned above and satisfies the following properties:

1. A B-tree is a balanced search tree in which each path from the root to any leaf has uniform depth.
2. Each node, except for the root, contains between FLOOR((m-1)/2) and (m-1) keys. This guarantees that storage utilization is at least 50 .
3. The root node contains between 1 and (m-1) keys.
4. All leaf nodes appear on the same level and have no successors..
5. Each nonleaf node with k keys has (k+1) successors (16).

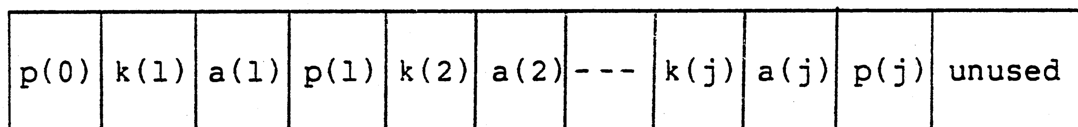


Figure 4. Page Organization of B-trees

Except for the root page which may be kept in internal memory during retrieval, pages of an index are usually kept in secondary storage and require an access to secondary stroage each time they are to be inspected. Once a page has been read into the internal memory, an internal search is required to locate the proper descendant pointer. Knuth(16) points out that a sequential search might be proper for small nodes, while a binary search might be useful if the node is large.

<u>Advantages of B-tree Based Indexing</u>

The superiority of B-trees over other index techniques is in the methods for inserting and deleting records. These methods always leave the tree balanced. This is done by restricting deletion and insertion at leaf nodes only. If the key to be deleted is in an upper level node, it is first swapped with its predecessor or successor, which always

appears on the leaf level. Therefore, nodes splitting off a sibling during insertion or two siblings being catenated into a single node during deletion are always initiated at leaves and propagate toward the root. In other words, B-trees are built from the bottom up. The only way in which the height of the tree can increase is that the root node splits and a new root must be introduced. The opposite process occurs if the tree contracts. The basic operations performed on B-tree based indexes such as searching, insertion and deletion will be examined when the prefix B^+-tree is discussed.

According to Bayer and McCreight (2, p.174), a B-tree based index offers significant advantages:

1. Storage utilization is at least 50% at any time and should be considerably better on the average.

2. Storage is requested and released as the file grows and contracts. There is no congestion problem or degradation of performance if the storage occupancy is very high.

3. Although the B-tree structure is originally designed to function as an index for dynamic random access files, the natural order of the keys in a B-tree is maintained and sequential processing based on that order is also allowed.

Besides, Knuth (16) points out that a B-tree based index makes it possible both to search and to update a large file with 'guaranteed' efficiency, in the worst case, using relatively simple algorithms. Comer (7) also states that there is no need for periodic 'reorganization' of the entire file if using a B-tree to index a file.

B^+-trees

As with most file organizations, variants of B-trees abound. Among them, B^+-trees are probably the most widely used variant of the original B-tree. VSAM, IBM's general purpose B-tree based organization and access method, is a well-known example of using a B^+-tree approach. The motivation, characteristics and use of B^+-trees are given in this section. It is intended that this section offers prerequisite background for the prefix B^+-tree.

Motivation of B^+-trees

The conventional B-tree is quite good for indexing a dynamic random access file, but a weakness of it is apparent in the case that sequential processing is required. A simple preorder tree traversal can be used to extract all the keys in order, while a significant amount of primary memory may be required to stack all the nodes along a path from the root to avoid reading these nodes twice. Additionally, processing a "find next" operation may require tracing a path through several nodes before reaching the desired key (7). Furthermore, in a conventional B-tree, associated information stored with the key may occupy a considerable portion of an index node, so that the order of the B-tree may be relatively small and the height of it may be relatively large. B^+-trees were designed to remove these weaknesses and provide a way which is suited to both a random and sequential processing environment.

<u>Characteristics of B⁺-trees</u>

The major deviations of B^+-trees from conventional B-trees are summarized by the following:

1. All keys of B^+-trees reside in leaves, Each upper level key is copied from a bottom level key during a node split on insertion.

2. Only the keys in the bottom level are associated with data records. In other words, an index entry in the upper level contains only key and pointer but no associated information at all.

3. Each leaf node of B^+-trees has a link field which points to the next leaf node to the right, except the link in the rightmost node which is null (7).

From the above, it is convenient to view a B^+-tree as having two independent parts as mentioned at the beginning of this chapter: the B^+-index and the sequence set which are depicted in Figure 5. The B^+-index that directs searches to the bottom level is organized exactly the same as a conventional B-tree. The sequence set is actually a linked list of all leaves in sequence order. Some implementations of a B^+-tree may have data stored with the keys in leaf nodes and others have accession lists or pointers to accession lists stored with the keys in leaves. Therefore, the structure of leaf nodes may differ from the structure of the upper level nodes.

A successful random search in a B^+-tree begins at the root as in a conventional B-tree but it is detected only when a matching key is found at the leaf level. Sequential processing begins at the leftmost leaf and is aided by following the horizontal links across the leaves. Other

requests such as 'find all records with key values between x and y' can be answered by locating the first qualified record in the bottom level and then processing sequentially the following records from that point until the key value exceeds y.

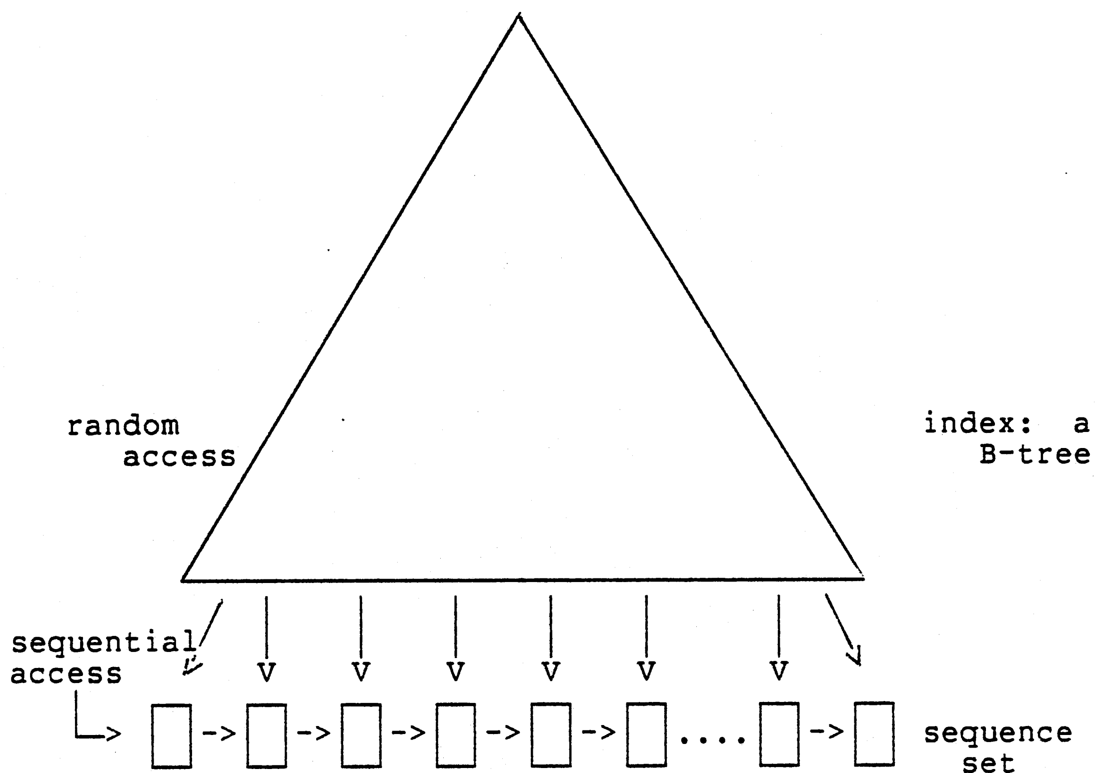


Figure 5. A B^+-tree with Index and Sequence Set

In order to fully appreciate a B^+-tree, one needs to consider the advantage of using it to perform sequential processing and 'find next' operations. Since horizontal

pointers can be followed during sequential processing of a file, no node will be accessed more than once, so space for only one node need be available in primary memory. Similarly, at most one access can satisfy a 'find next' operation. Besides these advantages, the B^+-tree approach retains logarithmic access time properties for random access. Thus, B^+-trees are well suited to applications which require both random and sequential processing.

Prefix B^+-Trees

In a B^+-tree, only the keys in the bottom level are associated with data records. Keys in upper B^+-index nodes are duplicated from bottom level keys and serve merely as a roadmap to guide the search to the correct leaf. This fact implies that there is no need to store actual keys in the upper level nodes as long as they can direct the search path correctly. This suggests a way for further improvement. Bayer and Unterauer (3) propose a refined structure, the Prefix B^+-tree, which stores parts of keys, namely, prefixes, in the upper index part of a B^+-tree. The major advantage of a prefix B^+-tree is that it decreases access time as well as saves space, as may be seen in the subsequent discussion.

Bayer and Unterauer (3) actually call their data structure a prefix B-tree even though they define their data structure based on a 'B*-tree'. There is some inconsistency in B-tree literature about 'B*' and 'B^+'. Since a 'B*-tree'

is defined as a B^+-tree in this report, the name prefix B^+-tree is then chosen for Bayer and Unterauers' data structure. Figure 6 illustrates the general concept of prefix B^+-trees. Suppose that a leaf is already full and contains the sequence of keys 'index', 'key', 'pointer' and 'search'. In order to insert the key 'separator', this leaf node must be split into two and the key 'pointer' could propagate into the upper index as usual. In fact, however, any of the strings, 'pointe', 'point', 'poin', 'poi', 'po', or 'p' would do as nicely as 'pointer' does. Since it makes no difference for directing searches to leaves, the shortest one among these candidates, say 'p', can be chosen to save space.

Two kinds of prefix B^+-trees are described by Bayer and Unterauer, simple prefix B^+-trees and prefix B^+-trees. A simple prefix B^+-tree is a B^+-tree in which the B^+-index is replaced by a B-tree of separators. Those separators are prefixes of actual keys which are chosen carefully to minimize their length. In prefix B^+-trees, the prefixes are not fully stored due to the fact that all the keys in a given B^+-tree subtree may share a common prefix. If the common prefix can be reconstructed from the subtree's predecessor as the tree is searched, then it need not ever be represented within the subtree itself. Therefore, the length of separators can be further reduced.

It should be noted at this point that in textual database environments, actual keys in leaves are variable in

length as well as separators in upper level nodes. Thus, both separators and actual keys can easily be accommodated by controlling the number of occupied bytes or words in a node rather than the number of keys or separators. However, additional structure information such as number of words or bytes used, number of separators stored, and length of each separator may be required to be kept in a given node in order to facilitate subsequent updates and internal searches. Two alternative node organizations of prefix B^+-trees are shown in Figure 7.

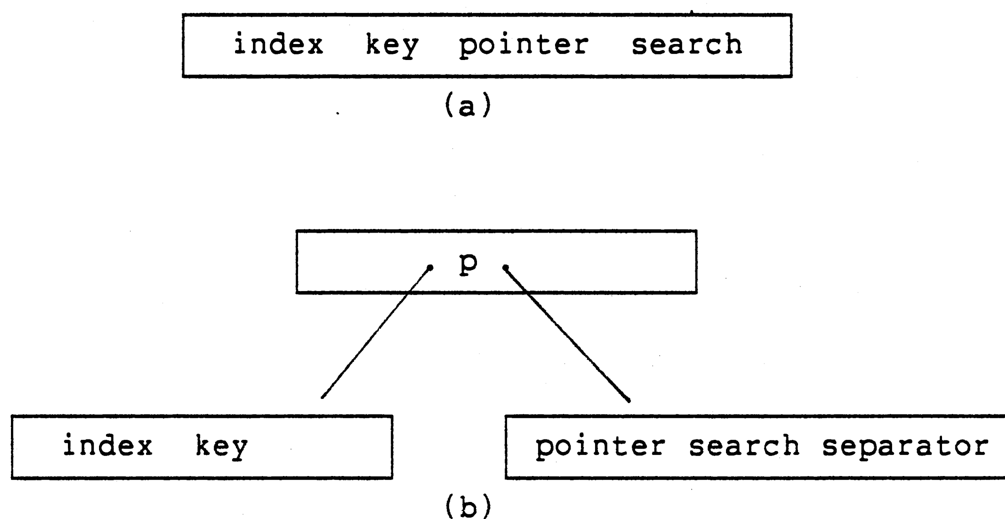
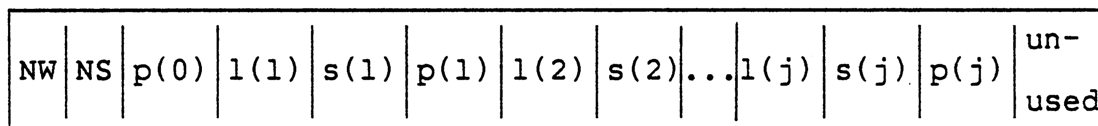


Figure 6. (a) A Leaf Node of a B^+-tree; (b) Result of Inserting the Key 'SEPARATOR' into the Leaf Node of (a)

NW, NS and l(i) represent number of words or bytes used, number of separators stored within this node and the length of the separator s(i) respectively. In the upper nodes, p(i) is a pointer to a descendant node as usual. However, in the leaf node, s(i) is an actual key and p(i) may have several interpretations, such as a pointer to an external node which might be a data record or an accession list identified by s(i), or the data record or the accession list itself. In the latter case, if the data record or the accession list is variable in length, then one more field which indicates the number of words occupied by such data record or accession list needs to be associated with p(i). The last pointer in each leaf node does not associate with any key in that node, so that it can be used as a horizontal pointer to the next leaf to the right. It should be noted that the structure of leaves need not be identical to that of upper level nodes. Moreover, it is possible to have several types of leaves residing in the bottom level in some practical applications.

Internal searches can make use of NW, NS and l(i) to either rapidly and precisely position the next separator, or detect whether or not successive separators reside within this node. During insertion and deletion, NW can be utilized to determine if splitting or merging is necessary to be performed. Of course, the information needs to be updated each time insertion or deletion is encountered.



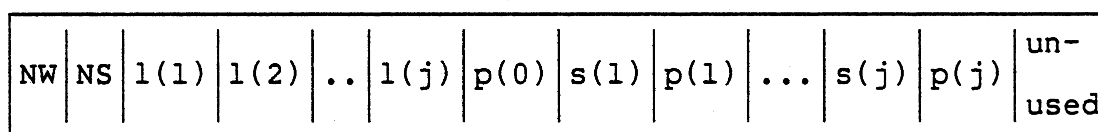


Figure 7. Two Possible Node Organizations for Prefix B^+-trees

Simple Prefix B^+-Tree

Bayer and Unterauer (3) defined the separator as: Let x and y be an arbitrary adjacent pair of real keys which consist of alphabetic characters and the ordering of the keys is the alphabetic order, then any string s with the property

$$x < s \leq y$$

can be used as a separator to separate x and y. Among those possible separators, a unique prefix $\bar{y}$ of y, such that no other separator between x and y is shorter than $\bar{y}$, is chosen to be the separator in the simple prefix B^+-tree approach. Thus, the separator used in this approach is the prefix of the larger key in a key pair and its length should be as

short as possible.

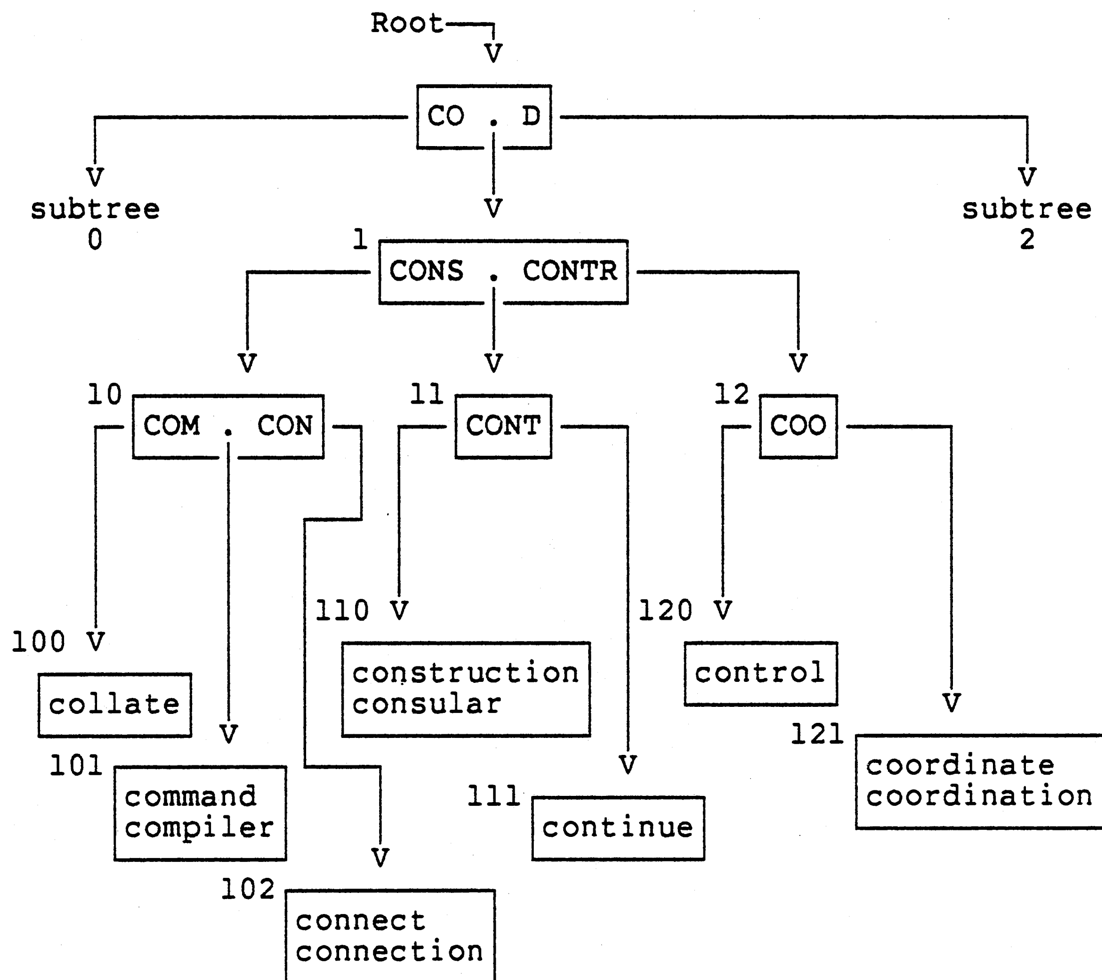


Figure 8. A Simple Prefix B^+-tree

According to Bayer and Unterauer's (3) suggestion, simple prefix B^+-trees only allow the shortest separators being moved from the leaf node to its predecessor node when the leaf node is being split. When a nonleaf node is being

split, one of the separators of that node is moved up one level, no further compression is performed on it. The insertion, deletion and search algorithms applied to simple prefix B^+-trees are similar to those on B^+-trees, except that variable length separators are used to guide the search. Figure 8 depicts an example of a simple prefix B^+-tree in which separators in upper level nodes are represented by upper case letters, while lower case letters are used for actual keys in the bottom level.

Prefix B^+-Trees

In fact, sets of keys that arise in practical textual database applications are often in clusters. This implies that the collating sequence 'distance' between successive separator words may be small and hence all the separators in a given subtree of a simple prefix B^+-tree may share a common prefix. With the goal of further reducing the height of the index part of simple prefix B^+-trees, the common prefix can be kept in the predecessor nodes rather than repeatedly stored in the subtree itself as long as the common prefix can be reconstructed from the subtree's predecessor. Based upon this idea, Bayer and Unterauer proposed the prefix B^+-tree.

Consider Figure 9 as a partial index structure of a simple prefix B^+-tree. Node P denotes an arbitrary upper level node, LL(P) and SU(P) are the largest lower bound and the smallest upper bound of node P respectively, which are

determined from the predecessor node of node P by tree structure definition. For all keys k or separators s which are or might be stored in node P or the subtree with node P being the root, the following holds:

$$LL(P) \leq k < SU(P)$$

$$LL(P) \leq s < SU(P).$$

In node p, p(0), p(1),....., p(j) are pointers to the successors of node P, which are denoted as node p(i) for $0 \leq i \leq j$, and can be either upper level nodes or leaf nodes; s(1), s(2),....., s(j) are separators, s(j) being the last one on node P. In order to focus attention on the separators and pointers, other structural information which may facilitate search and update processes is not presented.

Similar to LL(P) and SU(P), let LL(p(i)) and SU(p(i)) for $0 \leq i \leq j$ denote the largest lower bound and the smallest upper bound of node p(i). Therefore, LL(p(i)) and SU(p(i)), for $0 \leq i < j$, correspond to the leftmost and rightmost entries in each of the following pairs, respectively:

(LL(P), s(1)), (s(1), s(2)), ..., (s(j), SU(P)).

That is (3, p.17),

$$LL(p(i)) = \begin{cases} s(i) & \text{for } i = 1,2, \ldots.,j \\ LL(P) & \text{for } i = 0 \end{cases}$$

$$SU(p(i)) = \begin{cases} s(i) & \text{for } i = 0,1,\ldots..,j-1 \\ SU(P) & \text{for } i = j. \end{cases}$$

Then obviously, if all separators or keys in node p(i)

have a nonempty common prefix c(i), it must be the one defined as follows: Let $\bar{c}(i)$ be the longest common prefix (possibly the empty string) of LL(p(i)) and SU(p(i))), then the common prefix c(i) of node p(i) is defined as:

$$c(i) = \begin{cases} \bar{c}(i)l(j) & \text{if } LL(p(i)) = \bar{c}(i)l(j)z \text{ and } SU(p(i)) = \bar{c}(i)l(j+1)\text{, where } l(j) \text{ preceed } l(j+1) \text{ immediately in the collating sequence and } z \text{ is an arbitrary string} \\ \bar{c}(i) & \text{otherwise.} \end{cases}$$

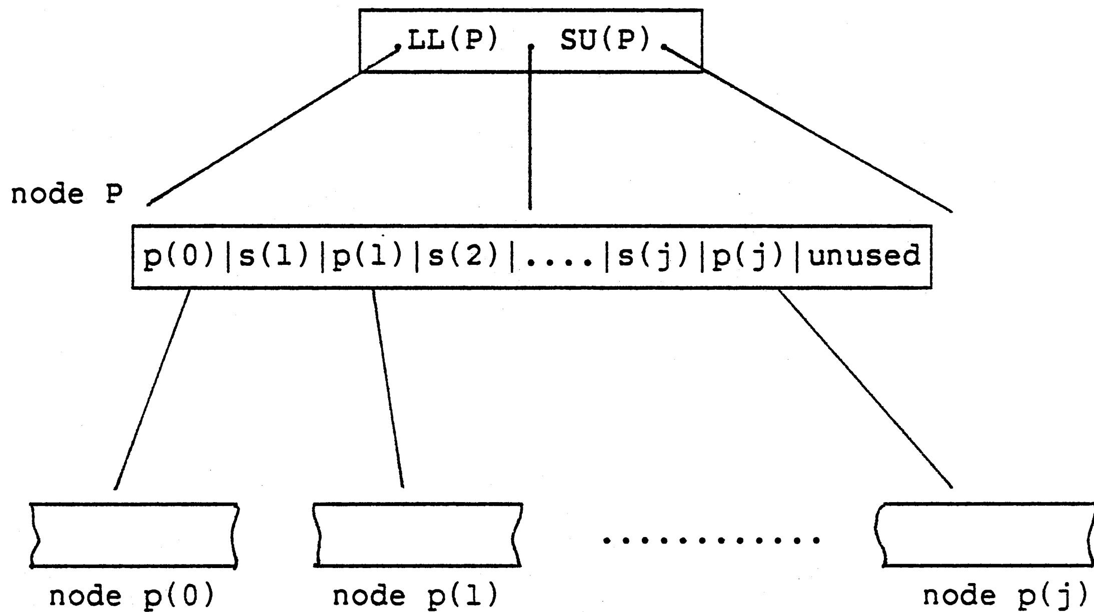


Figure 9. Partial Index Structure of a Simple Prefix B^+-Tree

Reconsider the simple prefix B^+-tree in Figure 8. It could be found that there are several adjacent separator

pairs sharing the common prefix which leads the same common prefix to be repeatedly stored in the lower levels. Based upon the simple prefix B^+-tree in Figure 8, Figure 10 shows the following: (a) the separator pairs sharing common prefixes, (b) the shared common prefix c(i), (c) the rule used to determine c(i): rule 1 represents c(i) = c(i)l(j), while rule 2 represents c(i) = $\bar{c}$(i), and (d) the nodes from which c(i) can be removed.

TABLE I

COMMON PREFIX IN SIMPLE PREFIX B^+-TREE OF FIGURE 8

separator pairs	common prefix	rule used	node from which c(i) can be removed
CO, D	C	1	1, 10, 11, 12
CO, CONS	CO	2	10
CONS, CONTR	CON	2	11

From Figure 10, it is apparent that the prefix 'c', 'co', 'con' and 'c' can be removed from node 1, 10, 11, and 12 respectively. Therefore, by using this prefix compression technique, the simple prefix B^+-tree in Figure 8 can then be modified to yield a prefix B^+-tree which is illustrated in Figure 10.

Prefix compression on a leaf node without regard to its

predecessor can be employed to facilitate sequential processing without ancestry information.

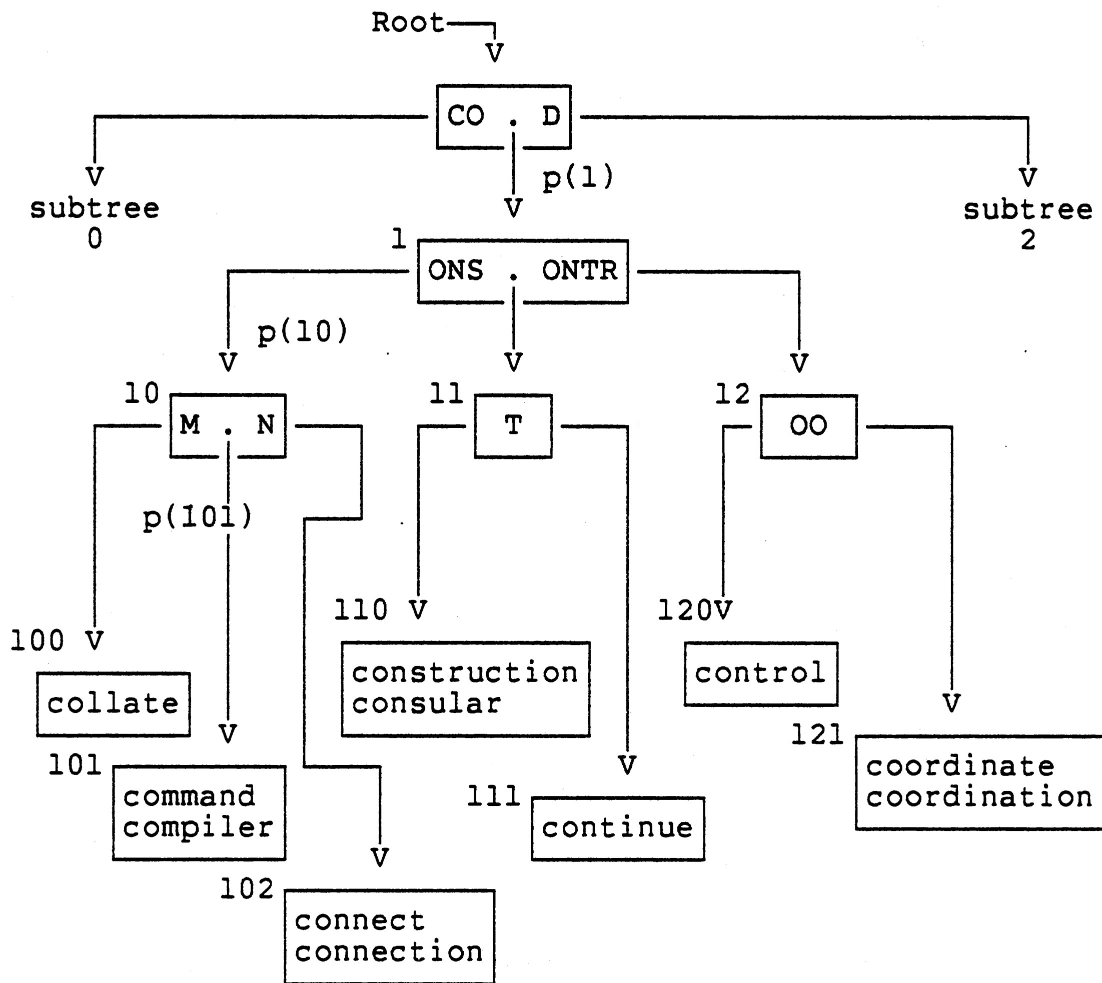


Figure 10. Prefix B^{+}-Tree Derived from the Simple Prefix B^{+}-Tree of Figure 8

<u>Search, Insertion and Deletion</u>

Prefix B^+-trees are designed to combine some of the advantages of B-trees, digital search trees and key compression techniques as may be seen in the subsequent examples which illustrate the underlying algorithms for processing prefix B^+-trees. All the examples are based upon the prefix B^+-tree depicted in Figure 10.

To search for a key 'COMPLEX', the following steps are encountered:

1. Search root node Root, pointer p(1) is to be followed, since 'CO' < 'COMPLEX' < 'D'.
2. Determine the common prefix for node 1 from 'CO' and 'D', yielding 'C'.
3. Remove 'C' from 'COMPLEX', yielding 'OMPLEX'.
4. Search node 1, pointer p(10) is to be followed, since 'OMPLEX' < 'ONS'.
5. Determine the common prefix for node 10 from 'CO' and 'CONS', yielding 'CO'.
6. Remove 'CO' from 'COMPLEX', yielding 'MPLEX'.
7. Search node 10, pointer p(101) is to be followed, since 'M' < 'MPLEX' < 'N'.
8. Search node 101 for the full key 'COMPLEX". Search terminates unsuccessfully since 'COMPLEX' is greater than the largest key 'COMPILER' in this node.

Searching for a key on an index node can be summarized as two steps: (1) Determine the common prefix for this node from its largest lower bound and its smallest upper bound. (2) Remove this common prefix from the original search argument, and then compare this new search argument against the partial separators in this node to locate the descendant

node which needs to be examined next.

General compression techniques, such as front compression, rear compression or combination of these two are to eliminate as many as possible characters from keys according to some rules, as long as the current key differs from the previous and the next one (5). Since the length and the characters of the removed parts are not the same, a significant amount of processing overhead is required as searching proceeds, namely, the need to decompress the keys on the current node first or to change the search argument to be used for comparison with each search step. On the other hand, the way the common prefixes of prefix B^+-trees are constructed is very similar to the way of constructing prefixes in traversing digital search trees, which are to be examined in the next chapter. The compressed portions for all the keys reside in one node of prefix B^+-trees, are identical and are easily determined along the search path. It is now clear that prefix B^+-trees avoid the main disadvantage of other compression techniques in terms of reducing the processing overhead.

Since a failed search operation may be immediately followed by an insertion operation and a successful one may be immediately followed by a deletion operation, there is a need to keep the common prefix of each node along the search path for later use.

To insert the key 'CONSTRUCT' and the associated information, the following steps are encountered:

1. Follow the search scheme just described to see if the key is already present. The search path is node Root -> node 1 -> node 11 -> node 110.

2. Since the search failed, the key 'CONSTRUCT' and the associated information needs to be inserted at the position in node 110, before the key 'CONSTRUCTION'.

3. Suppose an overflow on node 110 occurs as 'CONSTRUCT' is attempted to be inserted. Node 110 is then split into two nodes 110 and 110'.

4. There are two possibilities to rearrange 'CONSTRUCT', 'CONSTRUCTION', and 'CONSULAR' into node 110 and 110'. One of them is to place 'CONSTRUCT' and 'CONSTRUCTION' into node 110 and place 'CONSULAR' into node 110'. Thus, a new separator 'CONSU' is selected to separate node 110 and 110'. (another possiblity will be illustrated later.)

5. The common prefix of node 11, the predecessor node of node 110 and 110', is 'CON', therefore, the partial separator 'SU' is then inserted into node 11 without affecting the other separator 'T' on node 11.

Of course, splits may propagate toward the root and trigger further splits. In the worst case, splitting propagates all the way to the root and the tree increases in height by one level. Figure 11 depicts the new prefix B^+-tree after 'CONSTRUCT' is inserted according to the above steps.

In most cases, the insertion can be completed by simply inserting the key and the associated information into a leaf node. However, the insertion process is quite complicated if overflow conditions are encountered. There are two strategies, namely, node splitting and node equalization that can be used to handle overflow conditions. The former is the one used in the previous example which splits the

overflow node into two and propagates the separator of them into their predecessor node, while the later employes a local distribution scheme to delay splitting until 2 sibling nodes are full.

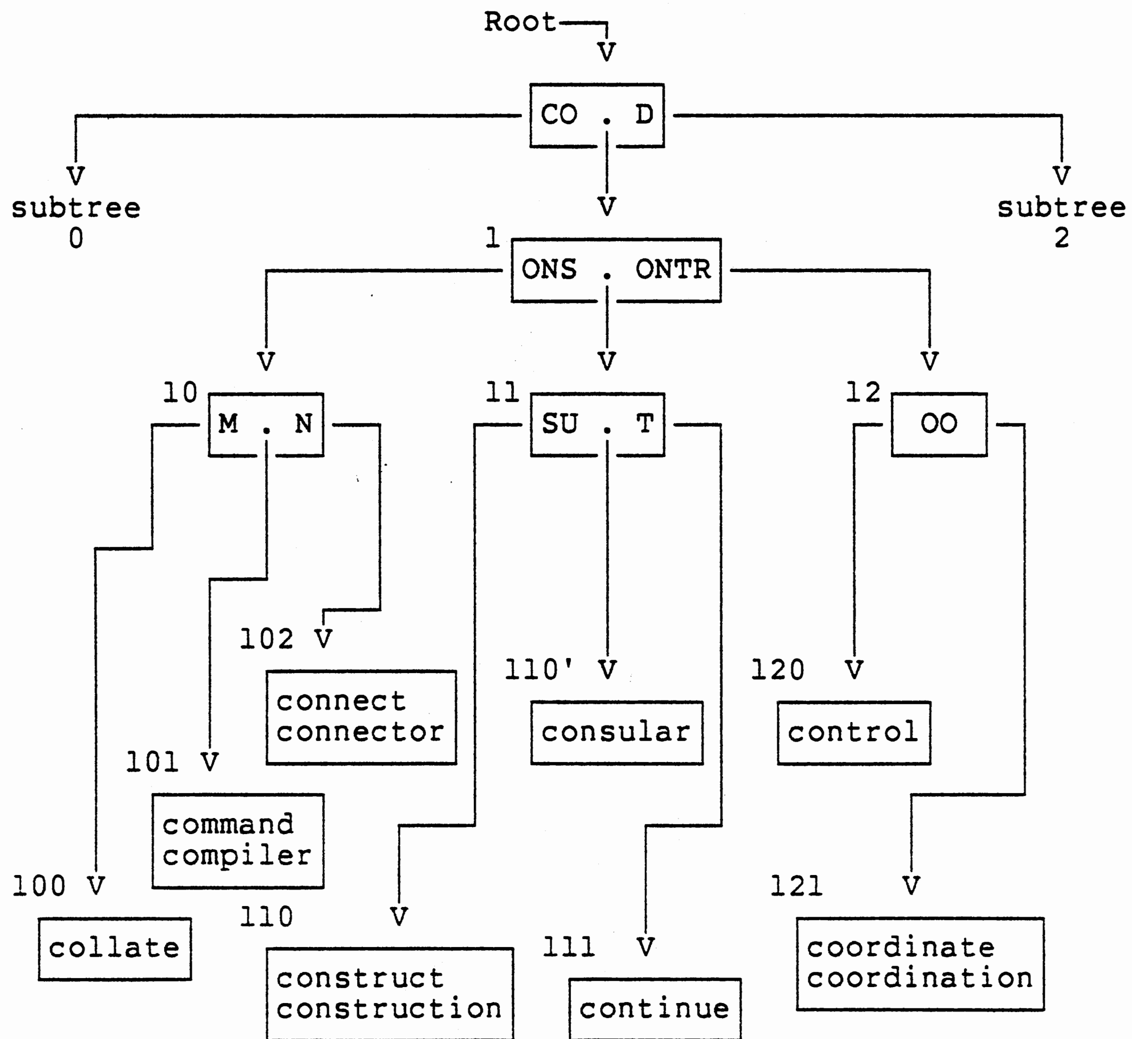


Figure 11. Result of Inserting the Key 'CONSTRUCT' into the Prefix B^+-Tree of Figure 10

More complications arise in a prefix B^+-tree environment than in a conv entional B-tree or B^+-tree. The lengths of partial separators stored in a given node, say P, are affected by its largest lower bound and smallest upper bound which are the partial separators stored in node Q, the predecessor node of node P. Inserting (in the case of node split) or replacing (in the case of node equalization) a partial separator into node Q might change the common prefix for node P and/or its sibling node, and will cause the partial separators on them to shrink or expand. therefore, both (1) predetermining whether or not the equalization to the sibling node is possible and (2) recomputing the partial separators on node P may be required for overflow handling.

Moreover, Bayer and Unterauer suggest splitting a node within an interval around the median key instead of splitting precisely in the middle when a node must be split. Their idea can be illustrated by the previous example: Recall when 'CONSTRUCT' is inserted into node 110, the key sequence in that node is 'CONSTRUCT', 'CONSTRUCTION' and 'CONSULAR'. Splitting this sequence in the middle between the first and second would yield 'CONSTRUCTI' as the shortest separator. Allowing a split point to be chosen one key to the right yields 'CONSU' as separator. This idea can be applied to split both leaf nodes and the upper level nodes. Since similar keys differing only in the last few letters are quite common in practical applications, allowing

selection of the shortest separator within a small split interval may decrease the length of the shortest separators, and increase the branching degree of nodes, so that it tends to decrease the height of the index part and improve performance. The tradeoff of allowing a split internal is a decrease of storage utilization because there can now be nodes less than half full.

To delete a key 'CONTROL' and the associated information, the following steps are encountered:

1. Follow the search scheme to locate the leaf node containing the key 'CONTROL', node 120 is found.

2. Delete 'CONTROL' and its associated information from leaf node 120.

3. Node 120 becomes empty after the key 'CONTROL' is deleted. Therefore, merging node 120 with node 111 or redistribution of keys between node 120 and node 121 is required.

4. Suppose that merging node 120 with node 111 is chosen, then node 120 is discarded. The merging process propagates one level up, that is node 12 must be merged onto node 11.

5. Delete the partial separator 'ONTR', which is between node 11 and the original node 12, from their common predecessor node, node 1, because it has no need to remian.

6. Recompute the common prefix for node 11 from its largest lower bound 'CONS' and smallest upper bound 'D', yielding 'C'.

7. Recalculate the old and new partial separators, 'T' and 'OO', on node 11, yielding 'C'.

Figure 12 shows the new prefix B^+-tree after 'CONTROL' is deleted according to the above steps. Figure 13 depicts the new prefix B^+-tree in the case that redistribution between node 121 and the original node 120 occurs after

'CONTROL' is deleted. Notice that in the latter case, deletion is done at the leaf level.

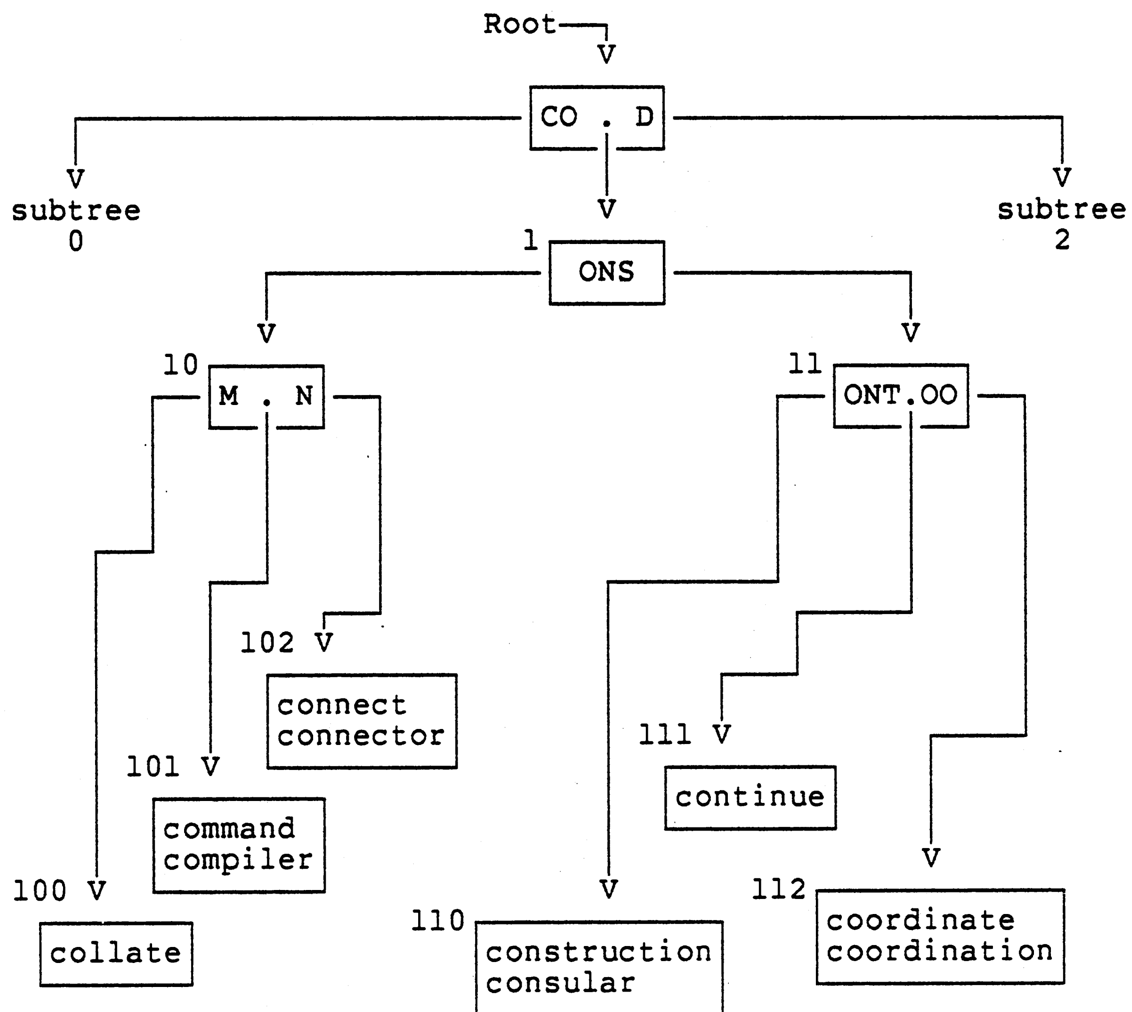


Figure 12. Result of Deleting the Key 'CONTROL' from the Prefix B^+-tree of Figure 10, Merging Scheme Is Used

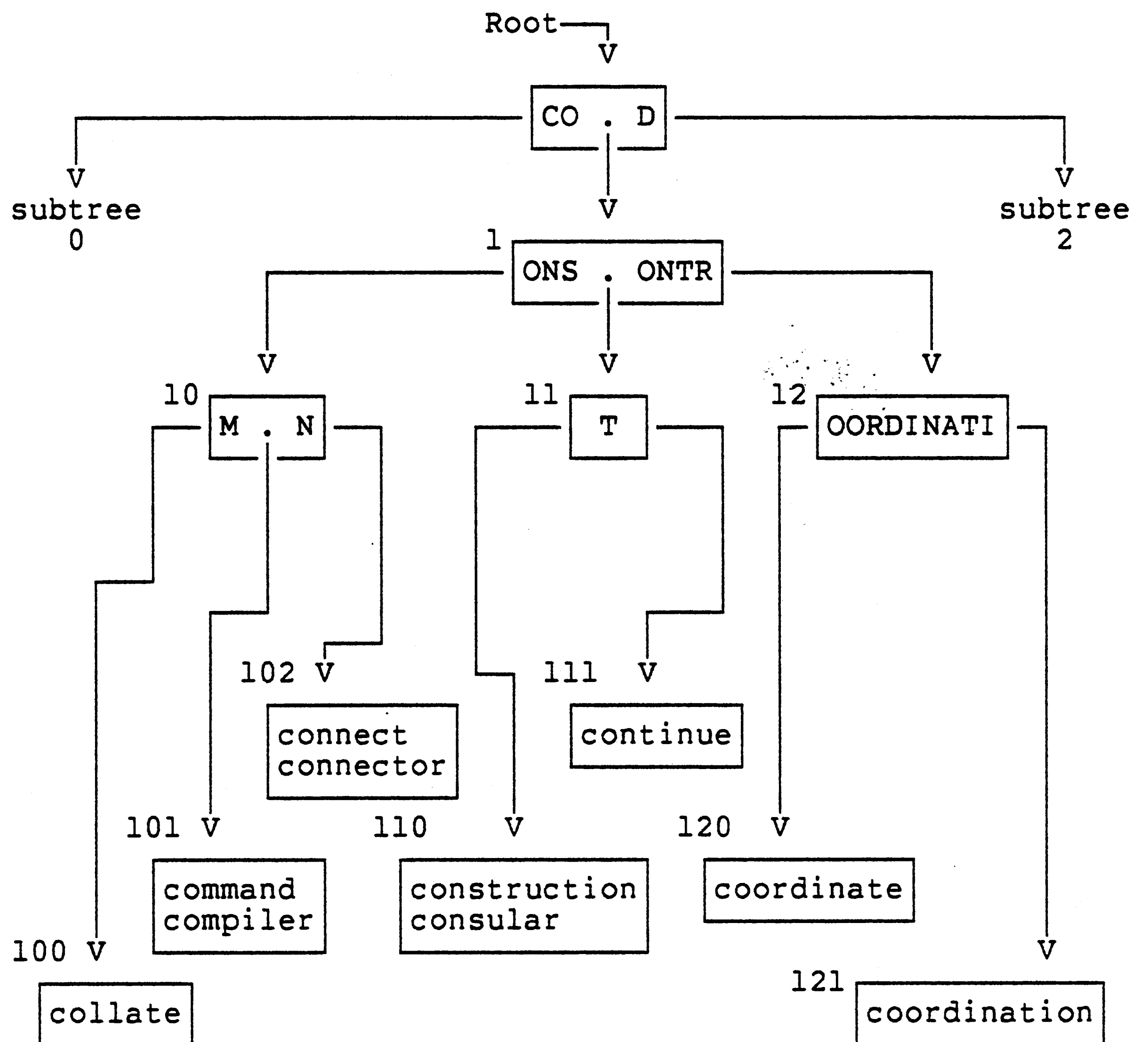


Figure 13. Result of Deleting the Key 'CONTROL' from the Prefix B⁺-tree of Figure 10, Redistribution Scheme Is Used

Deletion is the inverse of insertion. It always starts at a leaf node and in most cases, it is completed by simply

deleting the key and the associated information from the leaf node. However, deletions encounter similar complications that occur in splitt ing a node during insertion, if merging two nodes must be done. For example, merges may propagate toward the root, and the common prefix of the merged node may change because of altering its largest lower bound or smallest upper bound. Therefore, both predetermining and recomputing work which was mentioned in the insertion process is also required for deletion.

Nevertheless, it should be recalled here that in a prefix B^+-tree, the B^+-index is separate from the leaf nodes and all actual keys reside in the leaves. Therefore, it doesn't matter which values are encountered through the search path as long as the path leads to the correct leaf. This feature simplifies the deletion operation for prefiex B^+-trees. If the leaf remains at least half full after deleting keys from it, the index needs not be changed even though the prefix of this deleted key was selected as a separator. Of course, if the leaf node is less than half full, the merging or redistribution procedure is used to adjust values in the B^+-index as well as in the leaves.

Evaluation of Prefix B^+-Trees

Both simple prefix B^+-trees and prefix B^+-trees are alternatives of B^+-trees. They combine some of the advantages of B-trees, digital search trees and compression techniques without inheriting their main disadvantages. The

following summarize Bayer and Unterauer's (2, p.24) evaluation:

1. The basic advantages of B-trees, such as guaranteeing good worst-case performance and good storage utilization, are preserved.

2. The technique of constructing prefixes while traversing the tree during a search is similar to digital search trees without the danger of obtaining unbalanced trees.

3. The techniques of key compression, such as choosing shortest separators (as in rear compression) and pruning off the common prefix from shortest separators (as in front compression) are applied without excessive processing overhead.

The main advantage of simple prefix B^+-trees and prefix B^+-trees are to increase the branching factor, save space, decrease the height of the tree, and hence possibly decrease access times. However, this method of indexing also introduces additional complicating factors as follows:

1. The separators or partial separators are variable length strings, so that each node can have a different branching factor which is determined by the internal organization of a node. The index building and maintainance algorithms do not know beforehand how many separators can be packed into a node. They must have the capability of handling variable length separators.

2. The additional time required to search a node after it has been read is inevitable due to the varying location of separators within a node.

3. The separator which is propagated must be unique in that upper level node. A mechanism must therefore be added to the node splitting algorithm to insure uniqueness.

4. Additional processing may be required for some insertions or deletions which may alter the longest common prefix, if prefix B^+-trees indexing is used.

According to Bayer and Unterauer's (3) experimental results, the computing time and saving of disc accesses of

using simple prefix B^+-tree and prefix B^+-tree compared to using the B^+-tree in a dynamic environment are shown as follows:

Computing Time - The time to execute the algorithms for simple prefix B^+-tree is almost identical to the time for B^+-trees, while prefix B^+-trees need 50-100 percent more time.

Saving of disk accesses - If trees have less than 200 pages, no saving is achieved. For trees having between 400 and 800 pages, simple prefix B^+-trees require 20-25 percent fewer disk accesses than a B^+-tree. Prefix B^+-trees need about 2 percent fewer disk access than simple prefixB^+-trees.

The above results suggest that simple prefix B^+-trees are more cost effective than prefix B^+-trees in a dynamic environment. However, the prefix B^+-tree is probably superior to simple prefix B^+-tree in a static environment because minimizing the search time to an index is more important than minimizing its set up time. For relatively static databases, the prefix B^+-tree index can be constructed from a sorted list of keys which identify the records of that database. The largest common prefix of separators or keys can be factored out as usual, but kept within the same node as the separators or keys reside on. This modification requires slightly more storage space but simplifies the search logic. Although the index building process for prefix B^+-tree index is quite complicated and costly, it is still worth doing it to obtain the advantages of less storage and fast retrieval in a relatively static environment.

CHAPTER IV

TRIE STRUCTURE INDEXING

Besides prefix B^+-trees, the other particularly useful index structure for handling varying size keys is the trie. This name was suggested by E. Fredkin (10) in 1960 because it is a part of information "retrieval". The basic idea behind the trie structure is to view a key as having multiattributes. The branching at any level of a trie index is governed not by the entire key value but by only portion of it. That is to say, instead of basing a search method on comparing the entire key values in the conventional way, one can make use of their representation as a sequence of digits or alphabetic characters to build a trie index. The advantage of a trie implementation is having potentially fast access time, but the disadvantage is the relative inefficiency in using storage space. Several approaches are presented to improve the disadvantage of the inefficient storage utilization of a trie, as may be seen in the subsequent discussion.

Digital search trees are illustrated in the first section of this chapter. The digital search tree is a general structure for dealing with multiway branching decisions based on portions of keys. The trie structure is

commonly considered a particular type of digital search tree, even though trie structures were developed earlier than digital search trees. The second section presents the basic trie structure and the ways it can be improved. The third section examines some refinements and variants of trie structures, such as pruned tries, O-tries, linked list implementation of tries and C-tries.

Digital Search Trees

The search methods can be classified into two categories according to whether they are based on comparisons between keys or on digital properties of the keys (16). The conventional search methods, such as B-trees and binary search trees fall into the first category, while the digital search tree is a good example of the second category.

A digital search tree is essentially an m-ary tree. Keys of the digital search tree are considered binary coded to form 0, 1 bit strings. These bit strings are partitioned into substrings of equal length, such that these substrings viewed as binary numbers have values between 0 and m-1. As an example consider the binary case m=2, in which the search argument is scanned one bit at a time, that is, the length of the partitioned substrings is one. Figure 14 depicts such a digital search tree for 10 common programming languages and software packages, inserted in increasing lexical order. In order to simplify this example, Knuth's

MIX character code (16) is used to provide binary data for this illustration: the keys have been expressed in MIX character code which is then converted into binary numbers with five bits per byte. Table II shows Knuth's MIX character code and bit string representation for each alphabetic character.

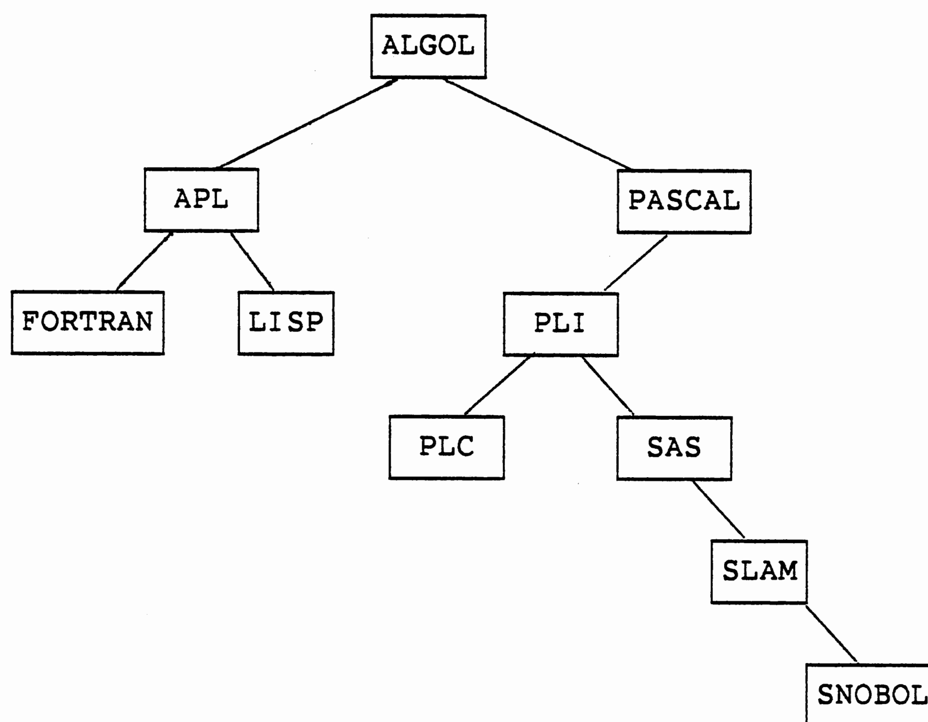


Figure 14. A Digital Search Tree for 10 Common Programming Languages and Software Packages, Inserted in Increasing Lexical Order

TABLE II

KNUTH'S MIX CHARACTER CODE AND BIT STRING REPRESENTATION FOR EACH ALPHABETIC CHARACTER

Alphabetic Characters	MIX Character Code	Bit String Representation
A	1	00001
B	2	00010
C	3	00011
D	4	00100
E	5	00101
F	6	00110
G	7	00111
H	8	01000
I	9	01001
J	10	01010
K	11	01011
L	12	01100
M	13	01101
N	14	01110
O	15	01111
P	16	10000
Q	17	10001
R	18	10010
S	22	10110
T	23	10111
U	24	11000
V	25	11001
W	26	11010
X	27	11011
Y	28	11100
Z	29	11101

From Figure 14 it should be noticed that full keys are stored in the nodes of the digital search tree as in the conventional tree structure, but bits of the search arguments are used to govern whether to take the left or right branch at each step. Suppose that the word SAS, whose bit string representation is '10110 00001 10110', is searched in the tree of Figure 14. SAS is first compared with ALGOL at the root of the tree. Since there is no match and the first bit of SAS is 1, the search path is turned to the right and SAS is compared with PASCAL. Since there is no match and the second bit is 0, the search path is turned to the left and SAS is compared with PLI; and so on, until SAS is found (in the case of Figure 14) or the appropriate place where SAS can be inserted is located.

It is understandable that if bit strings, which represent keys, are partitioned into substrings of length two, then the branching factor m of each node could be four, each of them corresponds to one of the values 0, 1, 2 and 3. Thus, the search arguments need to be scanned two bits at a time. It is not difficult to see that the same branching strategy used in the binary case could also be applied to an m-ary digital search tree for any m>>>2.

Basic Trie Structure

Although the trie structure is commonly considered a particular type of digital search tree, it differs from the basic digital search tree in two major aspects: First,

the branching at any node in a trie structure is governed by constituent character(s) or digit(s) rather than constituent bit(s) of the keys. Second, the key is not recorded in full in a trie structure until the first point where the key is uniquely identified.

Figure 15 shows the basic trie structure for the same key set as in the digital search tree of Figure 14. There are two types of nodes in a trie structure, namely, the branch node and the information node. In Figure 15, branch nodes are represented by solid-line blocks, while broken-line blocks are used for information nodes. Each branch node is an array of m pointer fields with components corresponding to digits or alphabetic characters. If keys are composed of character-valued attributes (English alphabet), there would be 27 entries in each branch node, one for each letter of the alphabet and one for the blank character which is used an end-of-key symbol to insure that no key may be a prefix of another. At level 1 all key values are partitioned into 27 disjoint classes depending on their first character. The i-th pointer field of the root node contains the pointer to a subtrie which contains all key values beginning with the i-th alphabetic letter. On the j-th level the branching is determined by the j-th character of the search argument. It should be obvious that the attribute values should have a small, contiguous range, such as characters or digits, otherwise the size of each node would be large. When a subtrie contains only one

value, it is represented by an information node, a leaf node. The information node contains a key value, together with other associated information, such as the data record or the accession list identified by the key, or the pointer to the data record or to the accession list.

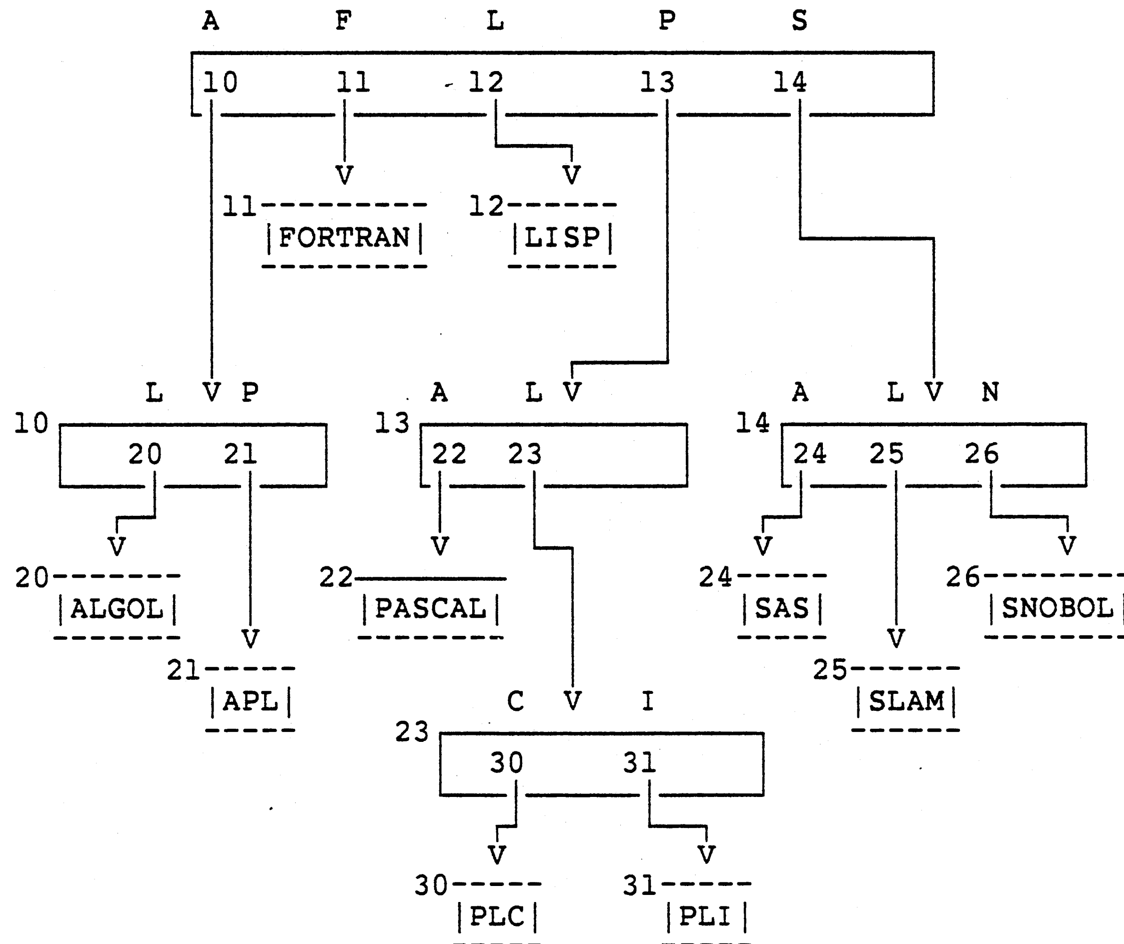


Figure 15. Trie Constructed for Keys of Figure 14, Sampling One Character at a Time, Left to Right

Searching the trie for a key value X requires breaking up X into its constituent characters and following the branching patterns determined by these characters. For example, the word SAS is searched in the trie structure of Figure 15, the first letter of SAS, namely S, is looked up at the root. The pointer field corresponding to S in the root node indicates to go on to node 14 and look up the second letter there. Then, node 14 indicates to go on to node 24 and look up the third letter there. Since SAS is uniquely identified at level 3, an information node (leaf node) is encountered and search is terminated successfully. On the other hand, if the word ASSEMBLER is searched, the root node directs the search to node 10, looking up the second character in the same way; node 10 indicates that the second character should be L or P, otherwise, the search argument is not in the trie. Thus, searching for ASSEMBLER is terminated unsuccessfully.

Both insertion into a trie and deletion from a trie are straightforward. Suppose that two entries, 'PLANS' and 'APPLY', need to be inserted into the trie of Figure 15. First, an attempt to search for 'PLANS' in the given trie terminates unsuccessfully at node 23. Hence, 'PLANS' is not in the trie and may be inserted here. Next, the search for 'APPLY' leads the search path to the information node 21. A comparison indicates that the key in node 21, APL, is not equal to 'APPLY'. Both 'APL' and 'APPLY' will form a subtrie of node 10. The two values 'APL' and 'APPLY' are

sampled until the sampling results in two different values. It happens when the third letter of 'APL' and 'APPLY' are compared. The resulting trie after inserting 'PLANS' and 'APPLY' is given in Figure 16.

Suppose that the key 'FORTRAN' needs to be deleted from the trie of Figure 15. It is done simply by setting the pointer field corresponding to 'F' of the root node to null, no other changes need to be made. Next, suppose that the key 'PLI' needs to be deleted. This deletion leaves only one key value in the subtrie 23. This means that the node 23 may be deleted and node 30 move up one level. the resulting trie after deleting 'FORTRAN' and 'PLI' is in Figure 17.

It is not hard to discover that the branching decision in a trie is simply made by indexing the array of pointers, i.e. the branch node. That is to say, the pointer in the fourth field of the current branch node is followed, if the character examined is D; fifth for E; and sixth for F; etc. Hence, the time required to decide which path to follow at each node is constant. A trie search is quite fast when nodes are already in internal memory. However, when nodes of a trie are kept in secondary storage and require an access to external storage each time they are to be inspected, performance of a trie is significantly affected by the number of levels in that trie. Performance of trie indexes in internal storage vs. external storage is examined in the final section of this chapter.

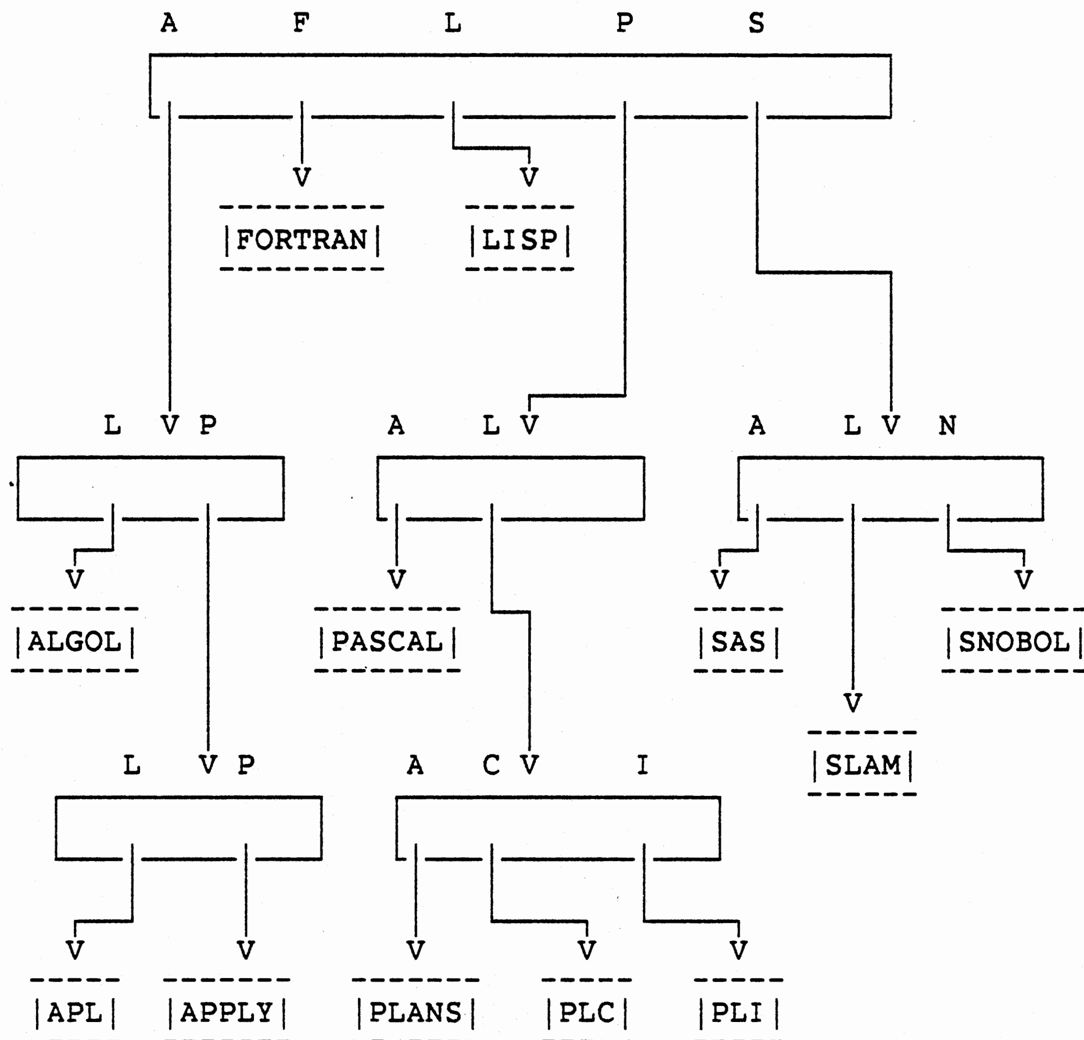


Figure 16. Result of Inserting Keys 'PLANS' and 'APPLY' into the Trie of Figure 15

An unsuccessful search might be faster in a trie index than in a prefix B⁺-tree index because it can be detected

before the leaf node is reached (recall the previous example of searching for ASSEMBLER). Unfortunately, nearly 90% of the arrray entries in Figure 15 are empty, which implies that trie structure may be quite wasteful in space utilization. In fact, high-storage cost is the primary difficulty with the basic trie structure idea.

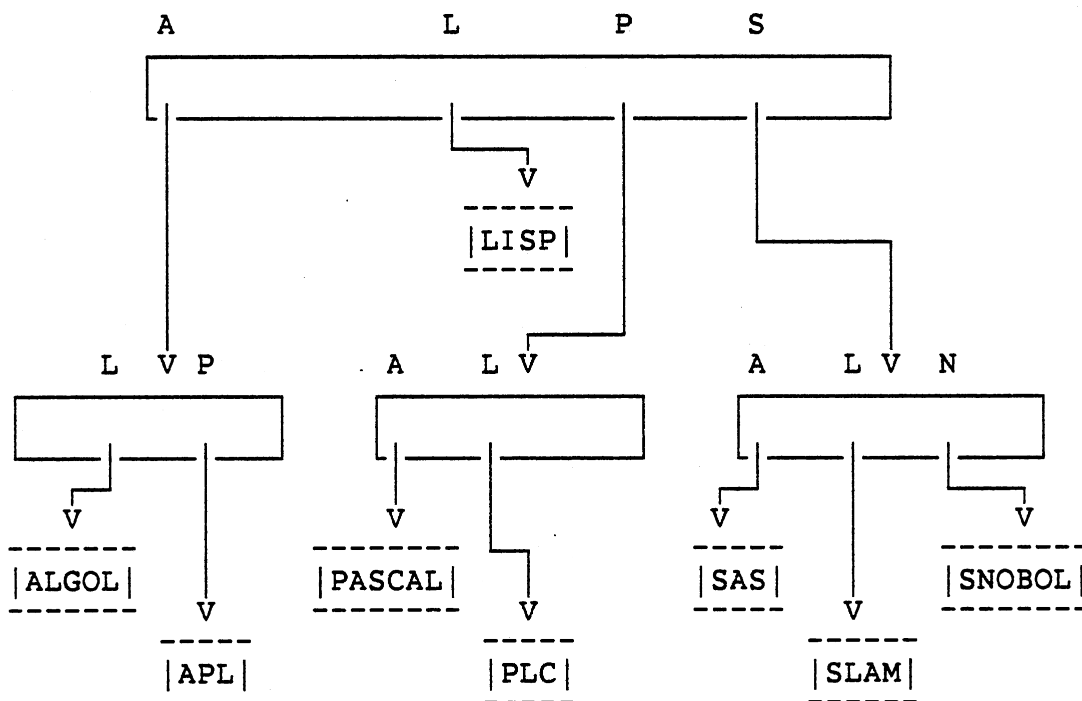


Figure 17. Result of Deleting Keys 'FORTRAN' and 'PLI' from the Trie of Figure 15

There are two approaches which can be used together to achieve better space utilization for using a trie index, namely, reducing the number of levels and reducing the space

required at each node. Several methods for achieving these two goals are known and will be examined in the next section.

Refinements and Variants of Tries

Consider building a trie index, using the same branching strategy as used in Figure 15 for the key set of Figure 8 in Chapter III on page 27. A trie loses its advantage because of the distribution of the keys. For example, a trie requires ten iterations to distinquish between COORDINATE and COORDINATION. Trie structures were originally designed for storing alphabetic character strings, therefore it is understandable that the attribute testing order is from left to right, one at a time. Nevertheless, when a key is viewed as a k-tuple, in which attributes are unrelated, both examining all the attributes of keys and testing attributes in left-to-right order are no longer necessary. This fact leads to several ways to reduce the space requirement of a trie: One is pruning a trie which eliminates useless attributes; the other is reordering attribute testing which moves the useless attributes to the end where they will not be reached during a search. Moreover, the number of levels in a trie can be limited to some fixed number by storing more than one key in each information node (leaf node), so that both the number of branch nodes and the number of information nodes can be reduced.

Pruning a Trie

There are two kinds of tries: (1) tries in which each attribute is tested, and (2) tries in which testing of attributes stops when a key has been distinguished. The former are called full tries, while the latter are often called pruned tries (8). Figure 18 depicts a full trie and a pruned trie in a simplified form (each '.' represents a 27 entry branch node and '=' represents an information node), which corresponds to the trie of Figure 15.

It is obvious that all nodes marked by * in Figure 18(a) do not further divide the sets of keys. Their omission would result in a smaller trie; they are useless. In the pruned trie of Figure 18(b), which is actually the same as the one of Figure 15, there is no internal node corresponding to only one leaf node: all useless attributes are eliminated. It should be noticed here that pruning a trie only eliminates leaf chains but not internal chains. Figure 19 (a) and (b) show a leaf chain and an internal chain respectively. A leaf chain starts a node, the head of the leaf chain, whose predecessor has more than one successor but its descendant and itself have at most one successor. A pruned trie is formed from a full trie for the given key set by deleting descendants of all the heads of leaf chains.

However, by pruning a trie, some information about keys may be lost. So, although correct queries are not affected, some incorrect queries may report success. In this case,

probably one more access to the information node is required to verify whether or not the search is successful.

Pruned tries can be further improved by eliminating internal chains, as will be seen in the section of order-containing tries. Since pruning is such a basic space-saving operation, it is assumed that all tries are pruned hereafter in this thesis.

Reordering Attribute Testing

Given a set of key values to be represented in a trie index, the number of levels in the (pruned) trie will obviously depend on the attribute testing order used to determine the branching at each level. This testing order can be defined by a sampling function SAMPLE(X,i) which appropriately samples the key X for branching at the i-th level (14). Several sampling functions are shown as follows, where X = x(1)x(2)....x(n):

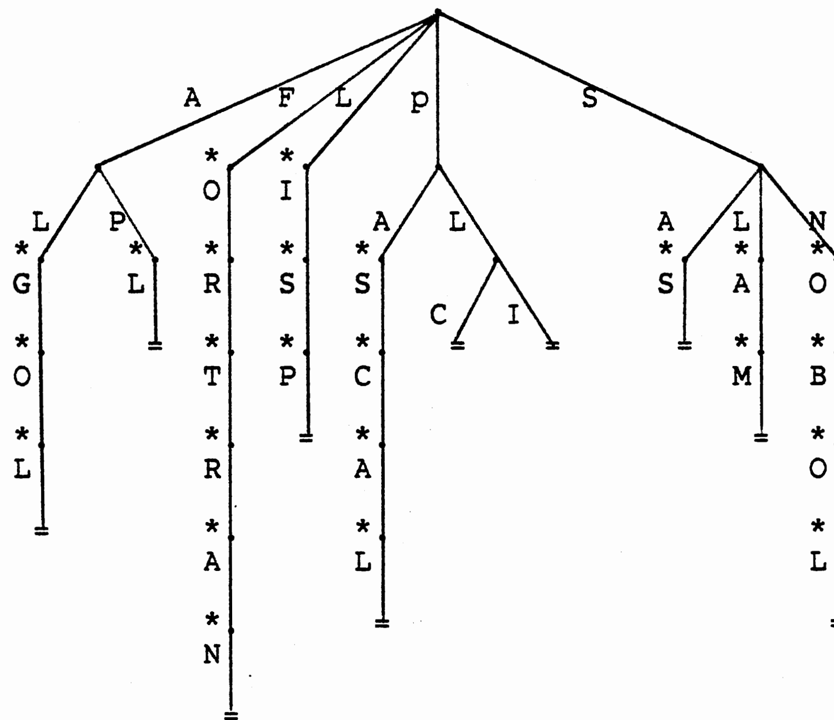
(a) SAMPLE(X,i) = x(i)

(b) SAMPLE(X,i) = x(n-i+1)

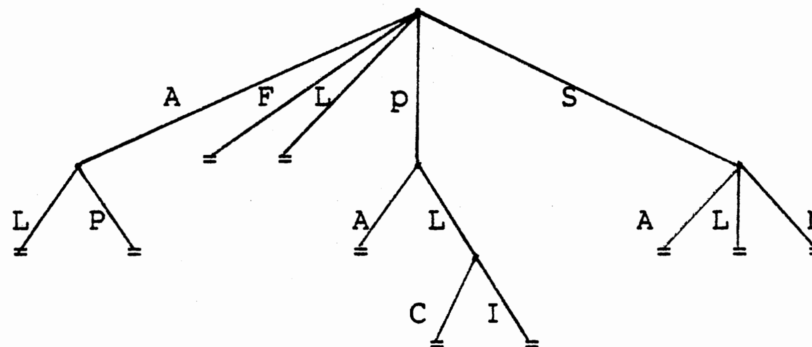
(c) SAMPLE(X,i) = x(r(X,i)), for r(X,i) a randomization function

(d) $$\text{SAMPLE}(X,i) = \begin{cases} x(i/2) & \text{if } i \text{ is even} \\ x(n-(i-1)/2) & \text{if } i \text{ is odd.} \end{cases}$$

The example trie of Figure 16 uses the sampling function (a) and results in four levels, requiring five branch nodes. Using the function (b), which is sampling one



(a)



(b)

Figure 18. (a) A Full Trie, and (b) a Pruned Trie for Keys of Figure 15, Sampling One Character at a Time, Left to Right

character at a time, right to left, on the same key values yields the trie of Figure 20, which has only three levels and requires only two branch nodes. Reordering attribute testing to reduce the size of a trie is an attractive proposition; an ordering which yields a minimum size trie is desirable. However, choosing the optimal attribute testing order or sampling function for any particular set of key values is very difficult. The property of an attribute being useful or useless is related to the occurrence of an attribute in a particular trie and may not be known beforehand. Therefore the attribute testing order for a volatile file must be dynamic and used during or after a trie has been constructed.

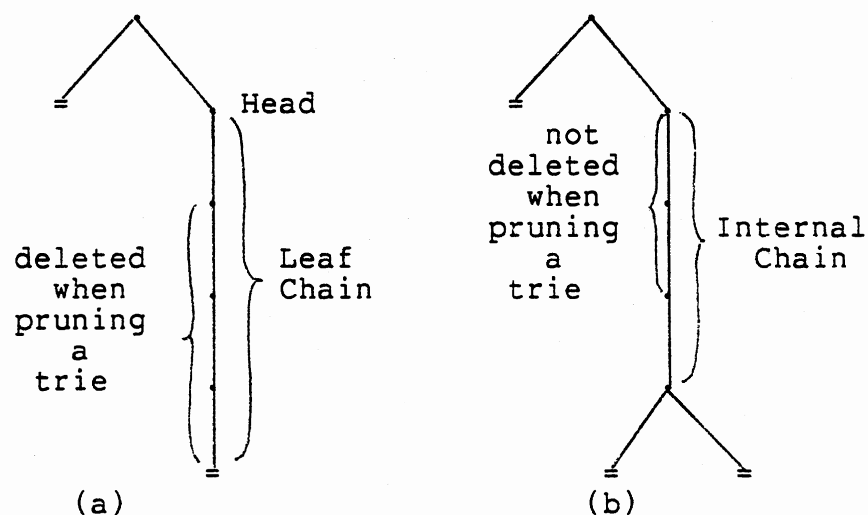


Figure 19. (a) A Leaf Chain and (b) an Internal Chain

Although optimal attribute testing order is very difficult to find, Comer (9) summarized four heuristics, which employ computationally efficient procedures to produce tries which are smaller than a randomly ordered trie, although they may not be minimum. The following shows these four heuristics (heuristic 2, 3 and 4 based on heuristic 1 but several extensions to that idea are considered):

Heuristic 1 - Elimination of Useless Attributes
When building a trie, select a useful attribute at each depth which can further divide the sets of keys.

Heuristic 2 - Splitting Heuristic
When building a trie, select an attribute at each depth which adds the most nodes (including leaves). Among all attributes adding the maximum number of nodes, select one which adds the most leaves.

Heuristic 3 - Greedy Heuristic
When building a trie, select an attribute at each depth which adds the most leaves. Among all attributes adding the maximum number of leaves, select one which adds the most internal nodes.

Heuristic 4 - Leaf Greedy Heuristic
When building a trie, select an attribute at each depth which adds the most leaves. Among all attributes adding the maximum of leaves, select one which adds the fewest number of internal nodes greater than zero.

Heuristic 1 attempts to reduce the space requirements of a trie by eliminating useless attributes. Heuristic 2 tends to break up the sets of keys rapidly, yielding leaves earlier in the trie. Both heuristic 3 and heuristic 4 extend the idea of generating leaves as soon as possible, using it as the primary criterion for selecting attributes. Heuristic 3 reverses the criteria used in heuristic 2, and

the resultant tries tend to be short, but wide. Heuristic 4 attempts to yield leaves as fast as possible, avoid those attributes which would make the trie wide. Thus, the tries produced by heuristic 4 are usually narrow, but long. A more thorough treatment of each heuristic and their cost criterion can be found in Comer (9).

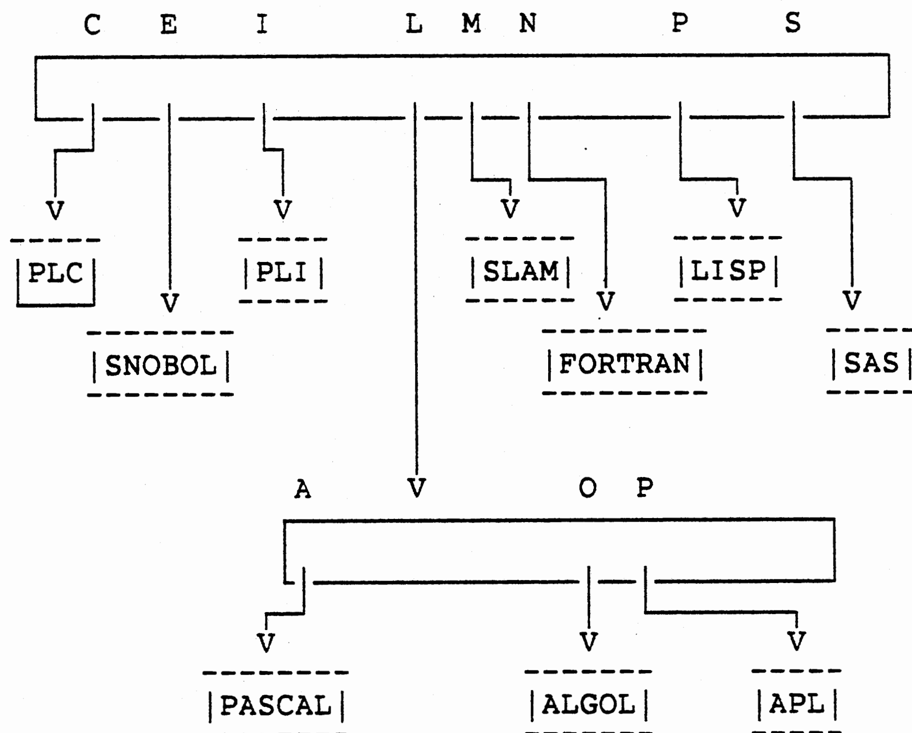


Figure 20. Trie Constructed for Keys of Figure 14, Sampling One Character at a Time, Right to Left

Besides the number of nodes, the maximum number of

levels in a trie is a critical element affecting the trie performance, especially in the case that most of the nodes of a trie index must reside on secondary storage.

Entering Multiple Keys

The maximum number of levels in a trie can be kept low by adopting a different strategy for managing information nodes (leaf nodes). If the maximum number of levels is limited to n, then all key values that are synonyms up to level n-1 can enter the same information node. That is to say, information nodes need to be designed to hold more than one key value. If the attribute testing order is chosen properly, there will be only a few synonyms in each information node and hence can be processed in internal memory during retrieval (14). Figure 21 shows the use of this strategy on the trie of Figure 16 with n=3.

All the above discussions deal with a fixed, global ordering of attribute testing which applies to all paths from a root to a leaf in the trie. Another alternative for reducing the space requirement of a trie is to test attributes in different orders along different paths from a root to a leaf. This implementation is called an O-trie, in which the order of attribute testing is contained in the node itself.

O-tries (Order-containing Tries)

The idea of an O-trie is taken from PATRICIA (Practical

Algorithm To Retrieve Information Coded In Alphanumeric), which is an economical and flexible indexing technique, based on digital properties of keys, designed by Morrison (19). In PATRICIA, each node in the tree includes extra information telling how many attributes to 'skip' in the ordering. Based upon PATRICIA, Douglas Comer (9) proposes an even more generalized structure, an O-trie, in which information is added to each node telling explicitly which attribute to test at that node. Suppose that there are k attributes, only log k extra bits are needed in each node to specify which attribute to test. Figure 22 shows one possible O-trie for the set of keys in Figure 8, in which every branch node has at least two sons. The numbers in the internal nodes of this O-trie represent the position of the letters which should be tested.

Figure 23 shows one of the optimum pruned tries with the global attribute testing order 6, 3, 8, 11. There are four levels and seven branch nodes in this trie and nodes marked by * indicate the internal chains which cannot be removed by just pruning a trie. All the four heuristics for pruned tries produce the same trie as shown in Figure 23, if the attribute testing order is 6, 3, 8 and 11. It can be now concluded that the size of a trie in terms of number of levels and nodes can be reduced further by relaxing the requirement that there be a global testing order.

Building an O-trie probably requires two passes: starting with an arbitrary attribute order, constructing a

trie, and then reordering attribute testing within the various subtries to reduce the size. Although an O-trie is a good approach to reducing the storage requirement for a trie, finding the optimum attribute testing orders for various subtrees still implies much complexity.

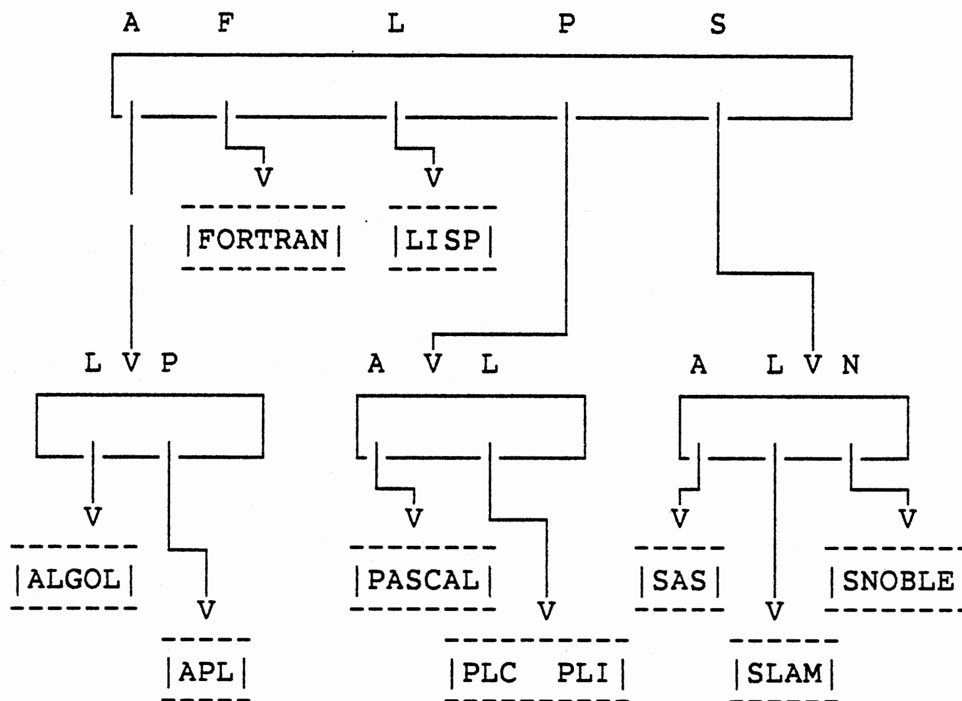
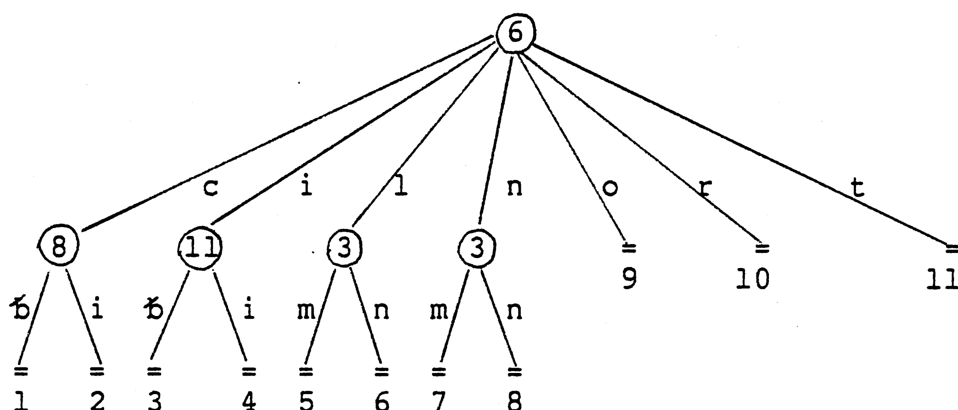


Figure 21. Trie Obtained for Keys of Figure 15 When Number of Levels Is Limited to 3, Key Has Been Sampled Left to Right, One at a Time

The refinements and variants mentioned above, such as pruning a trie, reordering attribute testing, entering multiple keys into the same information node and including the local attribute testing order in each branch node itself, are designed to reduce the number of levels and the number of nodes in a trie structure. The following discussion will illustrate ways to reduce the space required at each node.



1	Connect	2	Connection	3	Coordinate
4	Coordination	5	Compiler	6	Consular
7	Command	8	Continue	9	Control
10	Construction	11	Collate		

Figure 22. An O-trie for the Set of Keys of Figure 8 (# of Levels = 3, # of Branch Nodes = 5)

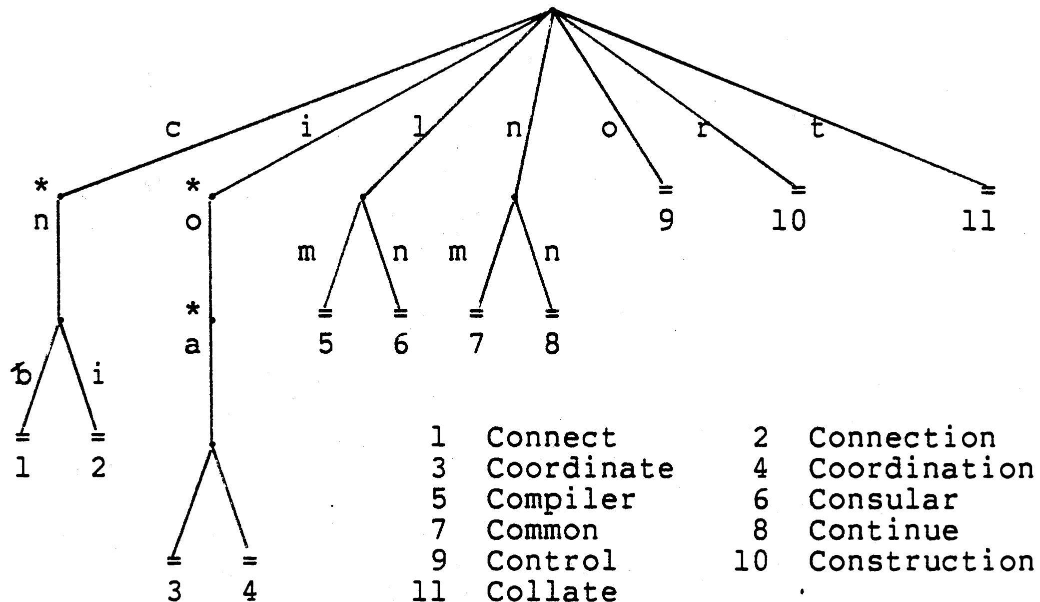


Figure 23. An Optimum Pruned Trie for the Set of Keys in Figure 8, with the Global Attribute Testing Order 6, 3, 8, 11(# of Levels = 5, # of Branch Nodes = 8)

Linked List Implementation

Since most of entries in the branch node tend to be empty, omitting these empty entries is highly desirable. Doubly chained trees, which are essentially linked list implementations, are proposed by Sussenguth (23) to save space at each node. In this linked list implementation, all sons of a node x are placed on a list, which corresponds to a branch node, with x pointing to the first entry. An entry

is added to a list only in the case when an attribute is present. It is clear that the number of entries in the linked list implementation is not fixed, but determined by the distribution of keys, so that storage corresponding to empty entries is saved. Figure 24 illustrates the linked list implementation of the trie shown in Figure 23, '$\wedge$' being the null link and '=' being the information node.

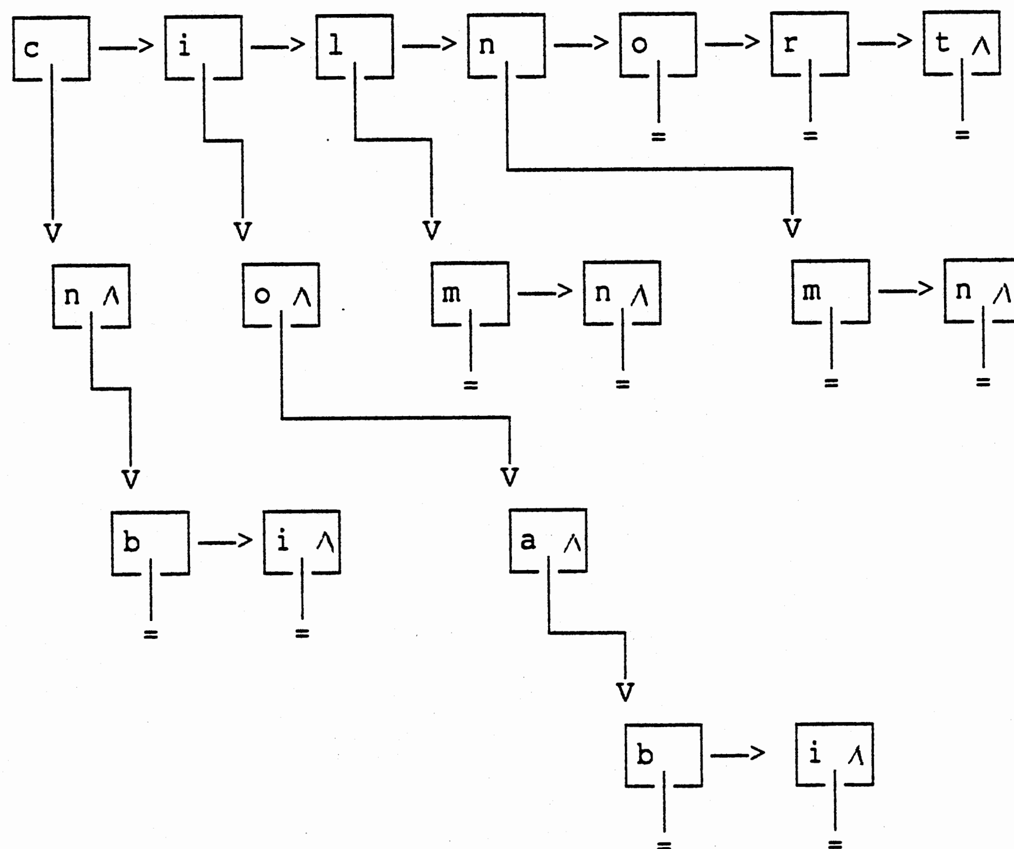


Figure 24. The Linked List Implementation of the Trie Shown in Figure 21

From Figure 24, it must be noticed that all the levels of the doubly chained tree take much less space than those of the trie of Figure 21. However, this advantage is at the expense of searching time, since branching is no longer determined by simply indexing the node array of pointers. Additionally, the time required to decide which path to follow at each linked list is no longer constant; nodes on the right-hand side requires a longer search than those on the left-hand side. D. Comer (9) summarized the heuristics proposed by Severence and Yao for a compromise about the space-time tradeoff of the doubly chained tree and the conventional trie structure, in which the first few levels are represented by branch nodes and the remaining levels by doubly chained trees.

C-tries (Compressed Tries)

The compressed trie or C-trie approach, presented by Maly (18), has the same underlying tree structure as a trie but can save a lot of space. The basic distinction between tries and C-tries is that, instead of storing explicit pointers in each branch node, C-tries utilize single bit fields facilitated by other information to locate the proper descendant at each node. This improvement in storage requirements is achieved at the cost of decreasing the flexibility of the structure. It implies that the main use of C-trie is for situations where index files are relatively

static, as will be seen in the subsequent illustration.

Similar to a trie structure, there are two types of nodes in a C-trie, ie., the internal branch node and the leaf node. A branch node N on level j in a C-trie has the structure of Figure 25(a). The field BRANCH-INDICATOR corresponds to pointer field of a trie. Retrieval of keys is made possible by referring to the fields BRANCH-INDICATOR, ADDRESS-OFFSET and the base address of the current level. Interpretation of each field is stated below:

NODE-TYPE: a one bit field
The internal branch node has the NODE-TYPE = 0

BRANCH-INDICATOR: a m-bit field
Each bit corresponds to a field of a trie with the first bit corresponding to a blank character. The k-th bit is set when one or more keys pass this node N and have their j-th attribute being the k-th character in the attribute set.

ADDRESS-OFFSET: a field less than or equal to log n bits where n is the number of keys. This field gives the number of nonzero bits of the BRANCH-INDICATOR fields in the nodes on level j to the left of node N.

In order to get the descendant address of the x-th field of BRANCH-INDICATOR on level j, both the number of 1-bits to the left of and including the x-th bit in BRANCH-INDICATOR field and the number in ADDRESS-OFFSET field of the given node be added to the base address of the nodes on level j. The base address of each level is computed after the C-trie structure is constructed. Hence, this descendant address calculation might become more clear after the construction of C-trie is illustrated.

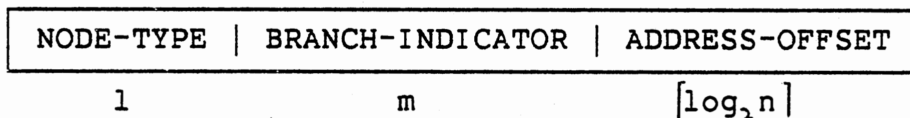
On the other hand, a leaf node in a C-trie may have the structure of Figure 25(b). Interpretation of each field is stated below:

NODE-TYPE - a one bit field
The leaf node has the NODE_TYPE = 1

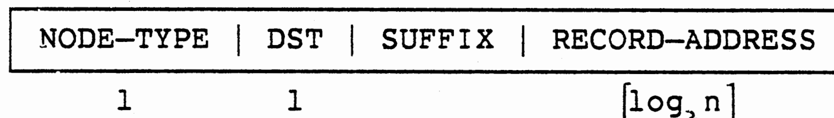
DST - a one bit field
DST = 0 if suffix can fit into SUFFIX
DST = 1 otherwise

SUFFIX - a multiple bits field depending on selection
This field contains the suffix x(j+1)...x(m) of the key x(1)....x(m) for those having DST = 0 or contains a pointer to a suffix table for those having DST = 1

RECORD-ADDRESS - a field less than or equal to $\lceil \log_2 n \rceil$ bits, where n is the number of keys. This field contains a pointer to the corresponding actual record residing in secondary storage



(a)



(b)

Figure 25. Structure of (a) an Internal Branch Node and (b) a Leaf Node in a C-trie

Of course, there is no restriction that a leaf node must have the same size as an internal branch node does. Hence, the size of the SUFFIX field can be chosen properly according the properties of the key set, so that the expense of both the space used to storing suffixes and the time spent to looking at a suffix table can be minimized as much as possible.

A C-trie for a given set of n keys can be constructed from an ordered list of keys, one level at a time, top to bottom. Each level j corresponds to the (j+1)-th attribute in all the 'unfinished' keys. A key is removed from the list when either its attribute currently processed is a blank or it becomes uniquely identified. The result of this construction is a sequence of nodes stored as a contiguous bit string. First is the root node, which is followed by all nodes on level 1 from left to right, followed by all nodes on level 2, etc. Each node can be packed into sequential words to form an addressible entity. The base address of the nodes on each level is one less than the address of the first node on the given level.

Evaluation of Trie Structures

The trie structure is a convenient way of indexing files in which a key consists of a number of attributes. A trie index is efficient in time if it is small enough to fit in primary memory. In this case, a trie index can be read in once and will be searched internally.thereafter. Since

branching at each node of a trie is simply by indexing an array of pointers, it is faster than other index structures. Furthermore, if there is a high probability of unsuccessful search, full tries with linked list implementations on the lower levels can be employed for this situation, because an unsuccessful search will work faster in the trie and the entire key can be checked in the trie index without externally searching the information node.

However, when a trie index is too large to fit in primary memory, that is, it must be kept in a secondary storage, the number of levels in a trie becomes a critical problem. In general, trie indexes require more levels and more nodes to represent a given set of keys than other multiway index structures, even though their nodes are usually much smaller than those of others. This fact implies that more accesses to the secondary storage might be required before a successful search can be reported, if a trie index is used. Forturnately, the size of a trie, namely, the number of levels as well as the number of branch nodes, can be dramatically reduced by selecting a proper order in which attributes are tested.

Choosing a global ordering of the attributes which produces a minimum size trie is difficult, especially when the key set is quite volatile. Although several heuristics have been presented to produce tries which are close to optimal in some sense, performance of them still inevitably degrades after significant number of insertions and

deletions have been done. In order to maintain good performance, updating attribute testing order according to the current occurrence of attributes in a trie after some period of time, might be desired. For this purpose, order-containing tries (O-tries) might be a good alternative. In order to determine when an O-trie needs to be partially reconstructed, it is useful to add a count field to each branch node. This count field will at all times indicate the total number of insertions and deletions made at all subtries with the given node being the root. Subtries could be reconstructed according to the new local attribute testing order which yields small subtries, as soon as the count field of their root node exceeds the predetermined limit. Therefore, local optimization, both in the number of levels and the number of nodes, can be always expected.

On the other hand, for large and relative static databases such as dictionaries, compress tries (C-tries) present a compact method of representation , yet facilitate reasonably fast searching. Since the fields in a C-trie are only one bit long, it can be expected that C-trie indexes are usually much smaller than other indexes and thus probably can fit in primary memory most of time. Hence, after the whole index is read in, all index searching can be done internally. External access to the information node is required only when the search of index is successful.

It can be concluded now that a trie index provides the following advantages compared to a prefix B^+-tree index, if

an appropriate way to represent the trie is chosen:

1. Shorter internal searching time
2. Greater ease of insertion and deletion
3. Greater convenience in handling arguments of varying lengths
4. Greater flexibility of key compression which is achieved by selecting attributes testing order.

However, the main disadvantages of trie indexes are those: (1) storage utilization is relative low and (2) number of levels is relative large. The latter is more significant when external searching to the trie index is required. Moreover, trie index is not in uniform depth, thus search along different path from the root to a leaf might require various number of accesses to a secondary storage, it is always undesirable. Since the efficiency of a trie reduces as the number of levels increases, it might be a good idea to index large databases using a trie for the first few levels and then switch to some other technique. Of course, if the whole trie index can fit in primary memory, O-trie and C-trie approaches are still good enough for dynamic and static databases respectively.

CHAPTER V

SUMMARY, CONCLUSIONS AND SUGGESTIONS FOR FURTHER RESEARCH

Among many existing techniques and structures used for indexing a database, prefix B^+-trees and trie structures as were presented in previous chapters, might be the best two candidates for indexing a textual database. A summary of their basic structure, advantages, disadvantages, possible refinements, possible usages and suggestions for further research follows.

Summary of Prefix B^+-tree Indexing

Basic Structure

A prefix B^+-tree is a variation of a standard B-tree. The B^+-tree structure implies that all actual keys reside in leaf nodes.

Key compression techniques are used. Shortest separators (rear compression) with common prefixes being pruned off (front compression) are stored in internal branch nodes to direct the search to leaf nodes. Since the length of separators is variable, the branching degree of each node is determined by its internal organization. It should be noted that pruned prefixes can be reconstructed while

traversing the tree during a search.

Advantages

Compared to tries, faster retrieval and less storage requirements generally result if the storage medium is a secondary storage device such as a disc. Besides good worst-case performance and good storage utilization of a B^+-tree are retained, the number of both levels and pages (nodes) required by the index part of a prefix B^+-tree are even less than those of a B^+-tree. Therefore, retrieval time, number of disc accesses and storage requirement are reduced by using prefix B^+-trees.

Predictable search performance results. The depth of a prefix B^+-tree is predictable and uniform. Hence, the search cost can be predicted and each request requires about the same time.

Disadvantages

The complexity of index building and maintenance algorithms of a prefix B^+-tree is increased significantly, so that the time required to execute the maintenance algorithms for a prefix B^+-tree is much more than the time required by a B^+-tree (50 -100 percent more). Additionally, internal searching time is increased due to the varying location of separators within a node.

Possible Refinement Schemes

Instead of splitting precisely in the middle when a node must be split, a split interval can be selected to decrease the length of the shortest separators and increase the branching degree of nodes, so that it tends to decrease the height of the index part and improve performance.

If the key set is relatively static, the common prefix can be kept within the same node as the separators and keys in order to simplify the search logic at the expense of slightly more storage space.

Possible Usages

Prefix B^+-trees are suitable for indexing large textual databases, in which key words are of varying length but are in clusters and the index has to be reside on secondary storage for external searching.

A prefix B^+-tree is suggested to be used in a relatively static environment because the advantage of less storage and fast retrieval can be earned by only building the index once without frequent maintenance. However, a simple prefix B^+-tree might be more cost effective than a prefix B^+-tree in a dynamic environment, because storing "shortest" separators in the branch node without pruning their common prefixes can save considerable amount of computing time required by some insertions and deletions which might occur very frequently in a dynamic environment.

Suggestions for Further Research

Bayer and Unterauer (3) argued that no satisfactory explanation can be made for the unexpectedly high computing time to execute the algorithms for a prefix B^+-tree. Tests can be performed to determine empirically, as well as analytically, the causes of the high computing time and alternatives to improving it.

In practical applications, sets of keys are often far from random and they tend to be in clusters with the identical leading letters. Choosing a suitable split point can be expected to reduce the length of the shortest separators. Of course, increasing the number of separators around the median key, which is considered for choosing a suitable split point results in shorter "shortest" separators and tends to decrease the height of the index part. However, the storage utilization might then be decreased since there can now be pages (nodes) less than half full. More empirical data can be obtained to find the effect of choosing different sizes of split interval.00680 .us Basic Structure;.sk 2 a The amount of the effect could be given in terms of height, the number of pages used and the average stroage utilization.

Summary of Trie Indexing

Basic Structure

A trie structure is a particular type of digital search

tree with the following characteristics: each internal branch node is an array of m pointer fields with components corresponding to digits or alphabetic characters, and the branching at any node in a trie structure is governed by constituent character(s) or digit(s) rather than entire keys. The key is not recorded in full in a trie until the first point where the key is uniquely identified.

Advantages

Compared to prefix B^+-tree, shorter internal searching time and greater ease of insertion and deletion result by using a trie index. Furthermore, an unsuccessful search might be faster in a trie index than in a prefix B^+-tree index because it can be detected before the leaf node is reached.

Disadvantages

Besides the main disadvantage of the relatively low storage utilization, a trie is usually of unbalanced structure as well as greater and unpredictable depth. The number of levels in a trie is determined by the distribution of the given key set and is usually larger than that in a prefix B^+-tree. Leaf nodes are not at the same level. The time required to locate a search argument is determined by the search path encountered, so that the search cost is not uniformly predictable.

Possible Refinement Schemes

The main disadvantage of a trie indeed is the relative inefficiency in using storage. The following variants were presented to improve this disadvantage:

1. Prune a trie to eliminate useless attributes.
2. Select the (global) attribute testing order to move the useless attributes to the end of the testing order where it will not be reached during a search.
3. Enter multiple keys into the same leaf node to limit the number of levels in a trie to some fixed number.
4. Include local attribute testing order in each branch node to indicate the optimum attribute testing order for various subtries - Order containing tries.
5. Use linked lists to implement branch nodes on the lower levels in a trie in order to save the storage corresponding to empty entries.
6. Compress each pointer field to a single bit field to save the storage - Compressed tries.

Possible Usages

Because of the shorter internal search time but the greater and unpredictable depth of a trie index, it is well suited for internal searching use without disc accesses. For instance, tables which are used to detect whether or not a search argument is a noise word or is equivalent to some other words, can be represented in trie indexes. Excellent performance may result not only because the tables are usually small enough to fit in primary memory and trie indexes can provide fast retrieval, but also because a high frequency of unsuccessful search might be encountered.

For large but relatively static databases, compressed tries present a compact way to represent them. If the whole C-trie index can be read in internal memory and all index search can be done internally thereafter, the compressed trie approach is suitable for this application.

Suggestions for Further Research

Performance of a trie is closely tied to the attribute testing order applied on it. Hence, how to obtain a highly desirable but computationally difficult attribute testing order for a given set of keys to yield a minimum-size trie could be an attractive research subject.

Conclusions and Suggested Further Work

Several approaches have been presented to improve the original B-tree index and trie index in order to achieve better performance. However, with all of these approaches, tradeoff situations arise concerning storage requirements and retrieval time or performance benefits and maintenance difficulties. In summary, prefix B^{+}-tree indexes have advantages of smaller storage requirements and fast retrieval from secondary storage but disadvantages of maintenance difficulties and much higher computing time, while trie indexes have advantages of very fast retrieval in primary memory and maintenance ease but disadvantages of inefficient storage utilization and improper characteristics for external searching.

Fortunately, one can confine all actual keys to leaf nodes. Hence, nonleaf nodes or internal branch nodes can then be filled with any kind of trees or any combination of trees which can lead the search path to the correct leaf node. Therefore, even though there is no way to eliminate all the disadvantages for any given index approach, selecting a most suitable approach among them, according to the practical use, can still optimize indexing performance.

There exists one possible scheme for large dynamic textual databases which utilizes the trie approach for the first few levels of an index and prefix B^+-tree approach for the remaining levels. The reasons behind this are (1) the most frequently accessed upper level trie nodes save internal searching time and tend to break up the sets of records rapidly, and (2) the lower level prefix B^+-tree structures gurantee the uniform depth of index and present a quite compact way to represent the index. Moreover, prefix reconstruction might be simplified to just concatenate characters which are encountered in the upper trie structure along the search path so that there is no need to rearrange separators due to the change of the length of common prefixes.

BIBLIOGRAPHY

(1) Appenzeller, T. "Evolution in CALR Systems." National Online Meeting Proceedings, New York (March, 1981), 37-40.

(2) Bayer, R. and McCreight, E. "Organization and Maintainance of Large Order Indexes." Acta Informatica, Vol. 1 (1972), 173-189.

(3) Bayer, R. and Unterauer, K. "Prefix B-tree." ACM Transactions on Database Systems, Vol. 2 (March, 1977), 11-16.

(4) Benson, D.A. "A Microprocessor-based System for the Delivery of Full-text, Encyclopedic Information." Proceedings of the 44th ASIS Annual Meeting, Vol. 18 (1981), 175-184.

(5) Chang, H.K. "Compressed Indexing Method." IBM Technical Disclosure Bulletine, Vol. 11, No. 11 (April, 1969), 1400.

(6) Christian, D.D. "A B-tree Index Approach to storing and Retrieving Records on Direct Access Auxiliary Storage." (Unpub. M.S. thesis, Oklahoma State University, 1977.)

(7) Comer, D. "The Ubiquitous B-tree." Computing Surveys, Vol. 11, No. 2 (June, 1979), 121-137.

(8) Comer, K. "Heuristics for Trie Index Minimization." ACM Transactions on Database Systems, Vol. 4 (1979), 383-395.

(9) Comer, D. and Sethi, R. "The Complexity of Trie Index Construction." J. Assn. Computing Machinery, Vol. 2 (July, 1977), 428-440.

(10) Fredkin, E. "Trie Memory." Communications ACM, Vol. 3 (Sept., 1960), 490-499.

(11) Grimson, J.B., and Stacey, G.M. "A Performance Study of Some Directory Structures for Large Files." Information Storage and Retrieval, Vol. 10 (1974), 357-364.

(12) Held, G. and Stonebraker, M. "B_tree re-examined." Communications ACM, Vol. 21 (Feb., 1978), 139-143.

(13) Hollar, L.H., and Stellhorn, W.H. "A Specialized Architecture for Textual Information Retrieval." Proc. Nat. Computer Conference, (1977), 697-702.

(14) Horowitz, E. and Sahni, S. Fundamentals of Data Structure. Computer Science Press Inc., Potomac, Maryland, 1976.

(15) Knapp, P.E. "Implementation of a Generalized Access Path Structure." (Unpub. M.S. thesis, Oklahoma State University, 1981.)

(16) Knuth, D.E. The Art of Computer Programming Vol. 3: Sorting and Searching. Addison Wesley Publ. Co., Reading, Mass., 1973.

(17) Lefkovits, D. File Structure for On-Line Systems. Spartan Books, New York, Washington, (1969).

(18) Maly, K. "Compressed Trie." Communications ACM, Vol. 19 (July, 1976), 409-415.

(19) Morrison D. "PATRICIA - Practical Algorithm To Retrieve Information Coded in Alphanumeric." J. ACM, Vol. 15, No. 4 (Oct, 1968), 514-534.

(20) Salton, G. Automatic Information and Retrieval. McGraw-Hill Book Company, New York, (1968).

(21) Salton, G. The SMART Retrieval System - Experiments in Automatic Document Processing. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, (1971).

(22) Siler, K.F. "A Stochastic Evaluation Model for Database Organization in Data Retrieval System." Communication ACM, Vol. 19 (Feb., 1976), 84-95.

(23) Sussenguth, E.H. JR. "Use of Tree Structures for Processing files." Communications ACM, Vol. 5 (May, 1963), 272-279.

(24) Wagner, R.E. "Indexing Design Considerations." IBM Syst. J., Vol. 12, No. 4 (1973), 351-367.

(25) Webster, R.E. "B^+-tree." (unpub. M.S. report, Oklahoma State University, 1980.)

(26) Williams, P.W., and Khallaghi, M.T. "Document Retrieval Using a Substring Index." Computer J., Vol. 20 (August, 1977), 257-262.

VITA 2

AN-LEE ANNE FENG

Candidate for the Degree of

Master of Science

Thesis: A STUDY OF TWO COMPETING INDEX MECHANISMS: PREFIX B^+-TREES AND TRIE STRUCTURES

Major Field: Computing and Information Science

Biographical:

Personal Data: Born in Tainan, Taiwan, Republic of China, October 25, 1949, the daughter of Mr. and Mrs. K. C. Feng.

Education: Graduated from Taipei Municipal First High Girls' School, Taipei, Taiwan, Republic of China, in June, 1968; received Bachelor of Science in Agriculture degree from National Taiwan University, Taipei, Taiwan, Republic of China, in June, 1972; completed requirements for the Master of Science degree at Oklahoma State University in December, 1982.

Professional Experience: Programmer, China Data Processing Center, Taipei, Taiwan, Republic of China, June, 1979 - December, 1979; programmer, Time Management Software, Stillwater, Oklahoma, Jan, 1982 - Aug, 1982; graduate teaching assistant, Department of Computing and Information Science, Oklahoma State University, Stillwater, Oklahoma, September, 1980 - September, 1982; programmer, Starduster Games, Norman, Oklahoma, October, 1982 - present.