THE UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

THE COMPLEAT PATRICIA

A DISSERTATION

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

degree of

DOCTOR OF PHILOSOPHY

By

CARTER BAYS

Norman, Oklahoma

1974

Downloaded from SHAREOK. Persistent Handle URL: https://hdl.handle.net/11244/3821

THE COMPLEAT PATRICIA

APPROVED BY

John B. Thompson

Stephen Tegard

Vytas B. Gyslys

James A. Payne

DISSERTATION COMMITTEE

(Each copy of the dissertation must bear the signatures of all members of the final oral examination committee.)

ACKNOWLEDGEMENTS

I wish to thank the members of my committee for their helpful and constructive comments, particularly as this dissertation was reaching its final stages. Also, I wish to thank Lisa Tadlock for her typing assistance during the early stages of the project, and Marie Reubiez for her beautiful final copy. Intangible thanks go to Caroline, Ginny, Janice Sue, JoAn, and Shirley.

TABLE OF CONTENTS

	1.0	INTRODUCTION	PAGE 1
	1.1	TERMINOLOGY	PAGE 4
	1.2	SEARCHING FOR KEYS	PAGE 6
	1.3	EXAMPLE OF A STRUCTURE DESIGNED FOR FAST KEY RETRIEVAL	PAGE 7
	1.3.1	SOME CHARACTERISTICS OF THE ALGORITHMS IN THIS DISSERTATION	PAGE 9
	1.3.2	ALGORITHM: SEARCH FOR A KEY IN A TRIE MEMORY	PAGE 9
	1.4	THE DIGITAL TREE	PAGE 10
	1.4.1	ALGORITHM: INSERT - SEARCH FOR A NODE IN A DIGITAL TREE	PAGE 12
	2.0	THE ESSENTIALS OF PATRICIA	PAGE 15
##	2.1	ALGORITHM: SEARCH FOR A NODE IN A PATRICIA TREE	PAGE 25
	2.1.1	COMMENTS ON ALGORITHM 2.1	PAGE 26
#	2.2	ALGORITHM: INSERT A NEW NODE INTO A PATRICIA TREE	PAGE 27
	2.2.1	COMMENTS ON ALGORITHM 2.2	PAGE 28
#	2.3	ALGORITHM: LIST ALL MATCHES IN A PATRICIA TREE	PAGE 29
	2.4	HOW WELL BALANCED IS A PATRICIA TREE?	PAGE 30
	2.5	CONCLUSIONS - SUMMARY OF CHAPTER TWO	PAGE 31
	3.0	ALTERING THE PATRICIA TREE - NODE DELETION	PAGE 32
#	3.1	ALGORITHM: DELETE A PATRICIA NODE FROM THE TREE	PAGE 38
	3.2	DETERMINING TD, FT, TT	PAGE 52
	3.2.1	ALGORITHM: FIND PD, TD, FT, TT	PAGE 54
	3.3	SUBTREE PRUNING - DELETION OF PREFIXES	PAGE 54
	3.3.1	PREPARING THE STRUCTURE FOR SUBTREE DELETION	PAGE 59
	3.3.2	ALGORITHM: PREPARE THE STRUCTURE FOR SUBTREE DELETION	PAGE 60
	3.4	CONCLUSION - DELETING NODES	PAGE 61
	4.0	DELETION OF TEXT MATERIAL	PAGE 62
	4.1	DELETING CONTIGUOUS TEXT	PAGE 63
	4.1.1	CONCEPTS BEHIND DELETING CONTIGUOUS TEXT	PAGE 68
#	4.1.2	ALGORITHM: DELETE CONTIGUOUS TEXT	PAGE 72
	4.1.3	ALGORITHM: DELETE A SPECIFIC KEY FROM THE TEXT	PAGE 75
	4.2	ALGORITHM: INSERT TEXT	PAGE 75
	4.3	CONCLUSIONS - ALTERING TEXT	PAGE 76
	5.0	ALTERNATE METHODS FOR REPRESENTING THE PATRICIA TREE	PAGE 77
	5.1	A RIGHT THREADED PATRICIA TREE	PAGE 77
##	5.1.1	ALGORITHM: CREATE A RIGHT THREADED PATRICIA TREE	PAGE 84
	5.1.2	AN IMPORTANT AND IMMEDIATE CONSEQUENCE - ELIMINATING LTAG	PAGE 85
	5.2	PREORDER SEQUENTIAL REPRESENTATION	PAGE 86
	5.3	PREORDER SEQUENTIAL PATRICIA	PAGE 88
#	5.3.1	ALGORITHM: TRAVERSE A PREORDER SEQUENTIAL STRUCTURE IN POSTORDER	PAGE 88
	5.3.2	CAN THE STRUCTURE BE UTILIZED?	PAGE 91
	5.3.3	ALGORITHM: SEARCH THE PREORDER SEQUENTIAL STRUCTURE FOR A KEY	PAGE 91

5.3.4 HOW TO HANDLE MODIFICATIONS PAGE 92
5.3.5 ALGORITHM: CONVERT A RIGHT THREADED TREE TO PREORDER SEQUENTIAL PAGE 93
5.3.6 CONVERTING OVER THE SAME MEMORY SPACE PAGE 93
5.3.7 ALGORITHM: CONVERT TO PREORDER SEQUENTIAL FORM OVER THE SAME SPACE PAGE 94
5.4 A SLIGHTLY DIFFERENT VERSION OF PREORDER SEQUENTIAL REPRESENTATION PAGE 101
5.4.1 ALGORITHM: SEARCH THROUGH STRUCTURE WITH COMBINED RLINK-LTAG PAGE 101
5.4.2 ALGORITHM: POSTORDER TRAVERSAL OF STRUCTURE WITH COMBINED RLINK-LTAG PAGE 101
5.4.3 ALGORITHM: RIGHT THREADED TO PREORDER SEQUENTIAL COMBINING RLINK-LTAG PAGE 102
5.5 FURTHER COMPRESSION PAGE 103
5.6 CONCLUSIONS - ADVANTAGES OF THE COMPRESSED FORM PAGE 105
6.0 PRACTICAL APPLICATIONS PAGE 106
6.1 A HYPOTHETICAL MEDIUM-SCALE SYSTEM PAGE 106
6.1.1 BOOLEAN OPERATIONS PAGE 108
6.2 ANOTHER APPLICATION - CALL FOR ACTION PAGE 109
6.3 SOME USEFUL TRICKS INVOLVING COMPARISON STRINGS PAGE 109
A.0 APPENDIX - OTHER ALGORITHMS PAGE 114
A.1 ALGORITHM: GET A NODE FROM AN AVAILABLE LIST PAGE 114
A.1.1 ALGORITHM: RETURN A NODE TO AN AVALIABLE LIST PAGE 114
A.2 ALGORITHMS FOR A DOUBLY LINKED AVAILABLE LIST PAGE 115
A.2.1 ALGORITHM: GET A NODE FROM A DOUBLY LINKED AVAILABLE LIST PAGE 115
A.2.2 ALGORITHM: RETURN A NODE TO A DOUBLY LINKED AVAILABLE LIST PAGE 116
A.3 ALGORITHM: PRINT A TREE PAGE 116
B.0 APPENDIX - THE TEST PROGRAM PAGE 119
B.1 THE FACILITIES OF THE PROGRAM PAGE 119
B.1.1 READ TEXT AND CREATE A TREE PAGE 119
B.1.2 DISPLAY THE TREE AND/OR THE TEXT PAGE 120
B.1.3 SEARCH FOR A KEY AND LIST ALL ITS MATCHES PAGE 121
B.1.4 DELETE A NODE FROM THE TREE PAGE 121
B.1.5 DELETE OR INSERT TEXT PAGE 121
B.1.6 CONVERT TO PREORDER SEQUENTIAL FORM PAGE 122
B.2 SAMPLE OUTPUT PAGE 123
C.0 APPENDIX - TIMING CONSIDERATIONS PAGE 142
C.1 TIMING FOR THE SEARCH LOOP OF ALGORITHM 2.1 PAGE 142
C.2 TIMING FOR A TRAVERSAL ITERATION PAGE 142
C.3 TIMING FOR OTHER ALGORITHMS PAGE 145
D.0 APPENDIX - THEOREMS AND PROOFS PAGE 147
E.0 APPENDIX - GLOSSARY PAGE 152
BIBLIOGRAPHY PAGE 154
INDEX PAGE 155

LIST OF ILLUSTRATIONS

FIGURE C-1 TIMING FOR THE LOOP IN ALGORITHM 2.1 PAGE 144
FIGURE C-2 TIMING FOR A TRAVERSAL LOOP USING ALGORITHM 2.3 PAGE 145
FIGURE C-3 TIMING OF MOST PATRICIA ALGORITHMS PAGE 146
FIGURE 1-1 A SAMPLE TRIE MEMORY PAGE 8
FIGURE 1-2 A SORTED BINARY TREE PAGE 11
FIGURE 1-3 A DIGITAL TREE PAGE 14
FIGURE 1-4 THE NUMBERS INSERTED IN DESCENDING ORDER PAGE 14
FIGURE 2-1 A SIMPLE PATRICIA TREE PAGE 16
FIGURE 2-2 IF WE ADD A KEY TO THE TREE OF FIGURE 2-1 THE TREE IS CHANGED PAGE 18
FIGURE 2-3 THE ACTUAL REPRESENTATION OF THE PATRICIA TREE OF FIGURE 2-1 PAGE 21
FIGURE 2-4 A PATRICIA TREE BUILT BY ALGORITHMS 2.2 AND 2.1 PAGE 23
FIGURE 2-5 THE BIT COMPARISONS THAT ARE ACTUALLY MADE PAGE 24
FIGURE 3-1 NOTATION THAT WILL BE USED THROUGHOUT THE DISSERTATION PAGE 33
FIGURE 3-2 TYPE 1 NODE STRUCTURE. BACKWARD POINTERS ARE INDICATED PAGE 34
FIGURE 3-3 TYPE 2 NODE STRUCTURE PAGE 35
FIGURE 3-4 TYPE 2A NODE STRUCTURE PAGE 36
FIGURE 3-5 TYPE 2B PAGE 39
FIGURE 3-6 TYPE 2C PAGE 40
FIGURE 3-7 TYPE 2D PAGE 41
FIGURE 3-8 TYPE 2E PAGE 42
FIGURE 3-9 TYPE 2F PAGE 43
FIGURE 3-10 GENERAL CONFIGURATION FOR A SUBTREE DELETION PAGE 56
FIGURE 3-11 SPECIAL CASE WHERE TFR=TA PAGE 57
FIGURE 3-12 SUBTREE DELETION CORRESPONDS TO A TYPE 2 DELETION PAGE 58
FIGURE 4-1 A STRANGE TREE PAGE 64
FIGURE 4-2 THE SPACE HAS BEEN ELIMINATED FROM THE TEXT OF FIGURE 4-1 PAGE 65
FIGURE 4-3 THE FOURTH A HAS BEEN ELIMINATED FROM THE TEXT PAGE 66
FIGURE 4-4 PREORDER AND ENDORDER VISITS PAGE 71
FIGURE 5-1 AN EXAMPLE OF A RIGHT THREADED PATRICIA TREE PAGE 78
FIGURE 5-2 THE TREE OF FIGURE 5-1 AS BUILT BY ALGORITHM 2.2 PAGE 80
FIGURE 5-3 IF WE USE ALGORITHM 2.2 AND INSERT THE KEYS IN REVERSE ORDER PAGE 81
FIGURE 5-4 A BINARY TREE PAGE 87
FIGURE 5-5 THE PREORDER SEQUENTIAL REPRESENTATION PAGE 89
FIGURE 5-6 PASS 1 CREATES THIS FROM THE STRUCTURE OF FIGURE 5-1 PAGE 97
FIGURE 5-7 THE SPECIAL SITUATION WHERE LLINK(J) = I PAGE 98
FIGURE 5-8 THE IMPORTANT LINK FIELDS DURING STEP 2 OF ALGORITHM 5.3.7 PAGE 99
FIGURE 5-9 AFTER STEP 2 OF ALGORITHM 5.3.7 PAGE 100
FIGURE 5-10 PREORDER SEQUENTIAL FORM WITH LTAG AND RLINK COMBINED PAGE 104
FIGURE 6-1 SOME TEXT OF THE CALL FOR ACTION FILES PAGE 110
FIGURE 6-2 SOME OF THE OUTPUT PRODUCED FOR CALL FOR ACTION PAGE 111

TABLE OF ALGORITHMS

5.3.5	ALGORITHM: CONVERT A RIGHT THREADED TREE TO PREORDER SEQUENTIAL	PAGE 93
#5.3.7	ALGORITHM: CONVERT TO PREORDER SEQUENTIAL FORM OVER THE SAME SPACE	PAGE 94
##5.1.1	ALGORITHM: CREATE A RIGHT THREADED PATRICIA TREE	PAGE 84
#3.1	ALGORITHM: DELETE A PATRICIA NODE FROM THE TREE	PAGE 38
4.1.3	ALGORITHM: DELETE A SPECIFIC KEY FROM THE TEXT	PAGE 75
#4.1.2	ALGORITHM: DELETE CONTIGUOUS TEXT	PAGE 72
3.2.1	ALGORITHM: FIND PD, TD, FT, TT	PAGE 54
A.2.1	ALGORITHM: GET A NODE FROM A DOUBLY LINKED AVALIABLE LIST	PAGE 115
A.1	ALGORITHM: GET A NODE FROM AN AVAILABLE LIST	PAGE 114
1.4.1	ALGORITHM: INSERT - SEARCH FOR A NODE IN A DIGITAL TREE	PAGE 12
#2.2	ALGORITHM: INSERT A NEW NODE INTO A PATRICIA TREE	PAGE 27
4.2	ALGORITHM: INSERT TEXT	PAGE 75
#2.3	ALGORITHM: LIST ALL MATCHES IN A PATRICIA TREE	PAGE 29
5.4.2	ALGORITHM: POSTORDER TRAVERSAL OF STRUCTURE WITH COMBINED RLINK-LTAG	PAGE 101
3.3.2	ALGORITHM: PREPARE THE STRUCTURE FOR SUBTREE DELETION	PAGE 60
A.3	ALGORITHM: PRINT A TREE	PAGE 116
A.2.2	ALGORITHM: RETURN A NODE TO A DOUBLY LINKED AVAILABLE LIST	PAGE 116
A.1.1	ALGORITHM: RETURN A NODE TO AN AVAILABLE LIST	PAGE 114
1.3.2	ALGORITHM: SEARCH FOR A KEY IN A TRIE MEMORY	PAGE 9
##2.1	ALGORITHM: SEARCH FOR A NODE IN A PATRICIA TREE	PAGE 25
5.3.3	ALGORITHM: SEARCH THE PREORDER SEQUENTIAL STRUCTURE FOR A KEY	PAGE 91
5.4.1	ALGORITHM: SEARCH THROUGH STRUCTURE WITH COMBINED RLINK-LTAG	PAGE 101
5.4.3	ALGORITHM: TRANSFORM RIGHT THREADED TO PREORDER SEQUENTIAL COMBINING RLINK-LTAG	PAGE 102
#5.3.1	ALGORITHM: TRAVERSE A PREORDER SEQUENTIAL STRUCTURE IN POSTORDER	PAGE 88

1.0 Introduction

The effort to use computers to store and manage large amounts of textual data continues to be one of the most formidable tasks confronting the computer scientist, for it seems that information is being generated at an ever increasing rate, almost in defiance of the attempt to contain it. Part of the problem deals with human creativity and human articulation, which cannot be computerized--at least not yet. Hence, we are concerned here with that portion of the problem that involves the management of information and its rapid and efficient retrieval. In essence, once information has been generated, we would like to know how to structure it within the computer so that:

a) It is classified in some hierarchial or alphabetical manner.

b) It is stored as efficiently as possible within the confines of the computer.

c) We can rapidly retrieve anything we want, with a minimal amount of noise in the form of excess information.

d) We are free as much as possible from the restrictions imposed upon us by computer manufacturers or--and more important--inefficiently written programs.

It is toward this end that the author has chosen to further develop PATRICIA[1], whose underlying concepts were discovered in 1968 by Dr. Donald Morrison at Sandia Laboratories. PATRICIA is not "just another information retrieval system." Rather, PATRICIA can be thought of as a "concept"

---

[1]PATRICIA is an acronym for "Practical Algorithm To Retrieve Information Coded in Alphanumeric." PATRICIA is also the first name of Dr. Morrison's wife.

which gives us a most natural and flexible means of classifying written matter within a computer. The underlying idea involves a binary tree search where individual bits of key words or phrases are used in determining the search path. But PATRICIA goes further than the ordinary binary search. The only bits that are looked at are those pertinent in determining the search path. If a key word or phrase is identical to the key being compared, all identical bits in the two keys are totally ignored. Thus, if two sentences differ only in the last letter, then in comparing these two sentences, PATRICIA would not bother with anything except the last letter of each sentence, ignoring all the preceding letters.

PATRICIA can search for key words or phrases of arbitrary length and almost arbitrary format. It defines a structure that automatically orders information both hierarchically and alphabetically. Moreover, the actual information is not rearranged or altered in any way, and yet the search for a desired piece of information is accomplished very quickly. As a practical example, consider the problem of locating an individual income tax return that exists somewhere on file containing 100,000,000 other tax returns. PATRICIA could find it in less than one second on a medium sized 360 system. Moreover, only two or three accesses to the file would be required.

So why are there not more computerized retrieval systems using PATRICIA? The apparent problems are given below.

1) It is quite difficult to alter the complex PATRICIA data structures without rebuilding them, and the rebuilding process is of necessity quite lengthy.

2) Space restrictions are imposed upon PATRICIA by its data structure.

3) There exists very little published material on PATRICIA besides the work of Knuth and Morrison.

This dissertation attempts to solve all three of these problems. The first problem is solved by including a group of heretofore unpublished algorithms that will perform any type of structural alteration that a practical application might require. These algorithms allow us to alter both the PATRICIA structure and the textual information from which the structure is built. The space restriction of the second problem is alleviated by showing how we can reduce the structure to about 40% of the size of currently used PATRICIA structures. The third problem is attacked by presenting a complete and logically organized set of algorithms. Performance is evaluated and operating times are given for a typical medium scale computer system.

The following sections of this chapter are concerned with structures that have certain characteristics in common with PATRICIA, but are more elementary. A knowledge of these structures will be of assistance in understanding the more complex PATRICIA structure.

Readers already familiar with PATRICIA may wish to skip to Chapter Two, which gives the essential algorithms required to build a PATRICIA structure, search for a particular entry, and list out all occurrences of the entry.

Chapters Three, Four, and Five constitute the heart of the dissertation, with Chapter Three being the first chapter dealing with unpublished algorithms. The important PATRICIA node deletion algorithms are thoroughly described, and their use is extended to deletions of entire substructures within a PATRICIA structure.

Chapter Four then uses the algorithms developed in Chapter Three as part of a package that allows portions of an existing body of information to be altered. At this point the set of algorithms is complete

from the standpoint of any user who might be working with material that is frequently altered.

In Chapter Five we propose and implement a standard structure for PATRICIA, which then leads itself to further reduction. In fact, four of the six individual elements of the PATRICIA data structure are eliminated, resulting in a great saving of space.

Chapter Six gives several practical examples, including the example afforded by the table of contents and index to this dissertation. Moreover, some additional dirty tricks are discussed which could be employed in the campaign to optimize further the PATRICIA structure.

Several appendices are provided. For quick reference Appendix E reproduces the description of the data structure terms given in the next section. Appendix A contains some peripheral algorithms, and Appendix B explains how the test program operates. (The test program checks all the algorithms of chapters two through five.) Appendix C gives timing estimates for most of the algorithms and shows specific times for a 360/50. Appendix D develops and proves several important theorems, and proves the validity of some of the original algorithms.

1.1. Terminology

This section describes terminology that is used throughout the dissertation. Most terms are commonly encountered in the study of Data Structures, and are described further in Knuth (1968) Chapter Two. For convenience this section is reproduced in Appendix E.

AVAILABLE LIST--A list of empty nodes. (A process which requires space for a new node can always get one by picking the top or bottom node from an available list.)

AVAIL LIST--Identical to an available list.

ANCESTOR--Within a tree, an ancestor of node X is on a path between node X and the root of the tree.

BACKWARD POINTER--A link field in a PATRICIA node, X, that points to X or to some ancestor of X.

BINARY TREE--A data structure in which each node has no more than two nodes hanging from it. These two nodes are commonly called "ROOTS of LEFT and RIGHT SUBTREES."

EBCDIC--A specific internal code where 8 bits represent one character within the computer. For example, the EBCDIC value of of "A" is binary "11000001."

ENDORDER TRAVERSAL--A method of looking at all the nodes of a binary tree in which we first look at all the nodes in the left subtree of a node, then all the nodes in the right subtree of the node, and finally, the node itself. Each node is "looked at" exactly once, although the algorithm for effecting an endorder traversal may actually pass by the node more than once.

FIELD--The smallest entity of information contained in a node. A field may be one or more binary bits in size.

KEY--A contiguous string of characters constituting a word or phrase that we wish to search for and, hence, use in some comparison scheme.

LAMBDA ("λ")--See NULL POINTER.

LEFT LINK--In a binary tree, the link field pointing to the left subtree of the node.

LINK--The specific field of a node that points to the next node in a list. (Actually, a given node can point to more than one node: for example, a node in a binary tree can point to two other nodes.)

LIST--A series of nodes which are physically stored at random, but which have an order that is specified by the LINK fields.

NODE--An entity of information. It will consist of one or more fields. (An example--a node could be likened to a single library catalogue card, and a field to an individual entry on the card, such as the author's name.)

NULL POINTER--(Sometimes called "LAMBDA" or "λ"). A specifically valued link field that indicates the last node in a list. When any link field points to no other node, it is given a value called "λ" (frequently zero). We sometimes say that this link "points to λ."

POINTER--Has the same function as a link, except sometimes a pointer is not contained in any node.

POSTORDER TRAVERSAL--A method of looking at all the nodes of a binary tree in which we first look at all the nodes in the left subtree of a given node; then we look at the node; then we look at all the nodes in the right subtree of the node.

PREORDER TRAVERSAL--Still another method of looking at all the nodes in a binary tree, in which we first look at the node, then the nodes in its left subtree, and finally the nodes in its right subtree.

RIGHT LINK--In a binary tree, the link field pointing to the left subtree of the node.

RIGHT THREADED BINARY TREE--A binary tree in which the right links of terminal nodes point to the next node that would be visited if we were traversing the tree in postorder.

ROOT--In a tree, the node from which all other nodes hang. (Thus, computer trees are usually upsidedown.)

SUBTREE--A branch of a tree. Pick any node in a tree--it is the root of a subtree.

TERMINAL NODE--A node in a binary tree that has no left and/or right subtree. In a PATRICIA tree, the affected right or left link then becomes a backward pointer.

THREAD--The same as a backward pointer.

VISIT--A term for what we do when we "look at" a node during a preorder, postorder, or endorder traversal. Usually a visit involves performing an algorithm, or printing out information.

1.2 Searching for Keys

In every information retrieval system, the main concern after we have stored the information is how to get it back out. Usually we will only be interested in a small fraction of the total amount of information stored, such as a particular student's record in a university student information system, or a list of constituents in voting district three, or all articles about "Computers and Chemistry" in a library. Moreover, in many situations the speed of retrieval is quite important--such as in an airline reservation system. In all cases we may reduce the problem to: "What is the best way of finding all keys that match a given key?"

A "key" as used in this dissertation could be a student name, a social security number, a subject topic--in fact any contiguous string of symbols that is supposed to occur one or more times within the main body of information.

For background purposes let us examine two methods of searching for matches to keys; these methods are predecessors to PATRICIA and are in widespread use today.

1.3. Example of a Structure that is Designed for Fast Key Retrieval

This data structure, which was described by Fredkin (1960), looks at every letter of a key, starting from the left, until it can be determined where the key is located in the main body of information. The structure commonly goes by the name "TRIE" memory (where "TRIE" apparently refers to reTRIEval) and should not be confused with a "tree," which is an entirely different structure.

"TRIE" memory is laid out in table form as shown in figure 1-1. Each vertical column corresponds to a "node" (in later sections, a node will refer to a much smaller entity--a "node" in a binary tree). The scheme used to look for a key is rather simple. Let us assume we are searching through the TRIE memory of figure 1-1 for the key "THEN." Initially, we go to the "T" row of the first node. The "3" means that we continue our search by going to node 3 and looking at the second letter, "H." The "H" row of node 3 contains a value of "4", which means that we go to the 4th node when we look at the 3rd letter, which is an "E." Similiarly, we arrive at the 5th node and go to the appropriate row, which is "N." The entry "(THEN)" means that we have found a key, and much now check to see if it is the correct one. If it is not, our key

		(A)			(THE)
A	2		(TAR)		
B					
C					
D					
E				5	
F					
G					
H			4		
I			(TIP)		
J					
K					
L					
M					
N		(AN)			(THEN)
O			(TOP)		
P					
Q					
R					
S					
T	3				
U					

Figure 1-1. A sample TRIE memory. Each vertical column corresponds to a node. The keys inserted were, in this order: THE, AN, A, TAR, THEN, TIP, TOP.

is not in the TRIE memory.

The above example illustrates the type of search that will be under discussion throughout the dissertation. The search involves looking at the first letter (or digit) then going on to the next letter, repeating the process until a match to the key is found.

1.3.1 Some Characteristics of the Algorithms in this Dissertation

The informal discussion just given should aid in the analysis of the more formal algorithm given below, which is typical of those throughout this dissertation and closely follows the style of Knuth. For the most part, certain variable naming conventions have been followed. P, Q, and R always refer to pointer variables, as do (usually) I, J, X, Y, and Z. The letter "K" frequently refers to a character string or key which we are searching for. Individual fields of nodes are always given variable names with at least three letters. In later chapters, we will introduce "ATOP" which points to the top node of an available list. Algorithms with one or more "#" in the margin next to their number merit special attention.

1.3.2 Algorithm: Search for a Key in a TRIE Memory (Knuth 1973)

In a TRIE alphabet, we allow N characters--normally all the letters, digits, and special symbols. Assume that if the characters are coded in, say, EBCDIC code, they will be translated so that they have integer values between 1 and N. Let each node consist of a vector of N subnodes (a subnode is a single rectangle in figure 1-1). Each subnode has two fields: a PTR field and a one bit TAG field. The PTR field can either be empty (indicated by PTR=0) or it can point to another node (indicated by TAG=1)

or it can point to the target key (indicated by TAG=0). For simplicity a key is assumed to be a single word, terminated by a blank. In the following algorithm, P is a node pointer and Q is a subnode pointer. Note that 1 < Q < N.

Input: Key we are searching for.

Output: Location of the matching key.

1) Set P ← 1 (P points initially to the first node), I ← 0, K ← key we are searching for, (K will be followed by a blank.)

2) Set I ←I+1, Q ←Ith character of K. (Initially, look in the first node at the Qth row. If Q=blank, then K has been completely scanned, and we will end up in step 3 or step 5.)

3) If PTR (P,Q)=0 (i.e., the subnode is empty), then K is not in the TRIE memory. (We may insert K by setting PTR (P,Q) ← the location of K, TAG(P,Q) ← 0)

4) If TAG (P,Q)=1 (i.e., the subnode points to another node), set P ←PTR(P,Q). Go to step 2. (Go to the proper node and compare to the next character in the key.)

5) We know that PTR (P,Q) points to a key. If we are sure that K is in the table, then PTR (P,Q) gives its location; otherwise, compare K with the key at PTR (P,Q).

END (End of Algorithm)

Although this example is inefficient insofar as five nodes are required for seven keys, most TRIE memories are much more efficiently organized, since the subnodes fill up as more keys are introduced. (See Knuth (1973) page 482 where 12 nodes are used to represent 31 keys.)

1.4 The Digital Tree

The digital tree is somewhat similar in concept to the ordinary lexicographically ordered binary tree, an example of which is shown in figure 1-2.

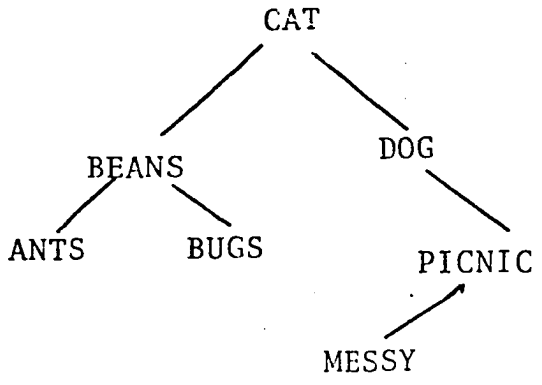


Figure 1-2. A lexicographically ordered binary tree. Since "BEANS" is in the left subtree of "CAT", we know that "BEANS" alphabetically precedes "CAT". Since "MESSY" is in the right subtree of "CAT", we know that "MESSY" alphabetically follows "CAT". A postorder traversal lists the keys in alphabetic order.

The difference is that the key insertion process is based on whether a particular bit position is a binary "1" or "0", rather than whether one key is alphabetically "less" or "greater" than another key. Bits are scanned one at a time to determine the position that a key is to occupy in a Digital tree. The advantage this tree has over the tree in figure 1-2 is that the Digital tree is relatively balanced, regardless of the order of insertion. (It is well known that a lexicographically sorted tree can be very badly out of balance, depending upon the order of key insertion. Note what happens when we build the tree of figure 1-2 by inserting the keys in the order: ANTS, BEANS, BUGS, CAT, DOG, MESSY, PICNIC.)

1.4.1 Algorithm: Insert--Search for a Node in a Digital Tree

Let KEY be the key we are inserting or are looking for. Let each tree node be composed of a LLINK field, a RLINK field, and an INFO field (which will point to or contain a key), Initially, TOP points to the root of the tree. AVAIL is a pointer to an available node, and "λ" denotes the null link.

Input: KEY

Output: The location of the match, or the updated tree if we are inserting a new key.

1) Set X $\leftarrow$ TOP, K $\leftarrow$ KEY. If X=λ, then, to insert the key, set Z $\leftarrow$ AVAIL, LLINK (Z) $\leftarrow$ λ, RLINK (Z) $\leftarrow$ λ, TOP $\leftarrow$ Z, INFO (Z) $\leftarrow$ KEY, and exit the algorithm. Otherwise (X$\neq\lambda$) proceed to step 2.

2) If KEY = INFO(X), we have a match, else set B = first bit of K, shift K left 1 bit.

3) If B = 0 then go to step 4; else to to step 5.

4) If LLINK (X) $\neq \lambda$, then set X $\leftarrow$ LLINK (X) and go to step 2; else go to step 6.

5) If RLINK (X) $\neq \lambda$ then set X $\leftarrow$ RLINK (X) and go to step 2.

6) (We had no match--insertion is done here.)
Set Z $\leftarrow$ AVAIL, LLINK (Z) $\leftarrow\lambda$, RLINK (Z) $\leftarrow\lambda$, INFO (Z) $\leftarrow$ KEY.
If B = 0 then set LLINK (X) $\leftarrow$ Z; else set RLINK (X) $\leftarrow$ Z. Exit the algorithm. (If the bit we are looking at is a 0, the key is inserted at the left; if it is a 1, we insert to the right. Nothing is sacred about this scheme--it could easily have been reversed.)

END (End of Algorithm)

The algorithm was used to create a tree of the binary numbers 000 through 111, inserted in ascending order. That tree is shown in figure 1-3. If the numbers had been inserted in descending order, the result would have been the tree in figure 1-4.

It should be noted that the maximum depth, or level, is 3, which happens to correspond to the number of bits in the key. This is the worst case. For longer keys (coded, for example, in EBCDIC) the maximum level will be much smaller than the number of bits in the key. Unfortunately, even though the digital tree is relatively balanced, it does not preserve lexicographic order. Thus, when we traverse the tree of figure 1-3 in postorder, we get the nodes out in the following sequence:

001 010 011 000 101 100 110 111

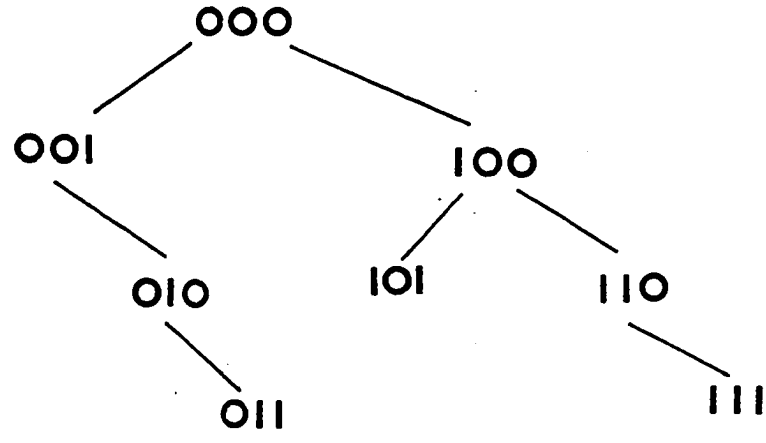


Figure 1-3. A Digital Tree
The numbers 000-111 were inserted in ascending order.

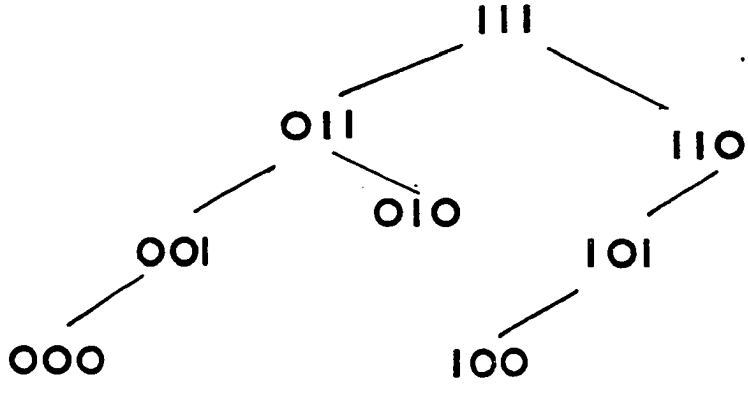


Figure 1-4. The numbers have been inserted in descending order.

2.0 The Essentials of PATRICIA

We are now ready to discuss the essential workings of PATRICIA, and to describe the basic algorithms for building the structure and searching through it. It is of some assistance to notice the similarities between PATRICIA and the structures described in Chapter One. As does the digital tree, PATRICIA uses individual bits to search for a key. However, when two keys have identical bit patterns for, say, N bits, these bits are skipped over. This eliminates the unnecessary comparisons of TRIE memory (for example, the letters "THE" of keys "THE" and "THEN"--see figure 1-1); moreover, unlike the digital tree, PATRICIA preserves lexicographic order in the manner of figure 1-2, and is not sensitive to the order in which keys are inserted.

An example of a PATRICIA tree is given in figure 2-1. Let us assume that our text is a string of binary "1"s and "0"s. Suppose we are searching for a key that starts with "10100." (PATRICIA only finds keys that start with a given pattern, but the pattern may be of arbitrary length.) We begin our search at the top of the PATRICIA tree, where we immediately advance 3 bits before making any comparison. This is a result of the fact that all keys in this particular structure start with the same two bits, although we do not see the value of these bits when we are searching. Hence they and other skipped bits are indicated by "X"s. The 3rd bit of our key is a "1"; thus we go to the right. (A "0" would have caused us to go to the left.) At this point we know that our key, in fact all keys in the right subtree, start with "XX1 - -." We advance two more bits and compare again. This time, our input key of "10100" produces a "0" (3+2= 5th bit position). Hence, we know that our

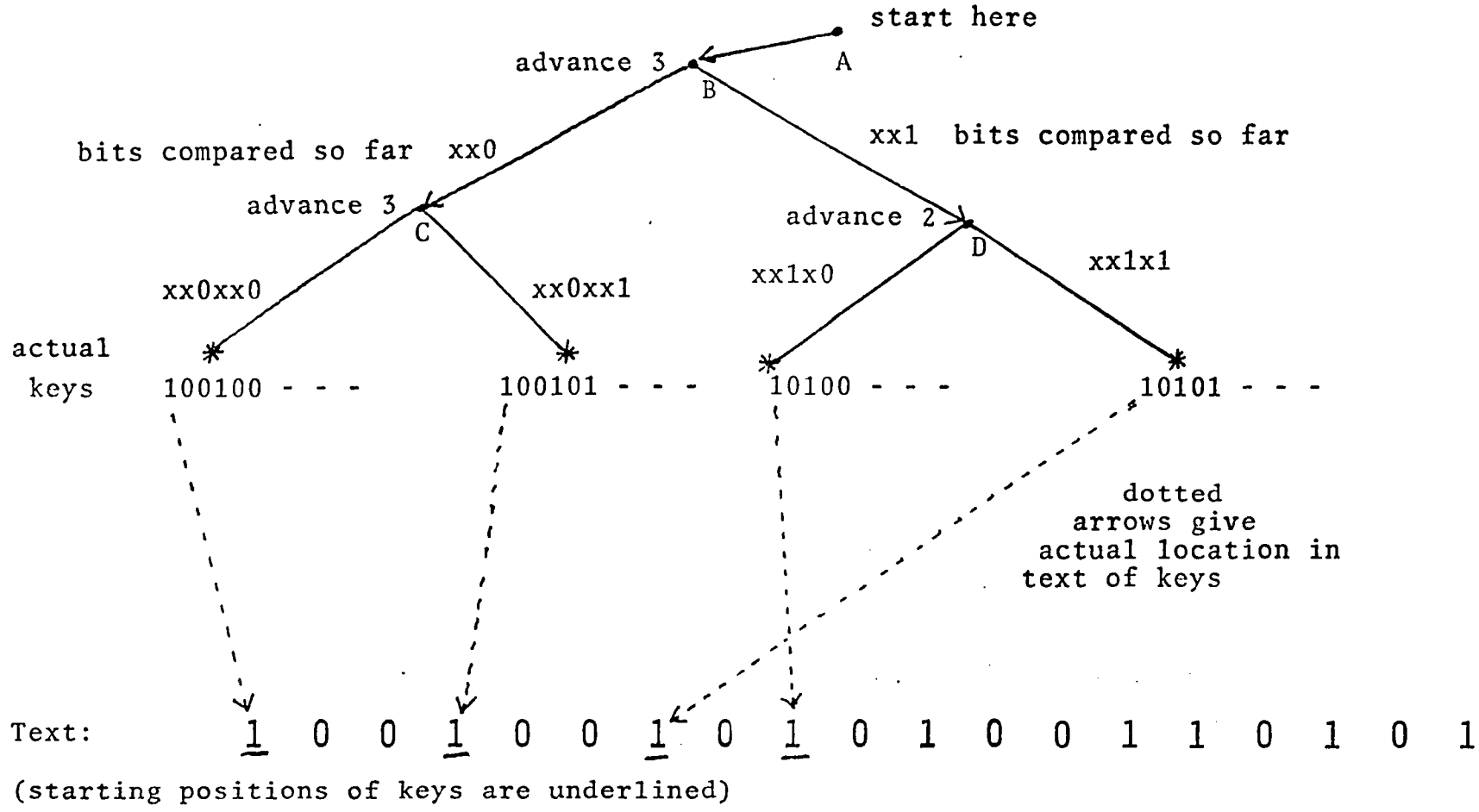


Figure 2-1. A simple PATRICIA tree. To search for a particular key, start at the top (node "A") and go to the left (node "B") where we advance 3 bits. If the third bit of our key is a zero go to the left (node "C"); otherwise go to the right (node "D"). Eventually we get to a dotted arrow, which tells us where to make the comparison with the text. We must go to the text to see if our key is actually present.

key starts with "XX1X0 - - -." Since we have reached the end of the tree (and thus the search path) we now go to the text Where directed and pick up the string "10100 - - - - ", which we compare with our key. In this case, they match. But suppose our pattern had been "11110 - - -"? The PATRICIA search would lead to the same place, but obviously the keys do not match. The mismatch is not noticed because of the bits we skipped over in our search down the tree. Hence we must always check the input key against the text that PATRICIA points to--this is the only way of verifying the existence of the key in the text. The great advantage of PATRICIA, however, is that we only do this once--namely at the end of a search. Moreover, we know that if our input key doesn't match the text starting in the position indicated by PATRICIA, then it does not occur in the text at all as a key (although the bit pattern might actually occur in the text as a non-key).

Suppose, for example, we wish to insert in the PATRICIA tree a key starting with the bit pattern "1110010 - - - -." This would require a comparison at a bit position, going from left to right, where this key first differs from all others in the tree. In this case, the key first differs from "10100" at bit position 2. But the tree of figure 2-1 has no comparison at this bit position. Hence, our tree would need a comparison at bit position 2, which would produce the structure shown in figure 2-2.

A very important property of the PATRICIA tree is that it can be thought of as a "free form" hierarchial structure. Thus, in figure 2-2 all keys starting with "10" are contained in the subtree at "B"; hence, if our input key were "10". we would find an entire subtree of matching keys--namely "100100 - - -", "100101 - - -", "10100 - - -", and "10101 - - -".

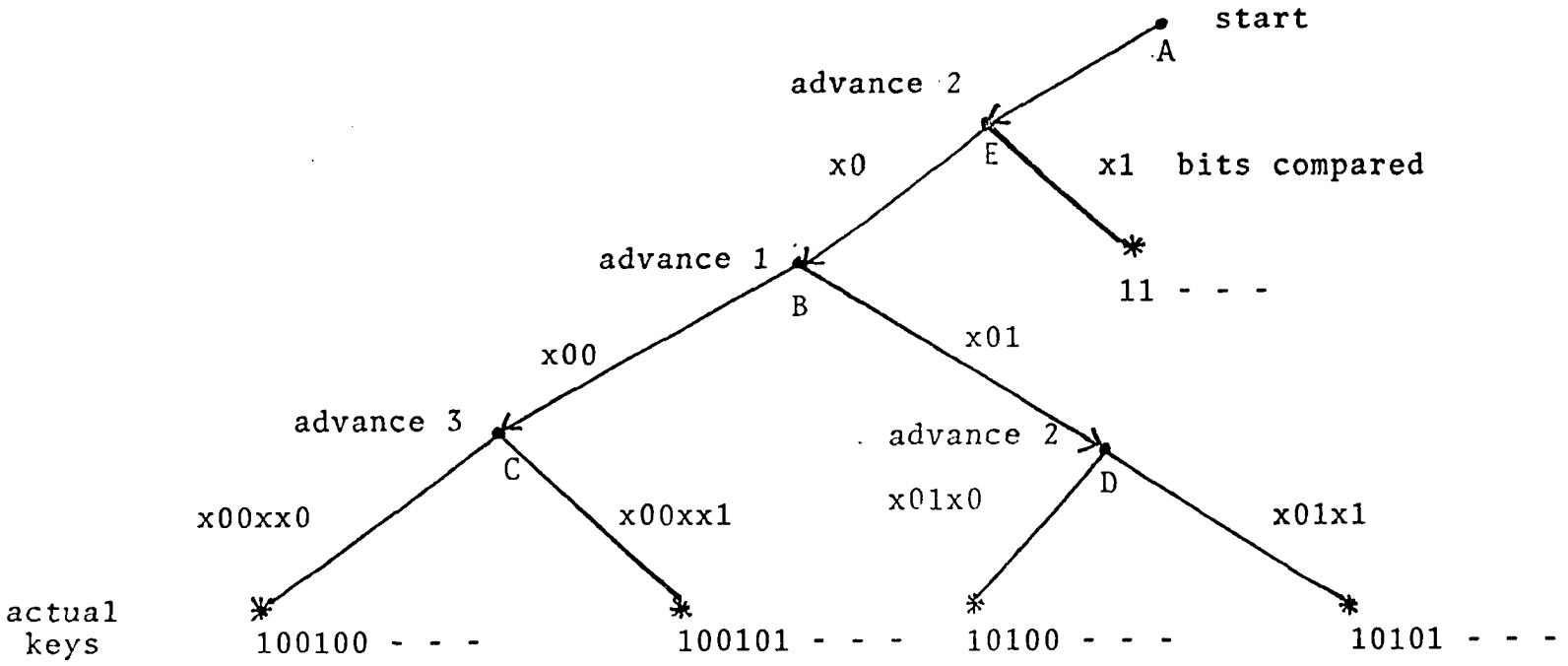
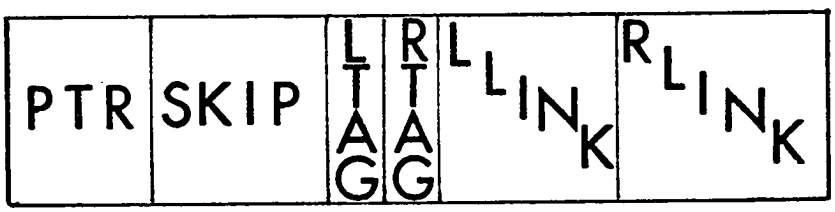


Figure 2-2. If we add a key that starts with "11 - - -" to the tree of figure 2-1, then that tree is changed into the one shown above. Note that the left subtree of node "E" is identical to the subtree of node "A" in figure 2-1, except the "advance 3" of node "B" is changed to "advance 1". This is because node "E" advances 2.

The actual data structure of a PATRICIA node is shown below. We will be working with this data structure throughout the remainder of this dissertation.



Here: PTR points to the starting position of a key in the text.

SKIP gives the number of bits to advance before making the next comparison.

LLINK, RLINK are pointers to left or right subtrees (in figure 2-1, represented by lines ending with arrows) or are pointers to PTR fields where we make a comparison with a key in the text (in figure 2-1, represented by lines ending with asterisks).

LTAG, RTAG are one bit fields indicating whether LLINK, RLINK points to a key. LTAG, RTAG may, when convenient, be represented by the sign bits of LLINK, RLINK.

A PATRICIA node actually contains two separate pieces of information. Assume we are given a node, X. Then during a search, SKIP(X) tells which bit in our input key to look at next, going to the left subtree of X if this bit is a "0" and to the right subtree of X if the bit is a "1". PTR(X) points to the start of a specific key in the text. However, PTR(X) is not looked at when we pass by node X in a search down the PATRICIA tree. When we reach the end of a search path, a backward pointer (some LLINK or RLINK where LTAG or RTAG = "1" - solid curved lines in figures 2-3 amd 2-4) points to node X, meaning we go to the text position at PTR(X) to get the key which our search path led us to. Each PATRICIA node will be pointed to by exactly one such backward pointer. Moreover, PATRICIA is constructed so that this backward pointer (or thread) always originates either in node X or in some descendant of node X. Nothing is

sacred about this; in fact the PTR fields need not be stored in the PATRICIA tree at all. Moreover, we could have the LLINK or RLINK fields point directly to the text. Unfortunately, this would be quite wasteful, for usually there are far fewer keys (hence nodes) than there are character positions in the text. Thus several extra bits would be required for a link field to represent a textual character position; these bits would be wasted whenever the LLINK and RLINK fields were used as links.

An actual example of a PATRICIA data structure (hereafter called a PATRICIA tree) corresponding to the example of figure 2-1, is given in figure 2-3. The SKIP fields, which correspond to the "advance" fields of figure 2-1, are indicated by the numbers above the parentheses. The curved lines are RLINK, LLINK fields in nodes whose RTAG or LTAG fields are "1". These curved lines correspond to those lines in figure 2-1 which end with asterisks and point to specific PTR fields. The PTR fields, which give the text position for starts of keys, are enclosed within parentheses. The dotted lines emanating from the PTR fields show where in the text the PTR fields refer to. They perform the same function as the dotted lines of figure 2-1. However, in figure 2-1 they are not specifically associated with any node. Note that the node at START has no RLINK or SKIP fields. This node serves mainly as an initializing structure (a PATRICIA tree with only one key) and a place to hold a PTR field.

The text is considered to be one contiguous character or bit string, each character position (including blanks) being numbered consecutively. Depending upon context, any character position may be indicated as the starting point for a key and thus be entered in a PTR field.

Perhaps the most distinctive feature about the PATRICIA tree is the fact that a given key has no fixed size and is unique. Keys are of

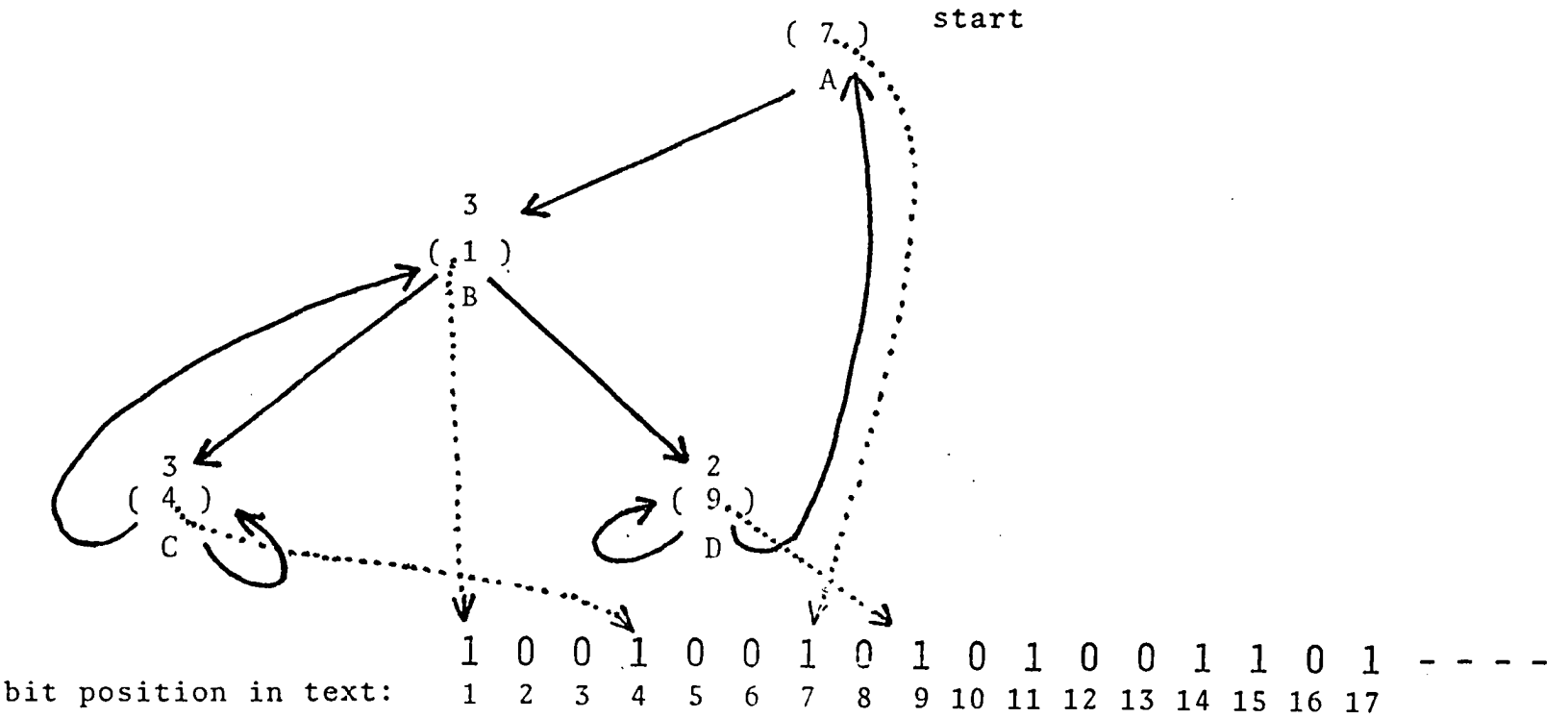


Figure 2-3. The actual representation of the PATRICIA tree of figure 2-1. The SKIP fields, which correspond to the "advance" fields of figure 2-1, are indicated by the numbers above the parentheses. The curved lines are RLINK, LLINK fields in nodes whose RTAG or LTAG fields are "1". These curved lines correspond to those lines in figure 2-1 which end with asterisks and point to specific PTR fields. The PTR fields, which give the text position for starts of keys, are enclosed within parentheses. The dotted lines emanating from the PTR fields show where in the text the PTR fields refer to. They perform the same function as the dotted lines of figure 2-1. However, in figure 2-1 they are not specifically associated with any node. Note that the node at START has no RLINK or SKIP fields. This node serves mainly as an initializing structure (a PATRICIA tree with only one key) and a place to hold a PTR field.

arbitrary length, each starting at a position indicated by a PTR field and continuing to the end of the entire character string, or text. In actuality, we will not be looking for equality between textual keys and some specific argument key; rather, PATRICIA will find all keys in the text that begin with a specific key (Knuth, 1973). Consider, for example, the sentence:

THIS IS THE HOUSE THAT JACK BUILT.

Then if we wish to indicate the start of every word as a key, the PATRICIA keys would actually be:

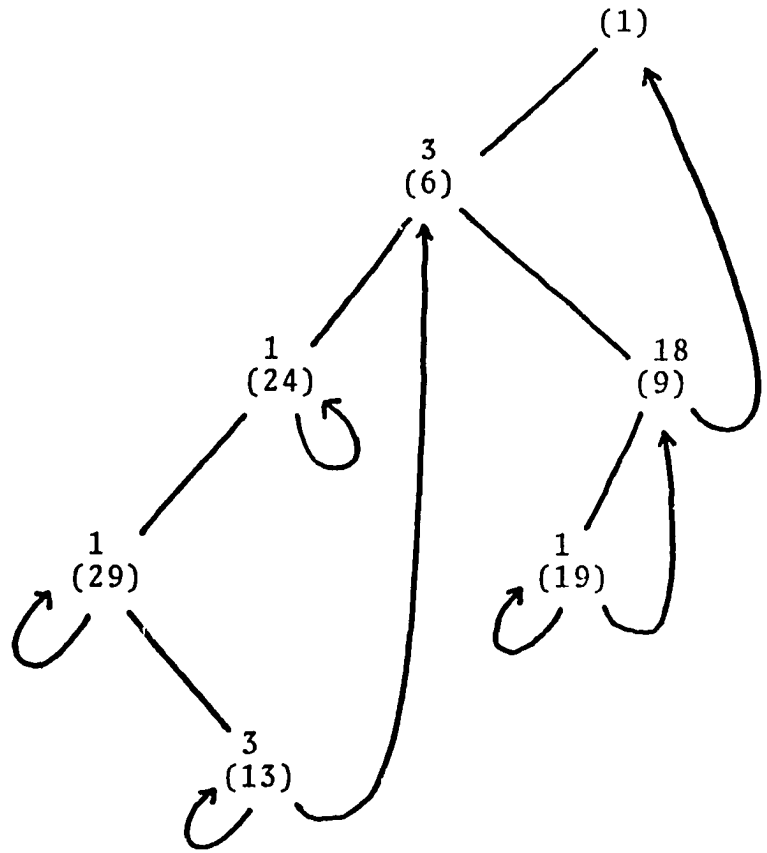
THIS IS THE HOUSE THAT JACK BUILT.
IS THE HOUSE THAT JACK BUILT.
THE HOUSE THAT JACK BUILT.
HOUSE THAT JACK BUILT.
THAT JACK BUILT.
JACK BUILT.
BUILT.

Of course, not every word need be a key; for example, if we wanted keys only at the start of the words BUILT, HOUSE, and JACK, we would have:

BUILT.
HOUSE THAT JACK BUILT.
JACK BUILT.

The problem of what to call a "key" actually falls upon the user, and will vary depending upon the application. For example, to produce a concordance, we would want every word to be the start of a key, whereas we would only want pertinent words if we were creating an index to a technical report. If a variable length field is used as a descriptor,

T	H	I	S		I	S		T	H	E		H	O	U	S	E		T	H	A	T		J	A	C	K		B	U	I	L	T
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33



Example: (19) corresponds to the key "THAT JACK BUILT".

Figure 2-4. A PATRICIA tree built by algorithms 2.2 and 2.1. Every word in the sentence is the start of a key. The PTR field is parenthesized. The SKIP field is directly above the PTR field. LLINK, RLINK fields are indicated by the lines. Curved lines mean that LTAG or RTAG = 1. Note that the top node has no SKIP or RLINK. EBCDIC internal code is used.

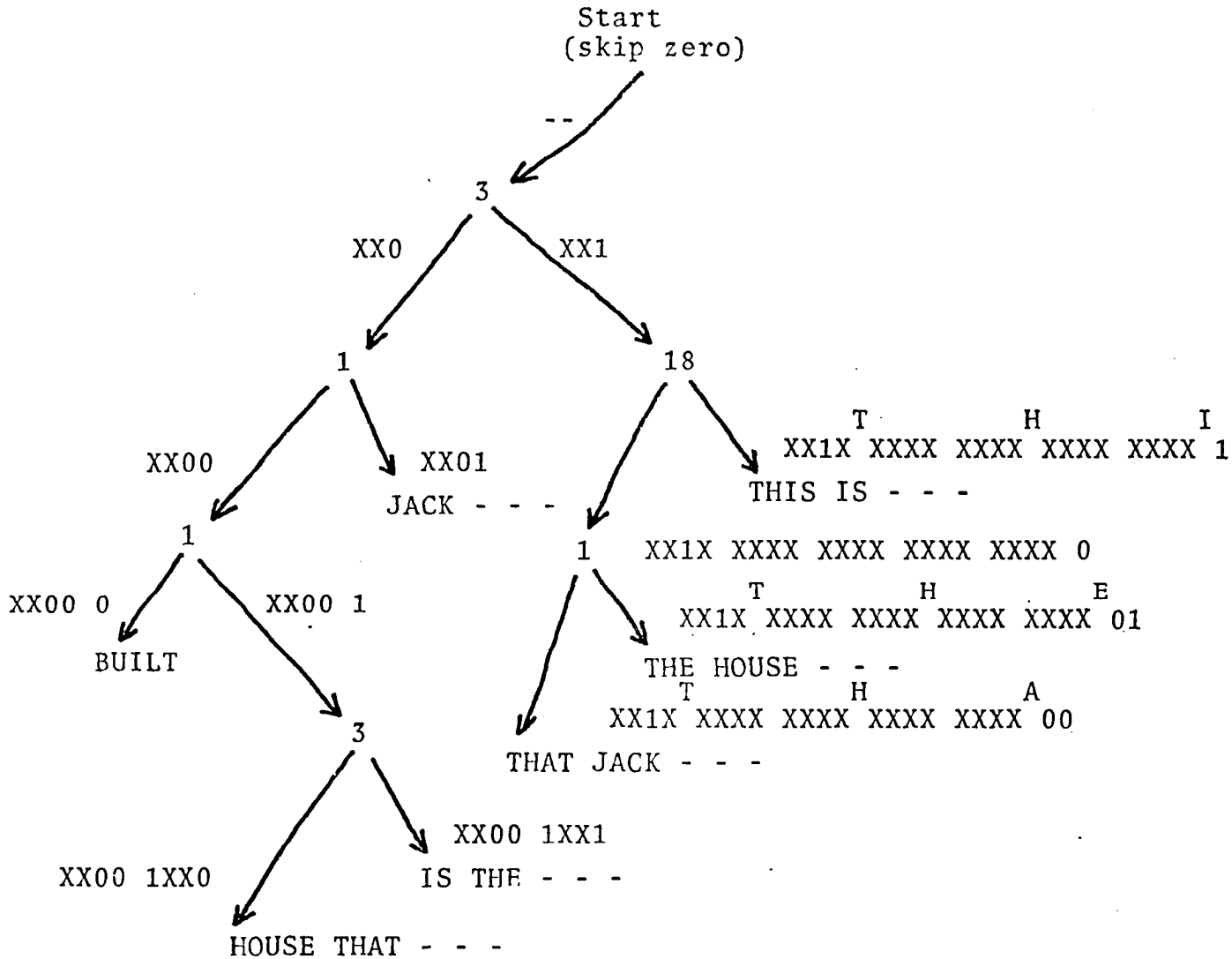


Figure 2-5. The bit comparisons that are actually made. An "X" indicates that the bit is skipped. EBCDIC code was used.

we would flag it as a key. Thus, in a student information system the start of the name field (in addition, probably, to the ID number field) would be flagged as a key. In all of the above cases, the criteria for determining keys are applied to the text during the PATRICIA tree building phase, which is the one time when the text is completely scanned. The text, "THIS IS THE HOUSE THAT JACK BUILT," where every word indicates the start of a key, was used by algorithms 2.1 and 2.2 to build the PATRICIA tree of figures 2-4 and 2-5. Since the insertion algorithm requires the use of the search algorithm, the search algorithm is presented first. It does not refer to the text until the time has come to determine whether the key being searched for is present. This is very important, since the text is probably quite lengthy and thus is relegated to secondary storage, such as a disk or a drum. An access to such storage is extremely slow compared to an access to main memory, which holds the PATRICIA tree.

2.1 Algorithm: Search for a Node in a PATRICIA Tree (Knuth, 1973)

This algorithm will search a PATRICIA tree for a specific key, K. Assume the pointer TOP points to the root of the tree. In subsequent chapters we will develop several algorithms where LTAG and RTAG are replaced by signed LLINK and RLINK fields. For this reason, steps in square brackets may be substituted if we wish to represent LTAG or RTAG =1 by a negative LLINK or RLINK. Throughout the remainder of this dissertation, P and Q will refer to specific pointer variables that are used by this algorithm.

Input: K, and the number of bits in K.

Output: P (which points to a subtree containing all the matches of K).

1) Set P ← TOP, J ← 0, N ← number of bits in K,
(See section 2.1.1 for comments about this algorithm.)

2) Set Q ← P, P ← LLINK(P). If LTAG(Q) = 1, go to step 6. [Set Q ← P, PP ← LLINK(P), P ← |PP|. If LLINK(Q) <0, go to step 6].

3) Set J ← J + SKIP(P). If J > N, go to step 6.

4) If the Jth bit of K= 0, go to step 2, else go to step 5.

5) Set Q ← P, P ← RLINK(P). If RTAG(Q) = 0, go to step 3. [Set Q ← P, PP ← RLINK(P), P ← |PP| if RLINK(Q) >0, go to step 3.]

6) Compare K to the key in the text pointed to by PTR(P). If they do not match, then K is not in the text. Assume that if they do not match, the first mismatch occurs in the L + 1st bit.

END (End of Algorithm)

2.1.1. Comments on Algorithm 2.1.

Step 1. J is the current bit comparison pointer.

Step 2. Moves to the left subtree. Initially (for the first comparison) this is always done. Subsequently, it is done whenever a "0" bit is encountered in K. An LTAG or RTAG equal to "1" indicates that LLINK or RLINK points to a node whose PTR field gives the starting character position of a key in the text.

Step 3. Skips over bits that are identical for all entries in the subtree at P. So far, our search has gotten us through the first J bits of the key. If J>N, then at this point all comparison bits will have been checked, and we will have (potentially) a subtree of matching keys.

Step 4. Determines whether to go to the left or to the right.

Step 5. Similar to step 2, except we are moving to the right.

Step 6. Since we have checked all comparison bits, we know that if K matches any key, it matches the key at PTR(P). If we compare these two keys and they do not match, we compute a value for L, where L+1 is the first bit (going from left to right) that is different for the two keys. L is used by algorithm 2.2 if an insertion is to be made. If the keys do match, we may wish to invoke algorithm 2.3, which will find all matches. Multiple matches are indicated if PP is greater than zero, which results when going to step 6 by way of step 3 when J>N.

2.2 Algorithm: Insert a New Node Into a PATRICIA Tree (Knuth 1973)

The insertion algorithm, which calls the search algorithm, is given below. It requires only two references to the text. Again, steps in square brackets are substituted when we are using signed LLINK and RLINK fields instead of LTAG and RTAG. T is a temporary variable and P, Q, and L are values returned by algorithm 2.1. (L+1 is the first bit encountered, going from left to right, where two keys are different.) (See section 2.2.1 for comments about this algorithm.)

Input: The key we wish to insert in the PATRICIA tree.

Output: The undated tree.

1) Set K ← key we wish to enter into the PATRICIA tree. R ← AVAIL, PTR(R) ← key position in text.

2) Perform algorithm 2.1. It will terminate unsuccessfully, since presumably K is not present, and will return values for L, P, and Q.

3) If L <J set N ← L (L+1 is the position of the first non-matching bit), perform algorithm 2.1 again, but exit before executing step 6. It will return values for P and Q.

4) If LLINK(Q) = P, set LLINK(Q) ← R, T ← LTAG(Q), LTAG(Q) ← 0 else set RLINK(Q) ← R, T ← RTAG(Q), RTAG(Q) ← 0. [If |LLINK(Q)| = P set T ← LLINK(Q), LLINK(Q) ← R, else set T ← RLINK(Q), RLINK(Q) ← R.]

5) If the L + 1st bit of K = 0, set LTAG(R) ← 1, LLINK(R) ← R, RTAG(R) ← T, RLINK(R) ← P, else set RTAG(R) ← 1, RLINK(R) ← R, LTAG(R) ← T, LLINK(R) ← P. [If the L + 1st bit of K = 0, set LLINK(R) ← -R, RLINK(R) ← P * SIGN(T), else set RLINK(R) ← -R, LLINK(R) ← P * SIGN(T)].

6) If T = 1 [if T<0] set SKIP(R) ← 1 + L - J. Otherwise, set SKIP(R) ← 1 + L - J + SKIP(P), SKIP(P) ← J - L - 1.

END

2.2.1 Comments on Algorithm 2.2

Step 1. Algorithm 2.1 will determine N, the number of bits in K. (Normally, we will set N to an arbitrarily large value.) We obtain a node from an AVAIL list and store the PTR field. Of course, the text has been looked at in order to obtain the key. This is the first of two text references.

Step 2. The second text reference occurs at step 6 of algorithm 2.1. Presumably, if the key had been found, an error message would be printed, because the key cannot already be present in the PATRICIA tree.

Step 3. L is found in step 6 of algorithm 2.1. If L<J, then somewhere before we reached a terminal node, there was a mismatch. We already know the search path through the PATRICIA tree, we just don't know how far down this path we must go. For this case, the new node is inserted within the tree rather than as a terminal node. The SKIP fields on the path to this node will add up to a value equal to L+1, which is the first point at which this key differs from those already encountered on the path.

Step 4. The new node is inserted to the left or right, depending upon the last successful bit comparison.

Step 5. The new PTR field will be to the left if the L + 1st bit of K is 0 (recall that the L + 1st bit is the first non-matching bit), and to the right if the L + 1st bit of K is 1.

Step 6. If T = 1 [or is < 0], the node is terminal, and we need only find the proper value for its SKIP field. Otherwise, the new node is internal to the tree, and the SKIP field of the following node must also be adjusted. In any case, the sum of the SKIP fields leading to the new node gives the first bit position of a mismatch (i.e. L+1).

2.3 Algorithm: List all Matches in a PATRICIA tree (suggested by Knuth, 1973, page 501 #14).

If more than one match for a key exists, this algorithm is used to list all occurrences in alphabetical order--for example, if the key were "T", then we would find all words starting with "T". Note that the algorithm also may be invoked for the case of exactly one match.

Let PP point to the root of the subtree containing all the key matches. PP is returned by a call to algorithm 2.1, and may already point to a PTR field, indicating a single match. For this algorithm, sign bits have been used in place of LTAG and RTAG, which means that a similar version of algorithm 2.1 must be employed. Also, for this and subsequent traversal algorithms, "A" is a sequential stack.

Input: PP (as returned by algorithm 2.1)

Output: A list of all keys in the subtree pointed to by |PP|.
or in the single PTR field pointed to by |PP|.

1) (Set stack empty, prepare to traverse the subtree at |PP|)
Set ATOP ← 0, set X ← PP

2) If X > 0, go to step 5.

3) "Visit" the key pointed to by PTR (|X|).
(X is a backward pointer, and hence is negative.)

4) If ATOP = 0 exit, else go to step 6.
(If the stack is empty, exit.)

5) (Stack node X--we shall visit it later)
Set ATOP ← ATOP + 1. Set A(ATOP) ← X. Set X ← LLINK(X),
go to step 2.

6) (Get a node from the stack)
Set X ← A(ATOP), ATOP ← ATOP - 1, X ← RLINK(X), go to step 2.

END

This algorithm is a variant of a postorder traversal (Knuth, 1968). By "visit," we mean "list the key in whatever form desired." If it is only necessary to indicate where the matches occur, we do not need to refer to the text; we simply list the PTR fields. If we wish to see the first several characters of the key, or the sentence containing the key, then we must refer to the text once for each key visited.

2.4 How Well Balanced is a PATRICIA Tree?

In subsequent chapters, we shall see that PATRICIA, as Knuth puts it, "is a little tricky and requires close scrutiny before all her beauties are revealed." One of her more useful beauties is the fact that a PATRICIA tree is usually well balanced. In fact, the only way to create an unbalanced tree is to specify a series of keys whose starting characters are not only similar, but propogate this similarity. Thus, consider the text:

HIS DOG SAID, "BOW WOW." HARRY'S DOG THEN SAID, "BOW WOW BOW WOW." THEN JOHN'S DOG SAID, "BOW WOW BOW WOW BOW WOW." MARTHA'S PARROT SUPRISED US WHEN IT SAID, "BOW WOW BOW WOW BOW WOW BOW WOW."

If every word in the above text is the start of a key, the PATRICIA

tree will be relatively balanced, except for projecting braches of "SAID BOW WOW . . . ." (See Appendix B.2)

2.5 Conclusions - Summary of Chapter Two

In this chapter, we have discussed the basic concept of PATRICIA and given the data structure which will be used throughout this dissertation. We have presented three important algorithms for:

a) Searching through a PATRICIA tree (algorithm 2.1).

b) Building a PATRICIA tree (algorithm 2.2).

c) Listing all matches to a given key (algorithm 2.3)

Moreover, we have explained that:

1) Upon building a tree, PATRICIA arranges nodes so that a post-order traversal lists keys alphabetically.

2) A PATRICIA tree is usually well balanced due to the nature of most keys.

3) It is the user's responsibility to decide upon the criteria for determining his keys; during the PATRICIA tree building phase, these keys are added to the tree by scanning the text and picking them out. This is the only time that the text as a whole is referenced.

3.0 Altering the PATRICIA tree - Node Deletion

The preceding chapter explains how to build a PATRICIA tree and search through it for a particular key. Unfortunately, many real applications require that the structure be subsequently altered--for example, we might want to eliminate from a certain PATRICIA tree the specific key "CHOCOLATE PUDDING - - - - -", or the subtree containing all keys starting with "CHOCOLATE". Without algorithms to accomplish the above types of alterations, the practical applications of PATRICIA are limited. However, if we can discover algorithms to perform such alterations, then PATRICIA becomes a powerful method for handling and updating large files containing variable length keys. The node deletion algorithms presented in the next few sections accomplish such alterations to a PATRICIA tree; moreover the algorithms are of additional importance in that they are used by the text alteration algorithms of Chapter Four.

Consider the problem of deleting a single key. The key is pointed to by some PTR field in some node, X. To delete the key we must remove this PTR field. Moreover, node X is pointed to by some backward pointer originating either in node X or in some successor to node X--namely the node at the end of the search path to our key. The final comparison in this search path must also be deleted--this involves eliminating an entire node.

Let us examine the different substructures that can occur within a PATRICIA tree. We start with two types, illustrated in figures 3-2 through 3-4. Figure 3-1 explains the special names associated with the nodes contained in the substructures.

TD -- Node containing the thread we wish to delete. Note that TD always points to PD.

PD -- Node containing the PTR field we wish to delete.

FT -- Father of TD.

TT -- Node containing the thread that points to TD.

Figure 3-1 Notation that will be used throughout the dissertation.

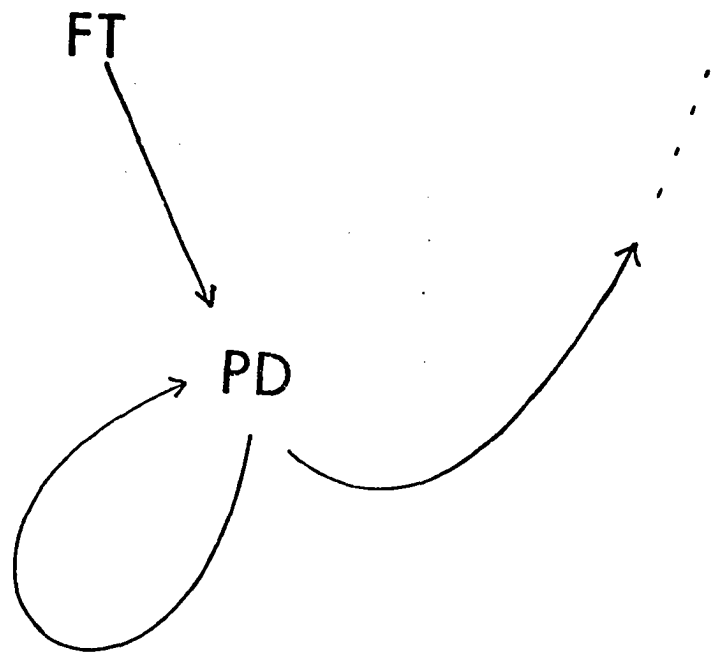


Figure 3-2 Type 1 Node Structure
Backward pointers are indicated by curved lines.

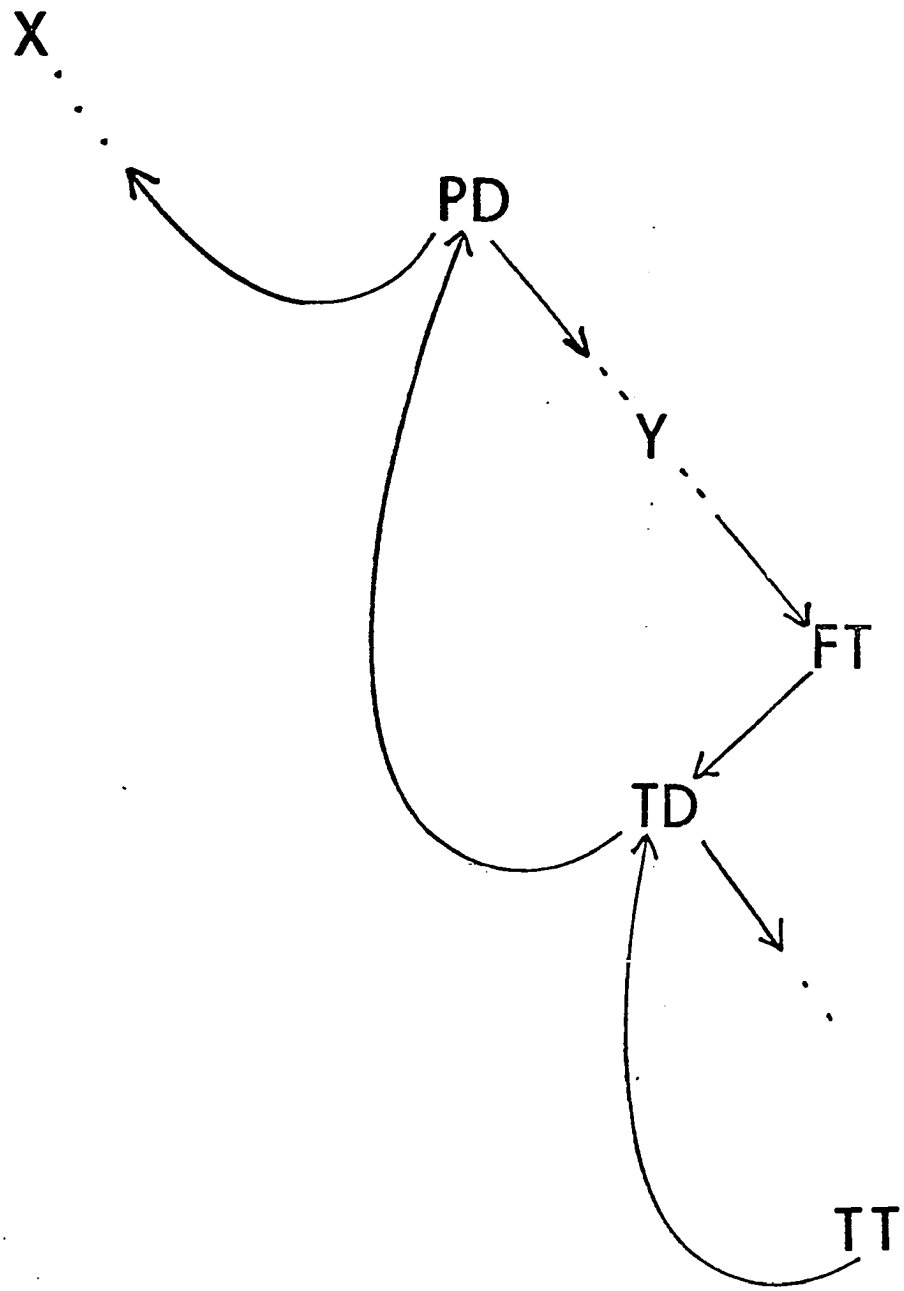


Figure 3-3 Type 2 Node Structure

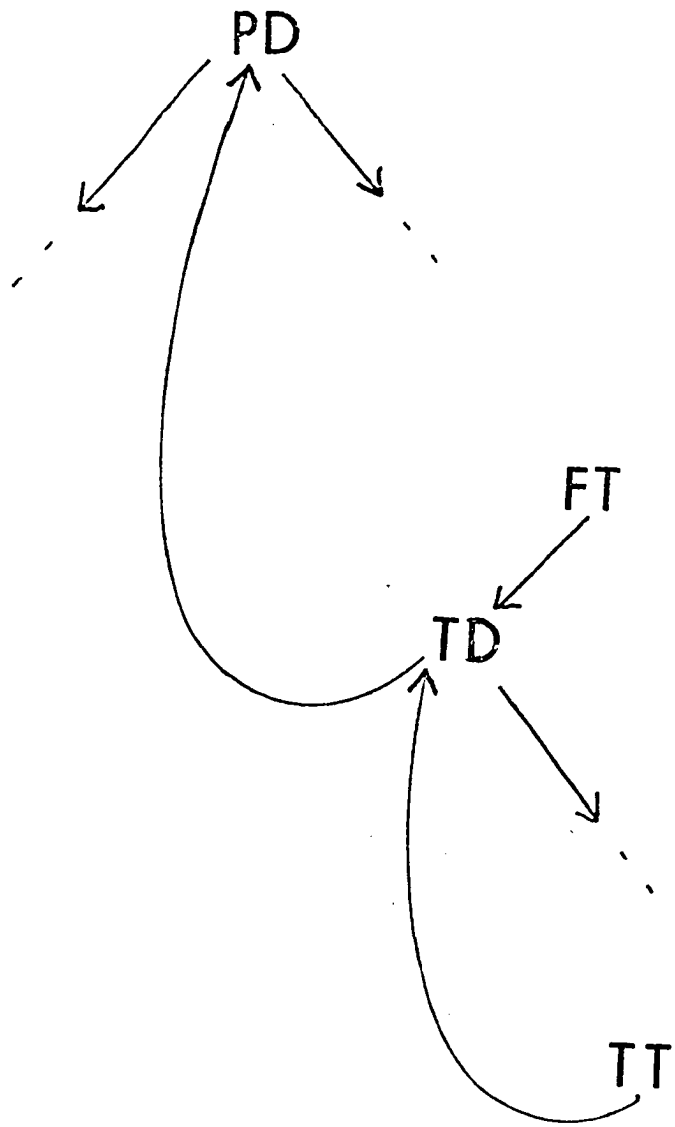


Figure 3-4 Type 2a Node Structure

Note the ambidexterity of the structures: in all three cases, pointers emanating to or coming from the right (left) could just as easily have been oriented left (right). The process of deletion for the different types is given below.

Type 1

Here we wish to delete the backward pointer that orginates from PD and points back to itself. Since the other backward pointer represents a key that is still active, it must be preserved. Note that the skip followed by the comparison at node PD is no longer necessary. Hence, we can simply remove the node, and replace the LINK field of FT which points to PD, with the backward thread of PD.

Type 2

We wish to delete the PTR field of PD (i.e. that field which points to the textual information which we no longer wish to consider as a key). However, we must preserve the comparison at PD, for the PTR field at X represents an active key; and the comparison at PD must still be made in order to differentiate between X and Y. The comparison that we are actually eliminating is at node TD. Hence we can eliminate the skip field and the node at TD, relocate PTR (TD) in PTR (PD), since PTR (PD) is no longer needed; and then adjust the backward thread in node TT that points to TD so that it now points to PD.

Type 2a

This seemingly distinct type is actually treated in a manner identical to Type 2. The only field of PD which we need to alter is the PTR field; the SKIP field and backward thread at TD are the other fields being eliminated, and their deletion is handled in the manner described in Type 2.

If we explore the deletion types a bit further, we find that the Type 2 structure has several special cases, all of which are given in figures 3-5 through 3-8. In every case the proper nodes for PD, TD, FT, TT have been indicated, even when some are identical.

Notice that Type 1 can be described as a special case of type 2; all we need to do is supply the proper values of TT and TD. This is done in figure 3-9, where type 1 has also been renamed type 2f.

It can easily be shown that no other types exist (see Appendix D); thus we need only to discover an algorithm that will properly delete a node from each of the above configurations.

3.1 Algorithm: Delete a PATRICIA Node from the Tree

The algorithm given here works for all the different types described in section 3.0. W is a pointer variable used for temporary storage. It is assumed that we have been given the key we wish to delete, determined the value for TD, and then found TT, PD, and FT. (How to do this is discussed in the next section.)

Input: TD, PD, FT, and TT

Output: PTR (PD) is deleted; the comparison at node TD is eliminated.

1) Set PTR(PD) ← PTR(TD)

2) If |LLINK(TT)| =TD, set LLINK(TT) ← -PD else set RLINK(TT) ← -PD (Note that this algorithm, as well as all others in Chapters 3 and 4, uses signed LLINK, RLINK fields instead of LTAG, RTAG.)

3) If |LLINK(TD)| = PD, set W ← RLINK(TD), else set W ← LLINK(TD)

4) If W>0 set SKIP(W) ← SKIP(W) + SKIP(TD)

5) If LLINK(FT) = TD, set LLINK(FT) ← W, else set RLINK(FT) ← W

Node TD may now be returned to free storage

END

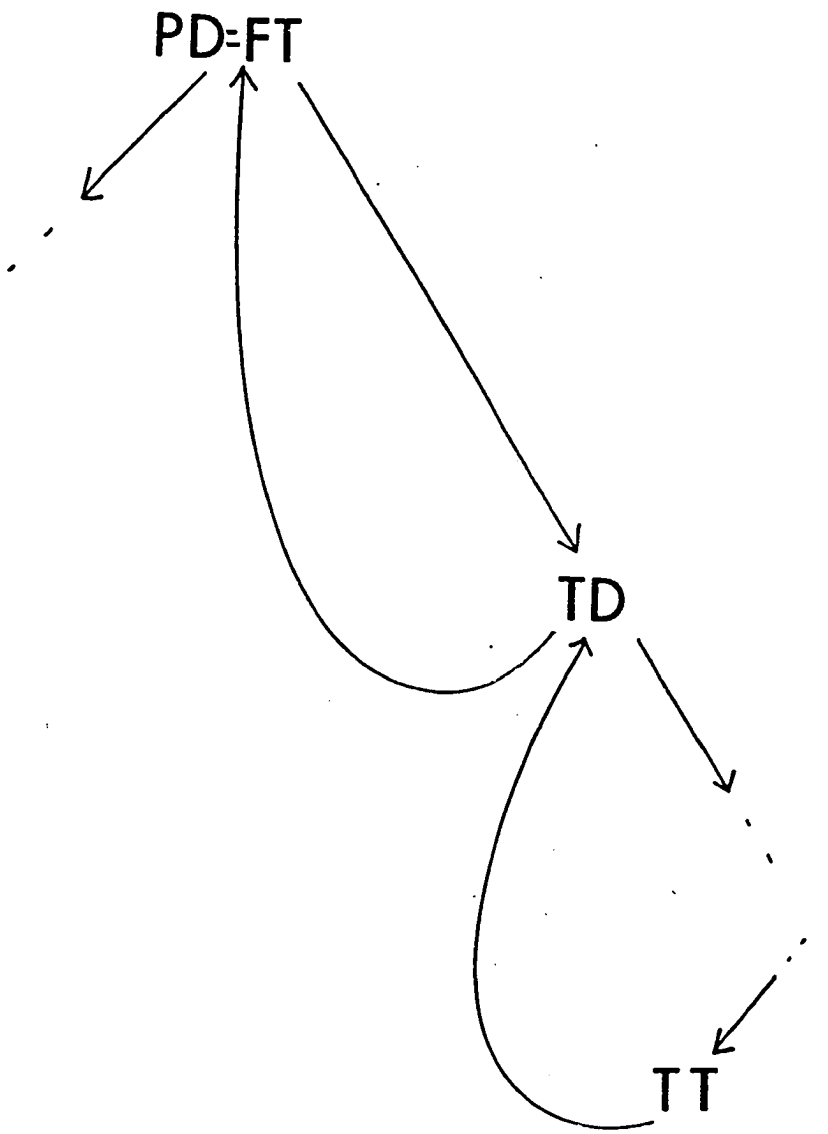


Figure 3-5. Type 2b

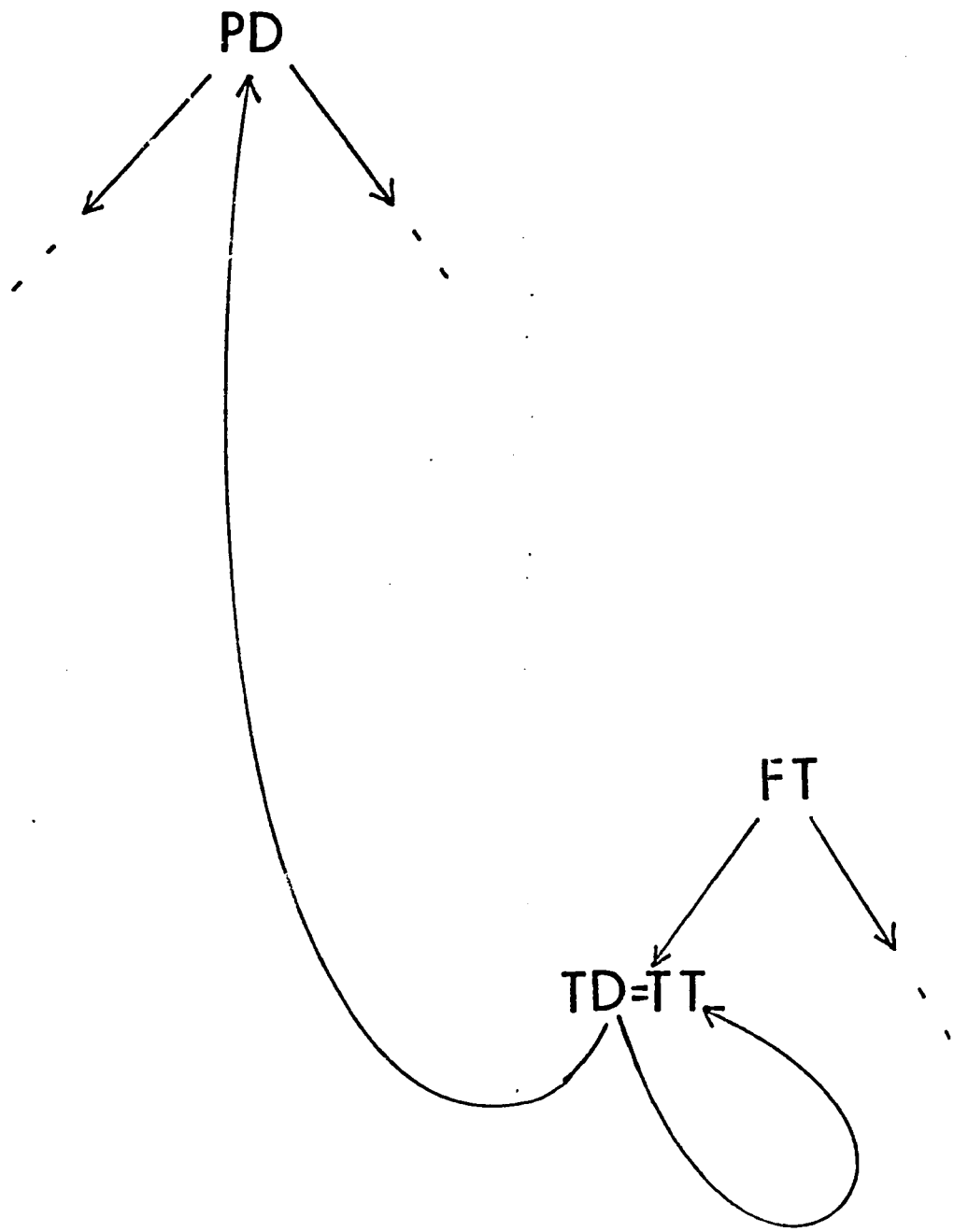


Figure 3-6. Type 2c

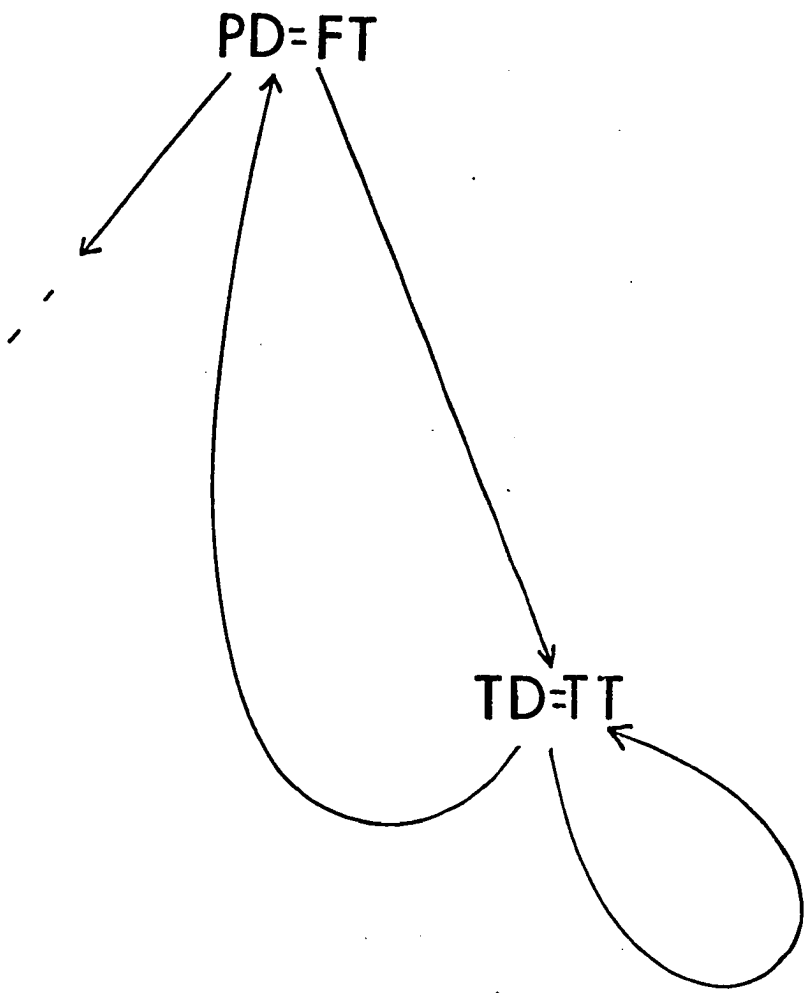


Figure 3-7. Type 2d

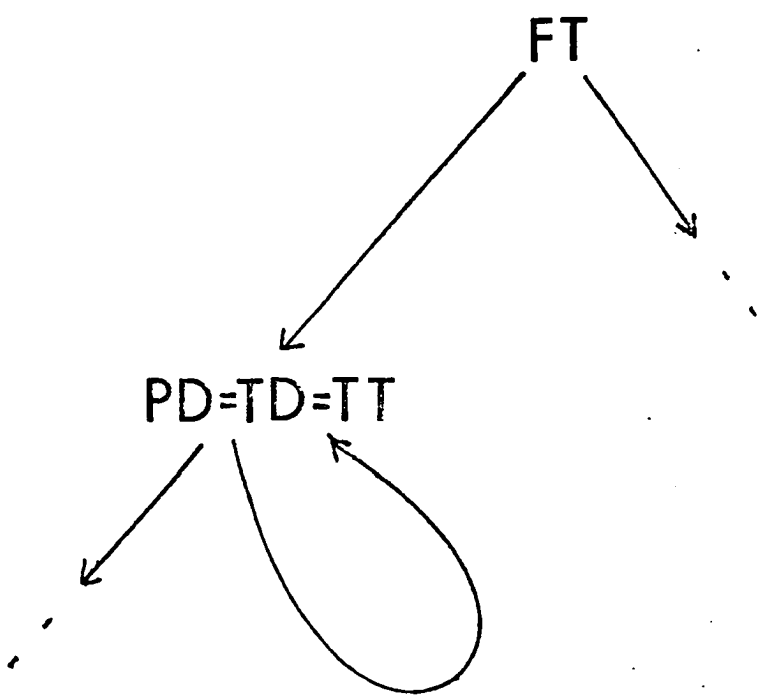


Figure 3-8. Type 2e

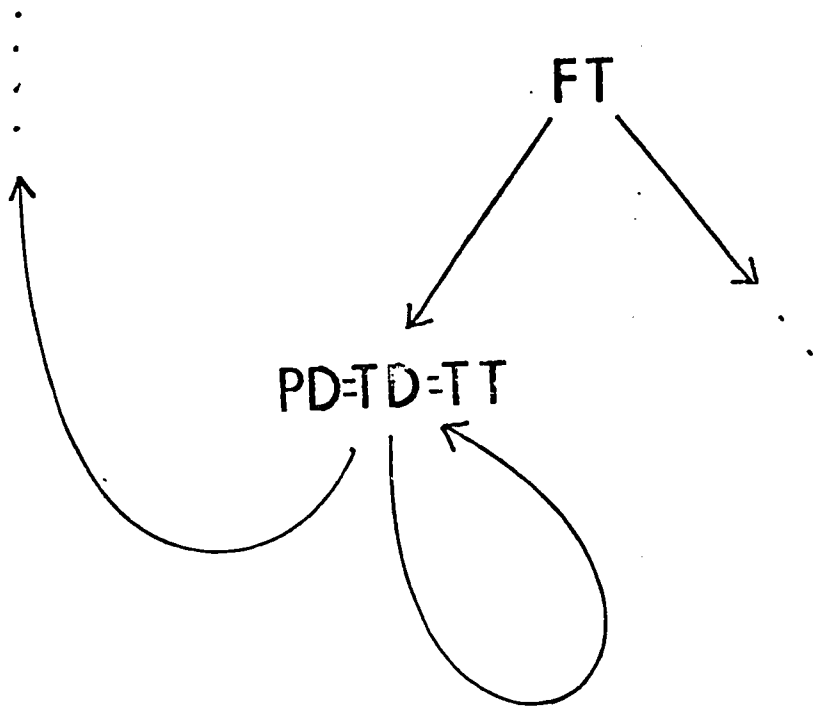
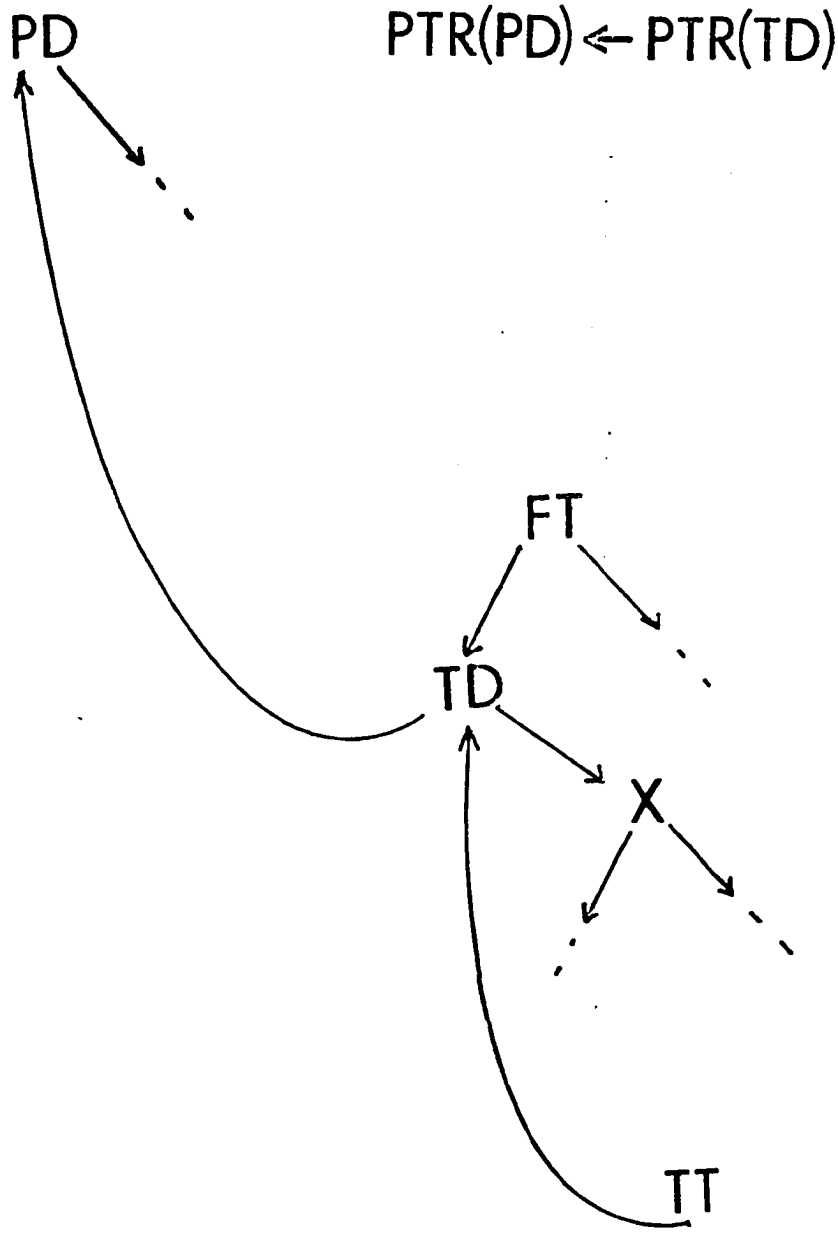


Figure 3-9. Type 2f

Type 2.

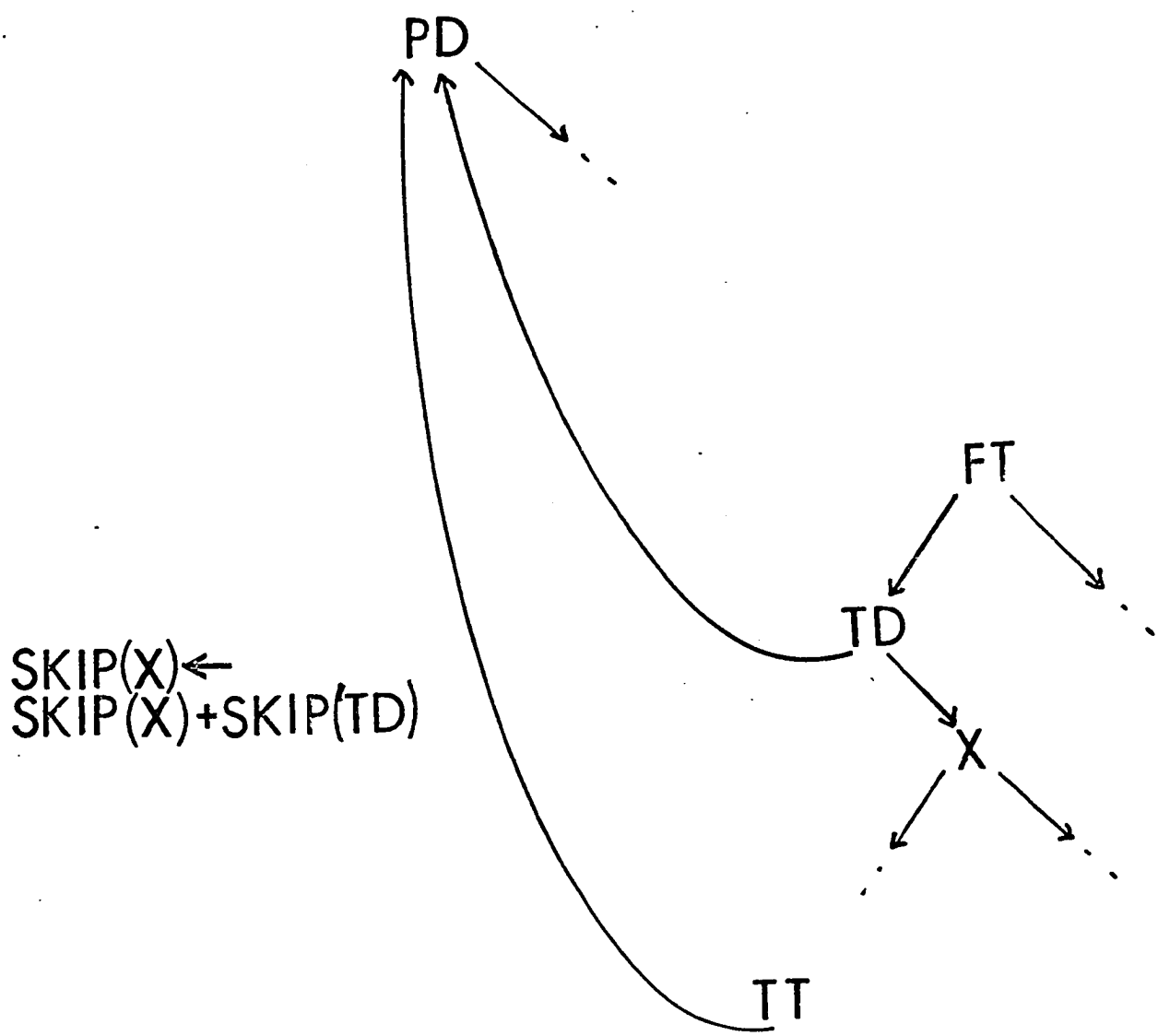
a) After step 1)

let X = node pointed to by RLINK(TD)

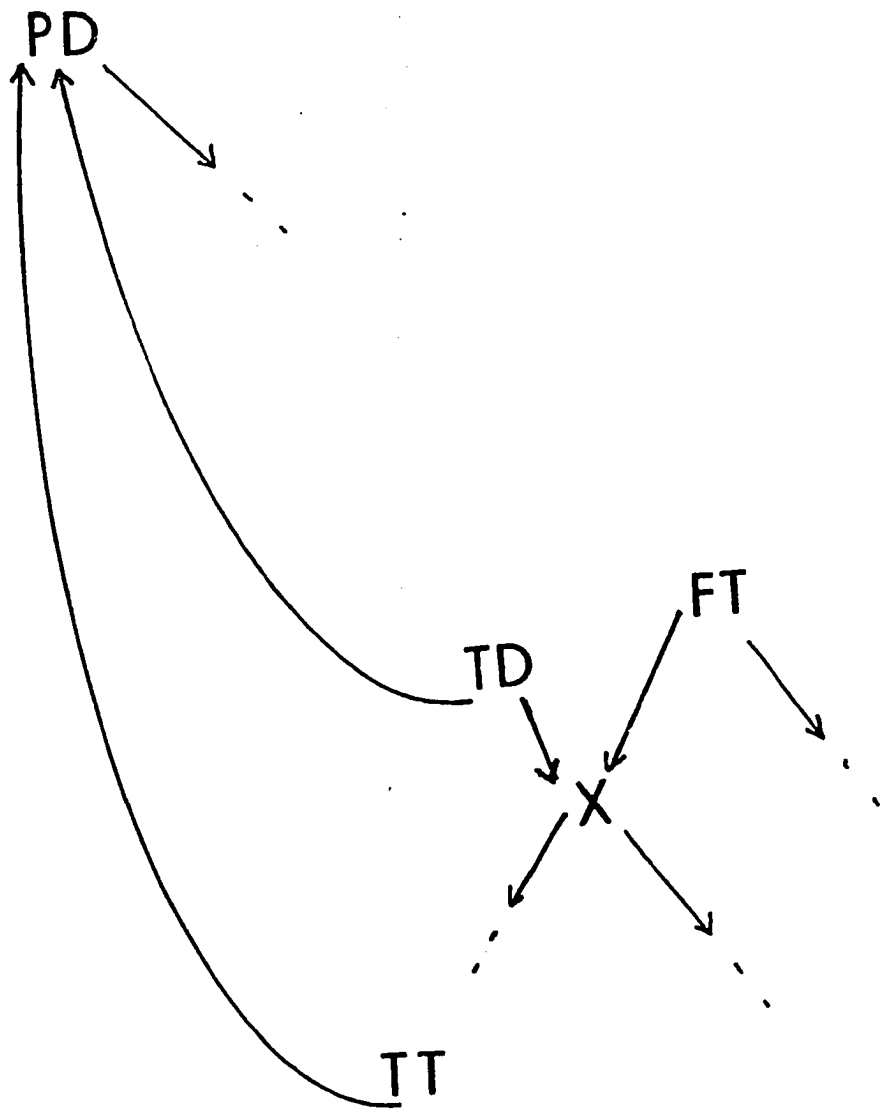


b) step 2 sets LLINK(TT)=PD as a thread

c) step 3 sets W=RLINK(TD), T=0
After step 4



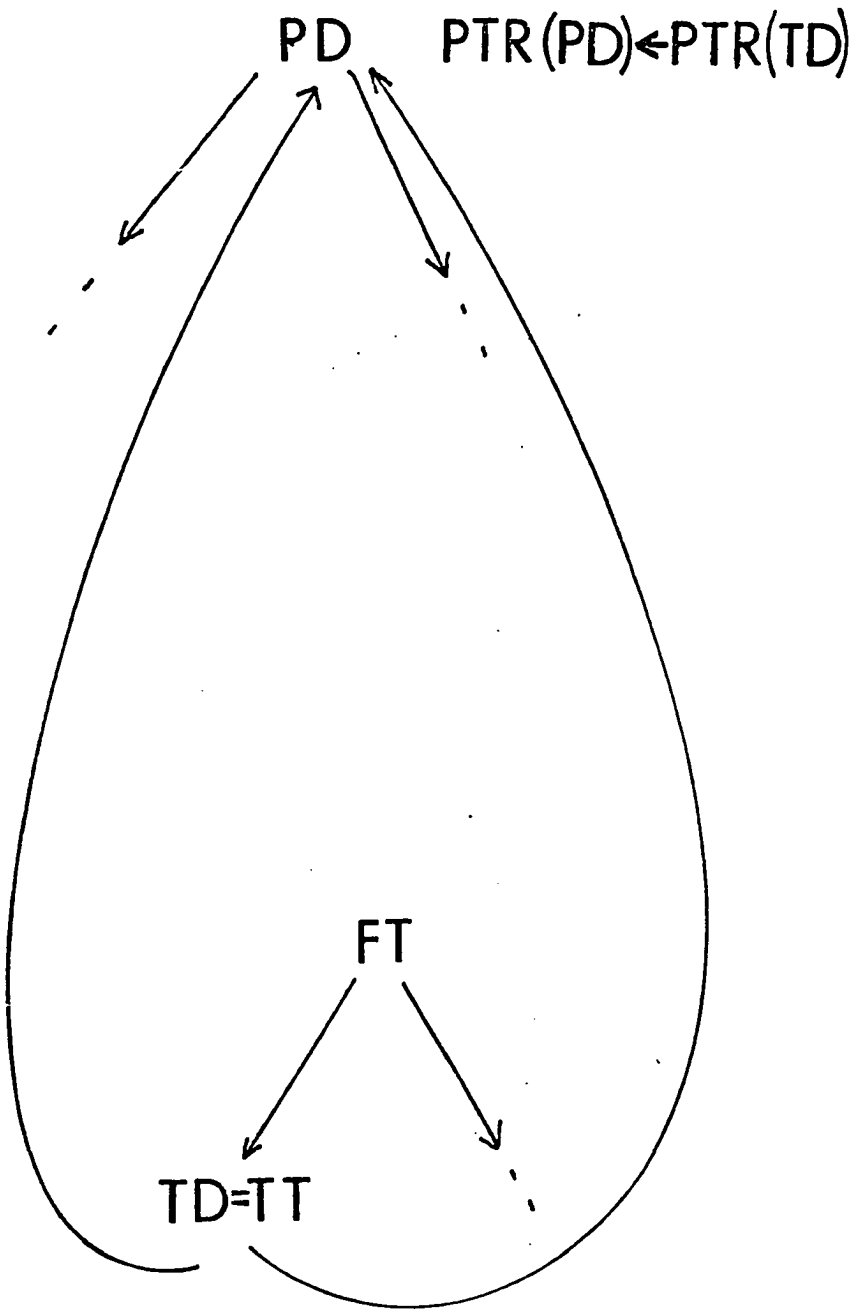
d) After step 5.



Node TD may now be returned to free storage.

Type 2c.

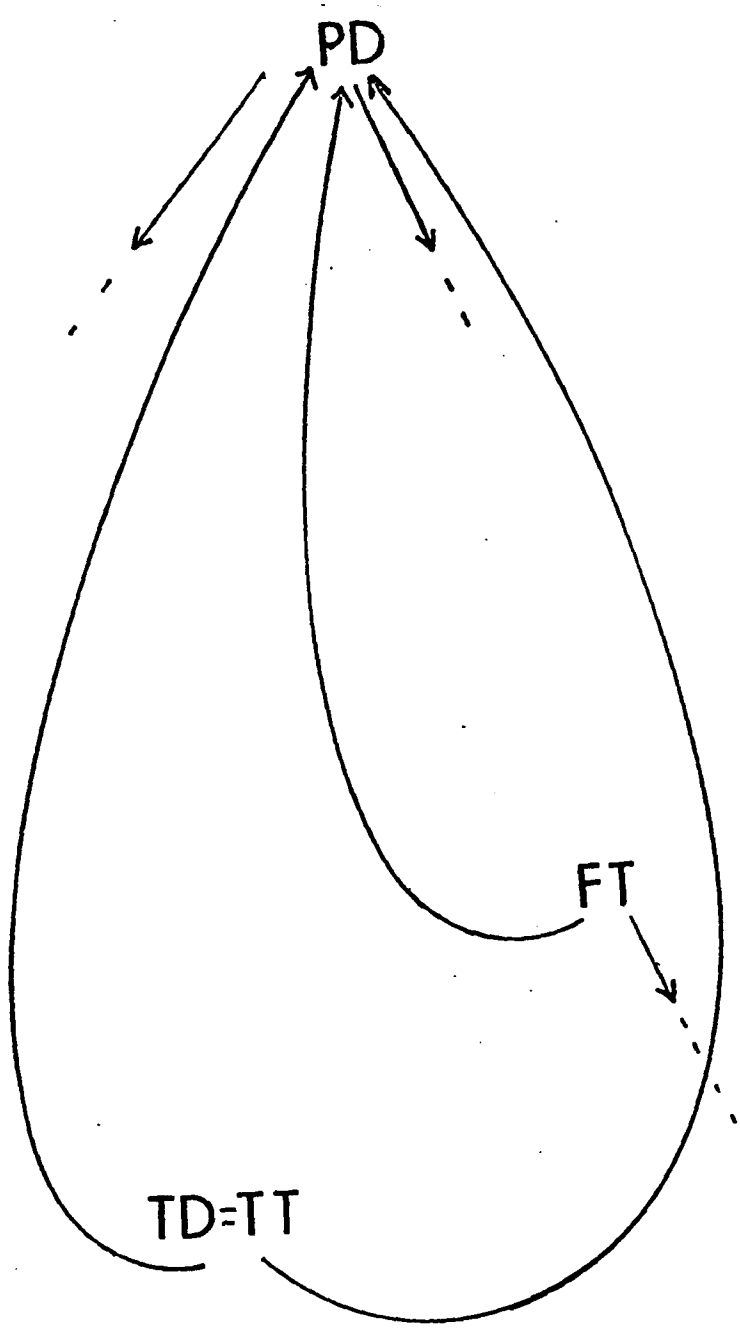
a) after step 2.



b) step 3 sets W=PD.

c) step 4 has no effect, since W is a backward pointer (W<0)

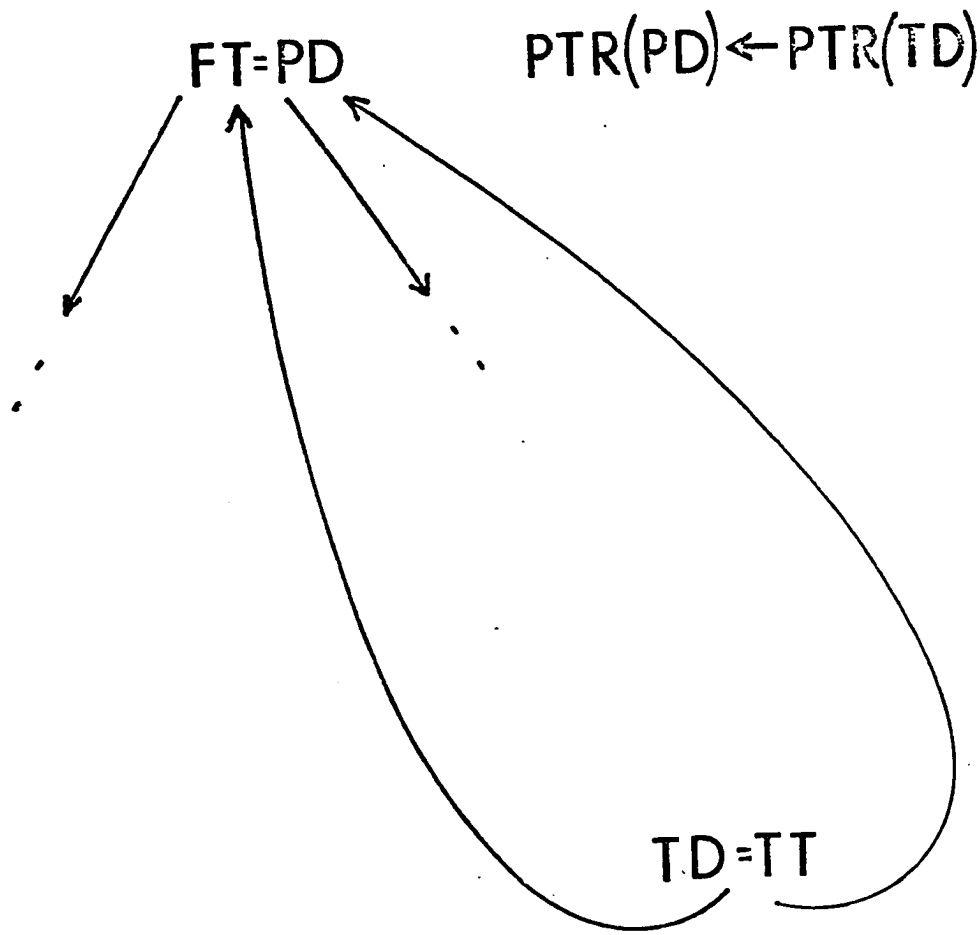
d) after step 5



Node TD may now be returned to free storage.

Type 2d.

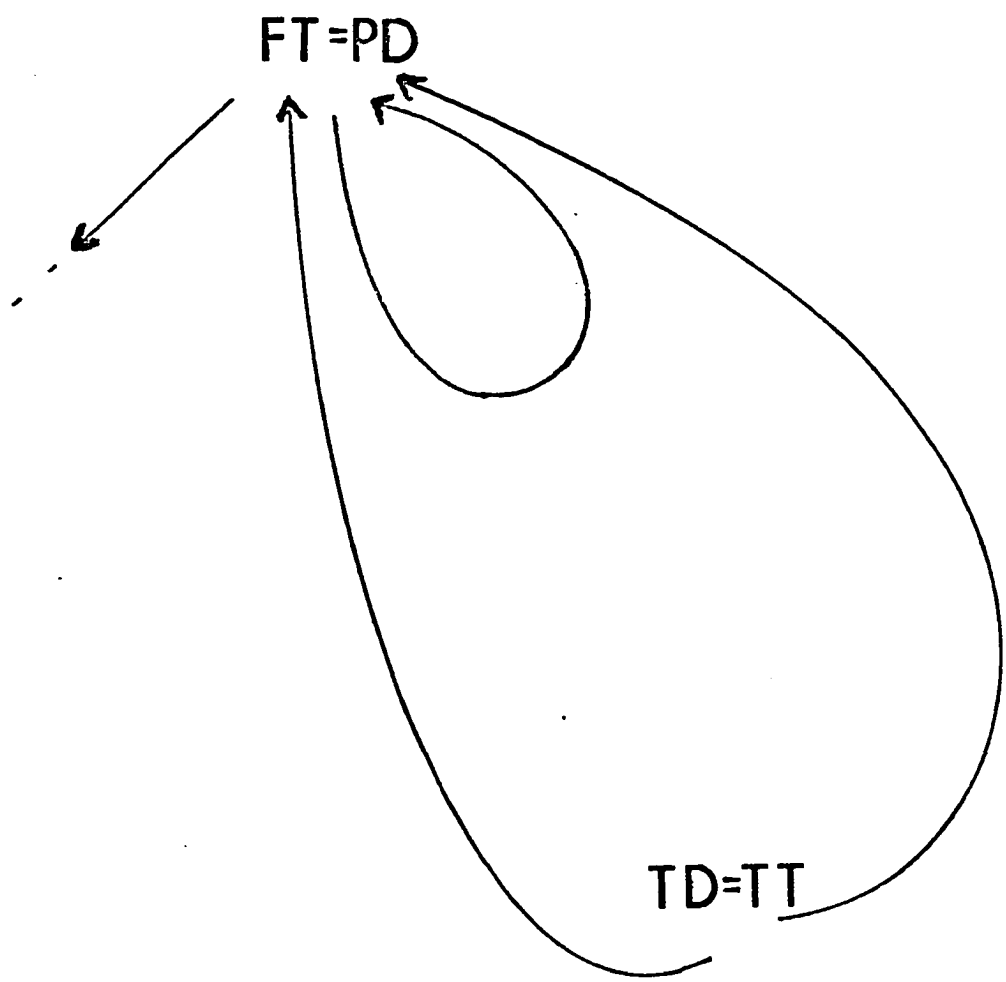
a) After step 2.



b) step 3 sets W=PD.

c) step 4 has no effect

d) after step 5



Node TD may now be returned to free storage.

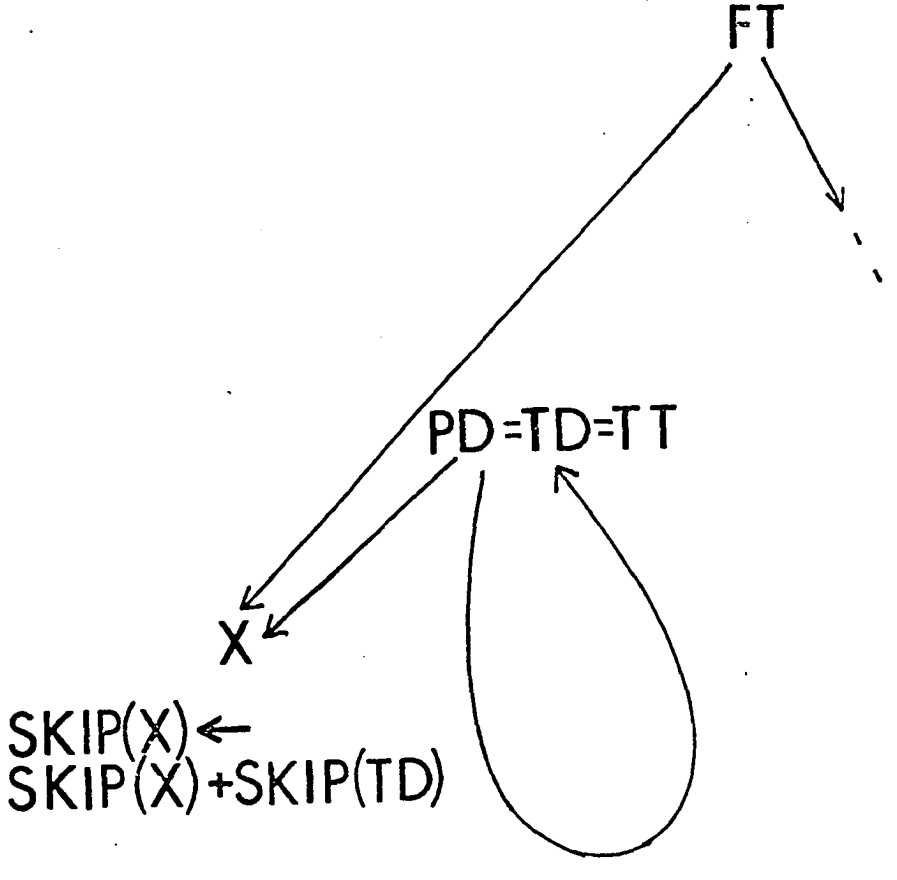
Type 2e.

a) Steps 1 and 2 do not change the structure.

b) Step 3 sets W = LLINK(TD).

c) Step 4 alters the SKIP field of the son of node TD
(It is called node "X" below)

d) after step 5



Node TD may now be returned to free storage

As an aid to understanding the deletion algorithm, each step has been traced through for Types 2, 2c, 2d, and 2e. The diagrams give the state of the particular structure immediately after the indicated step of the algorithm has been executed. (The initial LLINK - RLINK values are as illustrated in figures 3-2 through 3-9.)

3.2 Determining TD, FT, TT

As was stated earlier, initially we are given only the key we wish to delete. It is assumed that this key exists in the text somewhere (multiple occurrences are considered in the next section). By applying the search algorithm (algorithm 2.1) to this key we will be led automatically to node PD, which is the final value of P in the search algorithm. The final value of Q gives us node TD; moreover the value of Q just prior to this gives us node FT. The proper value for TT may be obtained in two or three ways. If both link fields of TD point backward, we know that one of them points to TD and originates at TD. The node called TT is the node that points to TD. Hence, in this case TT and TD are the same. This corresponds to the situation where LLINK(TD) and RLINK(TD) are less than zero; but since at least one of RLINK(TD), LLINK(TD) must be less than zero (i.e., TD has at least one backward pointer) then it is sufficient to test whether the product of RLINK(TD) and LLINK(TD) is greater than zero. If this is true, then they must both be less than zero; and hence, TT = TD.

Another special case where we can quickly determine the value for TT is when PD = TD (this arises in types 2e and 2f). In this case, the thread pointing to PD is also the thread pointing to TT, thus if TD = PD then TD = TT.

For types 2, 2a, and 2b, we must employ an alternative method to find TT. One way of doing this is to traverse the subtree hanging from

TD until we come to the thread which points back to node TD. This thread, of course, emanates from node TT.[1]

We can obtain an estimate of the average size of the subtree at TD. To make things easy, assume we have a balanced tree of 2^n-1 nodes, and thus n levels of search paths, including threads. Thus the total size of all the subtrees is given by the sum of the sizes of all subtrees at every level, or:

$$2^{n-1} + 2(2^{n-1}-1) + 4(2^{n-2}-1) + \ldots + 2^{n-1}(2^{n-(n-1)}-1)$$

$$= 2^{n}-1 + 2^{n}-2 + 2^{n}-4 + \ldots + 2^{n}-2^{n-1}$$

$$= (n-1)2^n - (1 + 2 + 4 + \ldots + 2^{n-1})$$

$$= (n-1)2^n - (2^n-1)$$

$$= (n-2)2^n + 1$$

and the average size is obtained by dividing the above expression by the total number of subtrees, which equals the total number of nodes. Hence, the average subtree size is given by:

$$\frac{(n-2)2^n+1}{2^n-1} \approx \frac{(n-2)2^n}{2^n} = n-2$$

Since node TT can occur anywhere in the subtree with equal probability the average search path length is half this, or $\frac{n-2}{2}$

---

[1] Another more straightforward method employs the search algorithm. In this case, we retrieve the key from the text which starts at position PTR(TD), then search for it. Of course, this method is slower since it requires a reference to the text; nevertheless, it is the method included in the algorithm presented below, because it was included in the test program of Appendix B.

3.2.1 Algorithm: Find PD, TD, FT, TT

This algorithm finds the values for PD, TD, FT, and TT. After the algorithm has been executed, algorithm 3.1 is used to complete the deletion. Assume that the search algorithm (Algorithm 2.1) has been modified in steps 2 and 5 to read:

Step 2: set FT ← Q, Q ← P, P ← LLINK(P), etc.

Step 5: set FT ← Q, Q ← P, P ← RLINK(P), etc.

Recall that in algorithm 2.1 P ends up pointing to the PTR field of the matching key, which in this case is contained by node PD, and Q ends up trailing one node behind P. The above modification thus effectively locates FT as the node just behind the node pointing to PD.

Input: The key we wish to delete, P (as returned by algorithm 2.1).

Output: TD, PD, FT, TT (all of these are defined in figure 3-1)

1) Set KEY ← the key we wish to delete, N ← number of bits in this key.

2) Call algorithm 2.1. The search will be successful. Set PD ← P, TD ← Q. FT will have been automatically determined if the search algorithm is modified as shown above.

3) If LLINK(TD) * RLINK(TD) > 0 or TD=PD, then set TT ← TD, go to algorithm 3.1, step 1.

4) Set N ← ∞, KEY ← Key at text position PTR(TD). Call algorithm 2.1 again. The search will be successful, and upon existing algorithm 2.1, set TT ← Q, go to algorithm 3.1, step 1 to complete the deletion.

END

3.3 Subtree Pruning - Deletion of Prefixes

We may wish to delete an entire group of nodes at one time; for example, all keys with a particular prefix, or all keys that start with

a certain word. This amounts to deleting an entire subtree. In figure 2-4, for example, to delete all words starting with "T," we delete the subtree whose root is at (9). Unfortunately, the backward pointers complicate matters, for in reality, we are deleting not the nodes of the subtree, but the PTR fields pointed to by all the backward threads in the subtree. Notice, however, that all the pointers except one are contained within the subtree. This means that we only need to fuss with one backward pointer during deletion--namely the one pointer to the ancestor of the subtree. Since the other pointers are all contained within the subtree, the subtree may be lopped off at its root and returned to free storage. The essential structure is given in figure 3-10, where R is the root of the subtree; and FR is the father of the root. TFR contains the thread pointing back to FR, and TA contains the thread pointing back to A, which is the one ancestor that contains a PTR field we need to delete. Note that TFR cannot be contained within the subtree, for a subtree must contain a backward pointer to an ancestor. The exception would be if FR and A were the same node. Then TFR and TA would also be the same node, which leads to the structure shown in figure 3-11.

Note that when we delete the subtree at R, the comparison at FR is no longer necessary. This means that node FR may be deleted. We must, however, save the PTR field of FR somewhere, and adjust TFR so that it points to the newly located PTR field. Since the PTR field of A is being deleted, we can save PTR(FR) in PTR(A). Thus, once we have saved PTR(FR) in PTR(A) we adjust TFR so that it points to A, and delete node FR in addition to the subtree at R. Luckily, this corresponds to a type 2 deletion! We need simply to insure that the proper structure is presented to the deletion algorithm. First, let's redraw the subtree

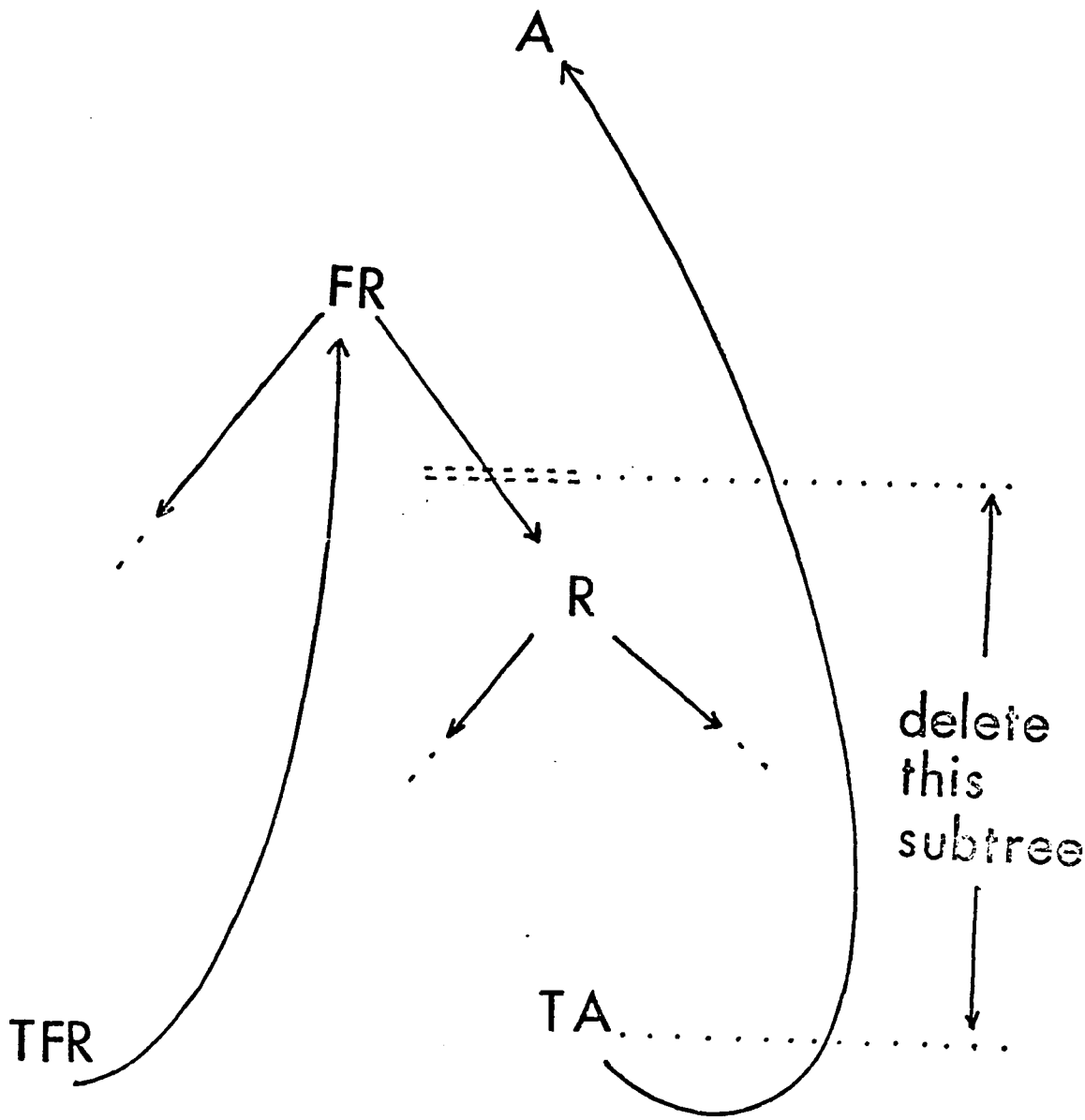


Figure 3-10 General Configuration for a Subtree Deletion
R is the root of the subtree being deleted.
TA contains the backward pointer that points
to an ancestor, A, of the subtree.

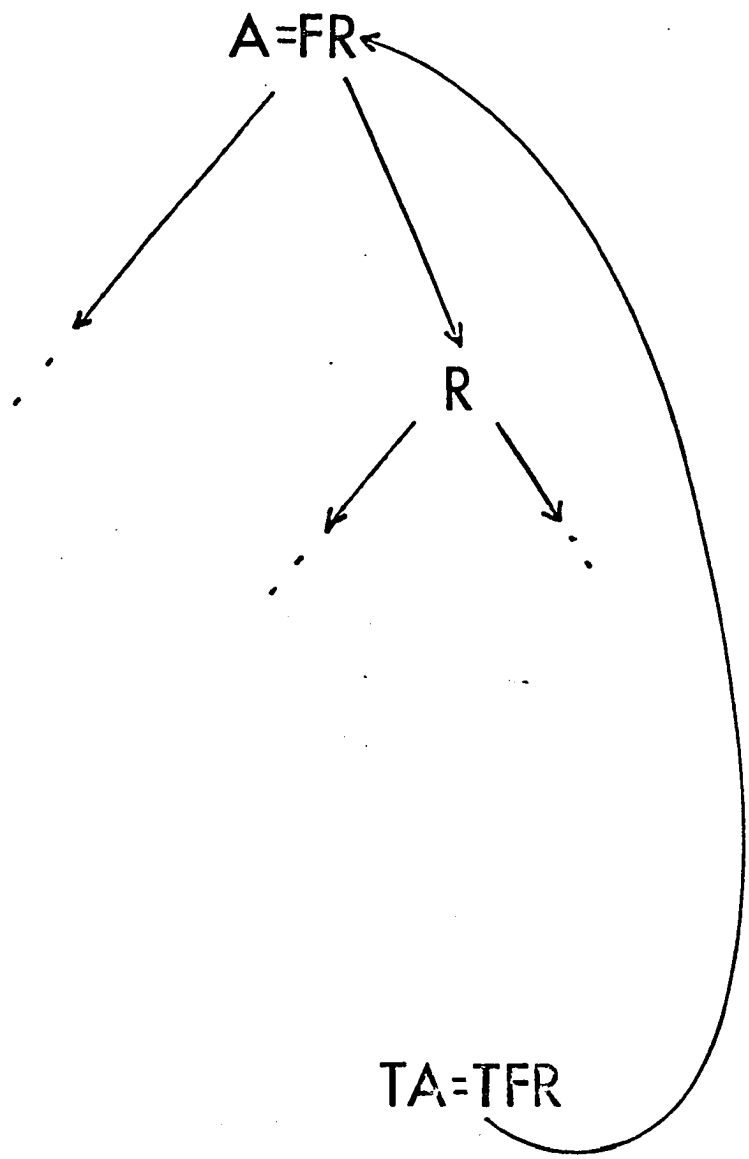


Figure 3-11. Special Case where TFR=TA

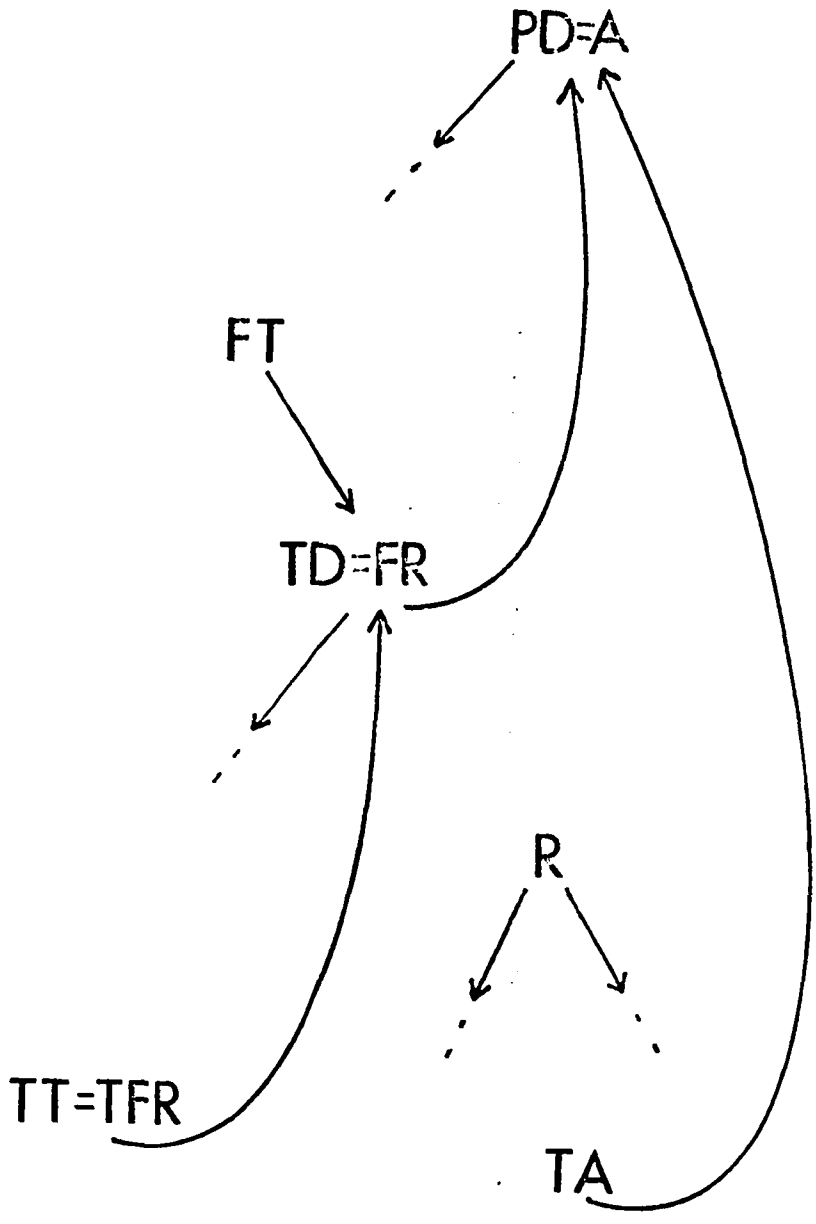


Figure 3-12 Subtree deletion corresponds to a type 2 deletion if the structure is properly set up. The subtree being deleted has its root at R. Once RLINK(TD) has been pointed to PD, we can run through the deletion algorithm. The nodes contained in the subtree and node TD are returned to free storage.

structure, putting TD, FT, TT, PD where they are required (figure 3-12). In doing this, we have changed the RLINK field of TD into a backward pointer to node A. The usual deletion process may now take place.

3.3.1 Preparing the structure for Subtree Deletion

If the search algorithm indicates that an entire subtree matches our key (see Algorithm 2.1, step three) then upon exit, P will point to the root of the subtree, Q will be at FR (note that for this case FR=TD, the node containing the thread we are going to delete). If algorithm 2.1 has been modified to find FT (section 3.2.1), we will have the proper value for FT also. Thus we need only locate TT (or TFR) and PD (or A--see figure 3-12). Locating TT is easy. We simply search for the key starting in the text at PTR(TD); upon exit from the search algorithm this time, Q will point to TT. (Again, we could avoid a text reference by traversing the subtree at TD until we find the thread that points to TD.)

Finding the node TA is, unfortunately, not quite so easy. All we know is that TA is in the subtree at R, and A is an ancestor of R and thus is along the search path to R. Hence, we can locate A only by looking at every thread in the subtree at R and seeing if it points to a node along the search path to R. One way of effecting this is given below:

1) Set KEY ← the key leading us to the subtree at R
2) Traverse the subtree at R. At each "visit," do the following:
   a) Note the node pointed to by the thread we are visiting. Call this node "A" (i.e. an ancestor of the subtree at R.
   b) Call algorithm a.1 and search for KEY. (The search will be successful and will end at R, the root of the subtree.) While going through the search, note whether we encounter

node "A". This requires a modification to algorithm 2.1. (The modification is given explicitly below.)

3) As soon as we encounter node "A" we may terminate the traversal; otherwise continue with the traversal.

We may combine the search for A or PD, with the search for TFR, or TT. This is done in the algorithm given below, which completes the preparation of the structure.

3.3.2 Algorithm: Prepare the Structure for Subtree Deletion

(Assume that we have just exited from the search algorithm and have determined that a subtree of matching keys exists.)

Input: P, Q, W (output by a modified version of algorithm 2.1. where W follows Q down the search path, and Q follows P.)

Output: PD, FT, TD, then exit to algorithm 3.2.1 step 3.

1) Set TD ← Q, FT ← W, R ← P, PD ← 0 (Again, algorithm 2.1 is modified--this time so that W is always set to the previous value of Q as Q and P are going down the tree. R is the root of the subtree we are deleting.)

2) Employ algorithm 2.3 to traverse the subtree at P. At each visit to node X set A ← X; then call a variant of algorithm 2.1 which, in addition to the introduction of pointer W, has been modified as shown below.

add to steps 2 and 5:

If P=A, set A ← 0.

(Recall that X is always a backward pointer to a node visited by algorithm 2.3, step 3. If node X is on the search path between the root of PATRICIA and R, the root of the subtree being deleted, then the modified version of algorithm 2.1 will set A ← 0.)

3) If A ≠ 0, repeat step 2 for the next node visited. Otherwise, set PD ← X. (If A=0 we have found the pointer to the node outside the subtree.)

4) If RLINK(TD) = R set RLINK(TD) ← - PD, else set LLINK(TD) ← - PD. (This forces the configuration of figure 3-12; the minus sign indicates that the RLINK or LLINK is changed to a backward pointer.)

5) Return the subtree at R to free storage. Then call algorithm 3.2.1 at step 3 to find TT and then delete node TD.

END

Note that the special modifications to the search algorithm, algorithm 2.1, do not in any way interfere with its operation except for the additional time involved. The same can be said about the modification suggested in section 3.2.1.

3.4 Conclusion - Deleting Nodes

The algorithms of Chapter Two have been supplemented with original algorithms that allow us to delete a particular node from a PATRICIA tree. Moreover, if an entire group of similar keys is no longer needed, then a subtree of nodes may be deleted. Thus, PATRICIA has been expanded into a much more flexible information retrieval system. To complete this expansion, we must still find a technique for deleting information from or inserting information into the main body of text. This is accomplished in the next chapter.

4.0 Deletion of Text Material

The processes of deleting textual information can be broken down into three specific classes.

a) Deletion of a specific key prefix which starts at exactly one point in the text.

b) Deletion of contiguous text that contains zero or more keys.

c) Deletion of a specific key which occurs at several points in the text.

Processes (a) and (b) can be combined since (a) is actually a subset of (b). The major difference is one of specification: process (a) involves specifying a particular key and having it searched for. Process (b) requires that we give the starting and ending points of the area we wish to delete; thus any starting points of keys that are contained within this area will cause the appropriate keys to be deleted from the structure. Consider the text in figure 2-4; namely:

THIS IS THE HOUSE THAT JACK BUILT

Process (a) implies that we may request, for example, that the key "HOUSE" be deleted. This would cause "HOUSE" to be squeezed from the text, and the thread pointing to the pointer to "HOUSE" to be deleted. If we had chosen to delete "HOUSE THAT J" then we would have deleted "HOUSE THAT J" from the text, and eliminated the keys:

"HOUSE THAT JACK BUILT."

"THAT JACK BUILT."

"JACK BUILT."

Process (b) is different only in that we give "coordinates" of starting and ending points to be deleted. The above example could be handled using process (b) by specifying that we wanted to delete the text over positions (13-24). The one important difference is that we could use process (b) to specify a single, non-key deletion, or a multiple-key deletion that in itself was not a key. Thus, we could specify that we shall delete (11-24), but if we invoked process (a) and searched for the key "E HOUSE THAT J" we would not succeed.

Process (c) may be accomplished by locating the duplicate keys and then re-applying process (b) until they have all been deleted. Hence, although process (c) is not a subset of process (b), it requires little more than a subtree traversal, where each "visit" means that a particular text position (containing the key of interest) is deleted. With the above in mind, we shall attack process (b), realizing that only minor modifications are required to effect processes (a) and (c).

4.1 Deleting Contiguous Text

Before getting into the text deletion process, it should be pointed out that the process is rather involved; in that sense, the examples of the previous section may have been misleading. To illustrate just how drastically a simple text deletion can alter the PATRICIA structure, consider the example given in figures 4-1, 4-2, and 4-3, where every occurrence of A, K, or Z is flagged as the start of a key. In figure 4-2, we have deleted the space. In figure 4-3, the fourth "A" has been deleted. (An explanation of how to interpret the minutiae of the printed

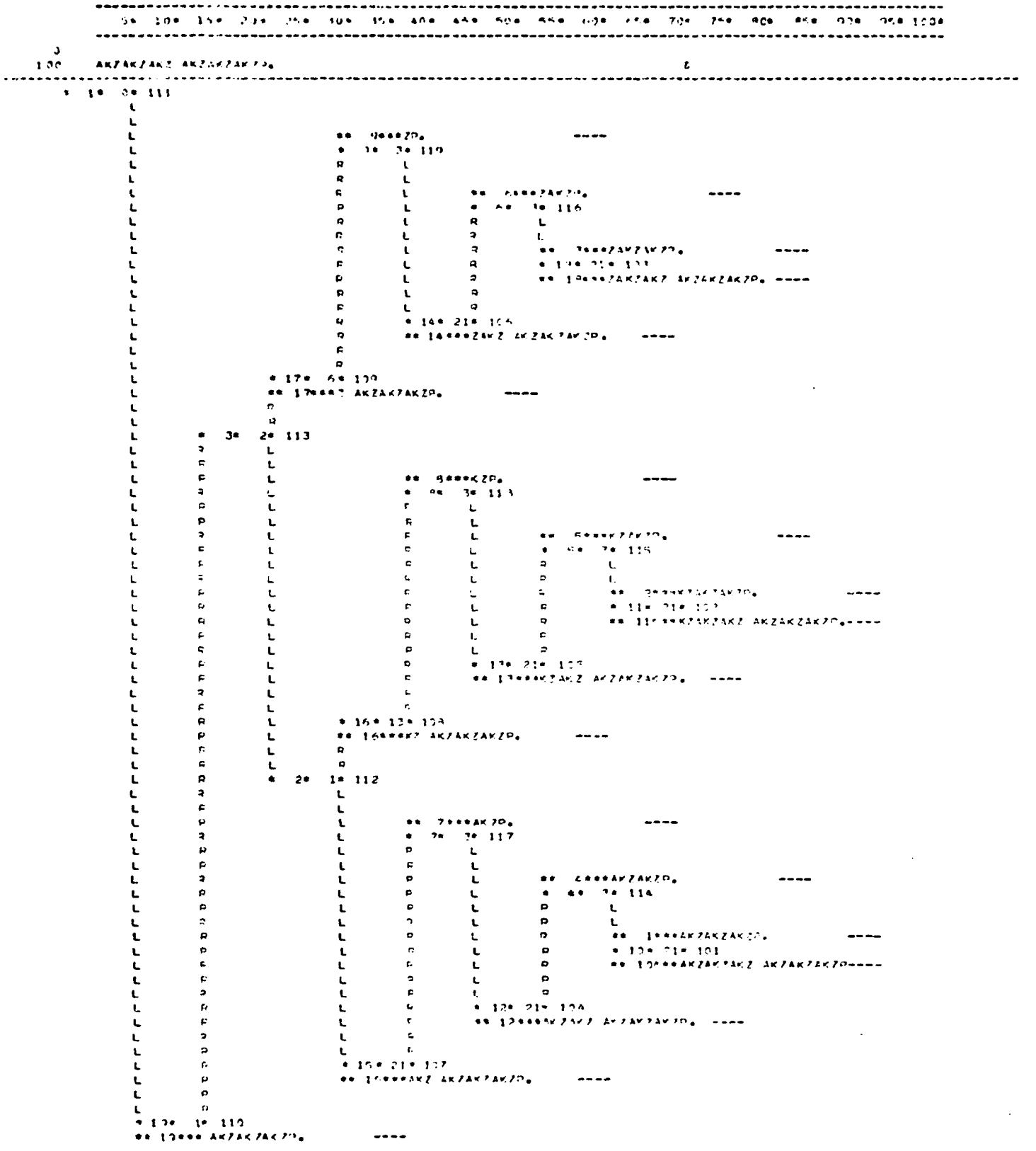


Figure 4-1. A strange tree. Every character in the text is a key.

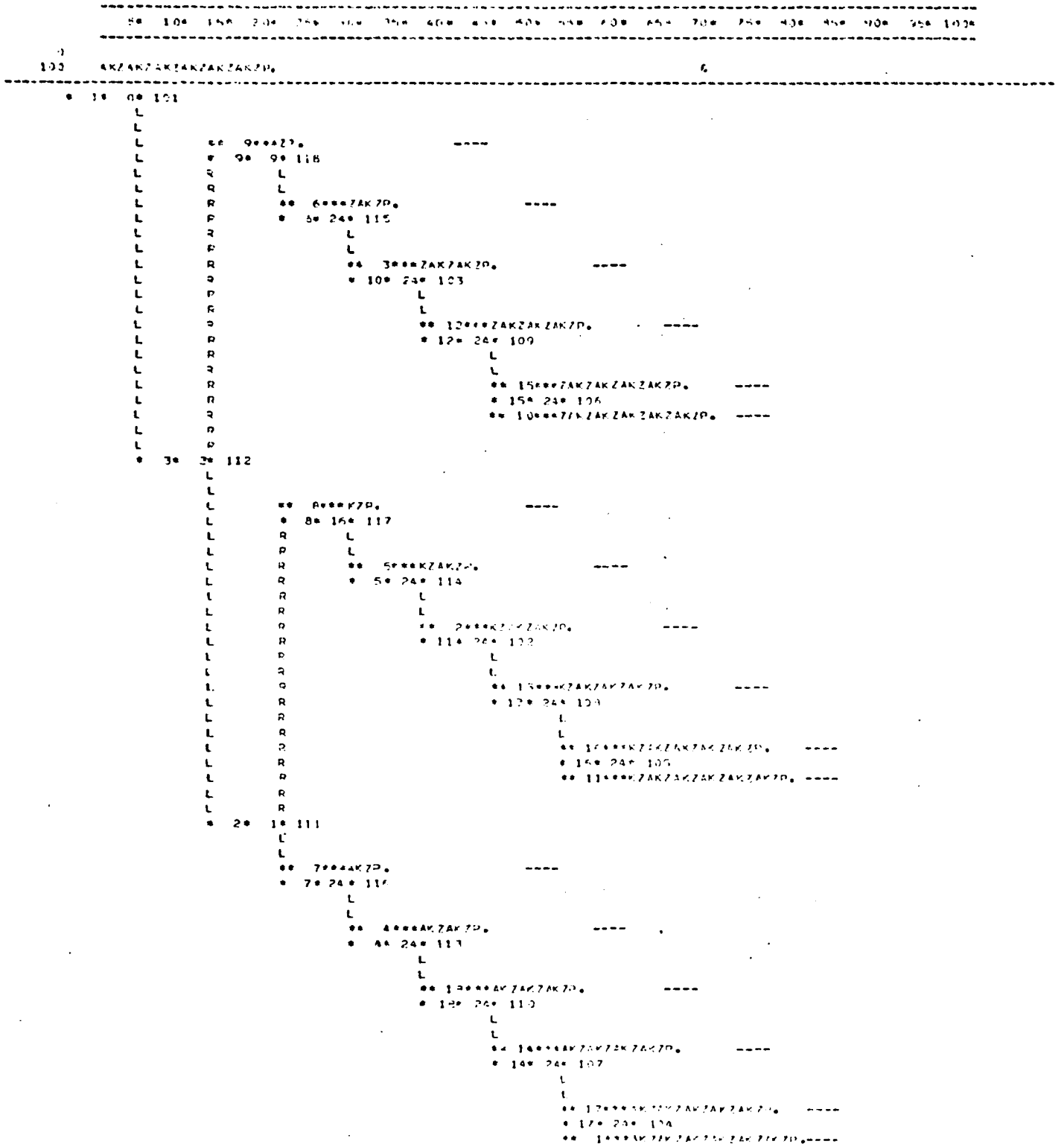


Figure 4-2. The space has been eliminated from the text of figure 4-1.

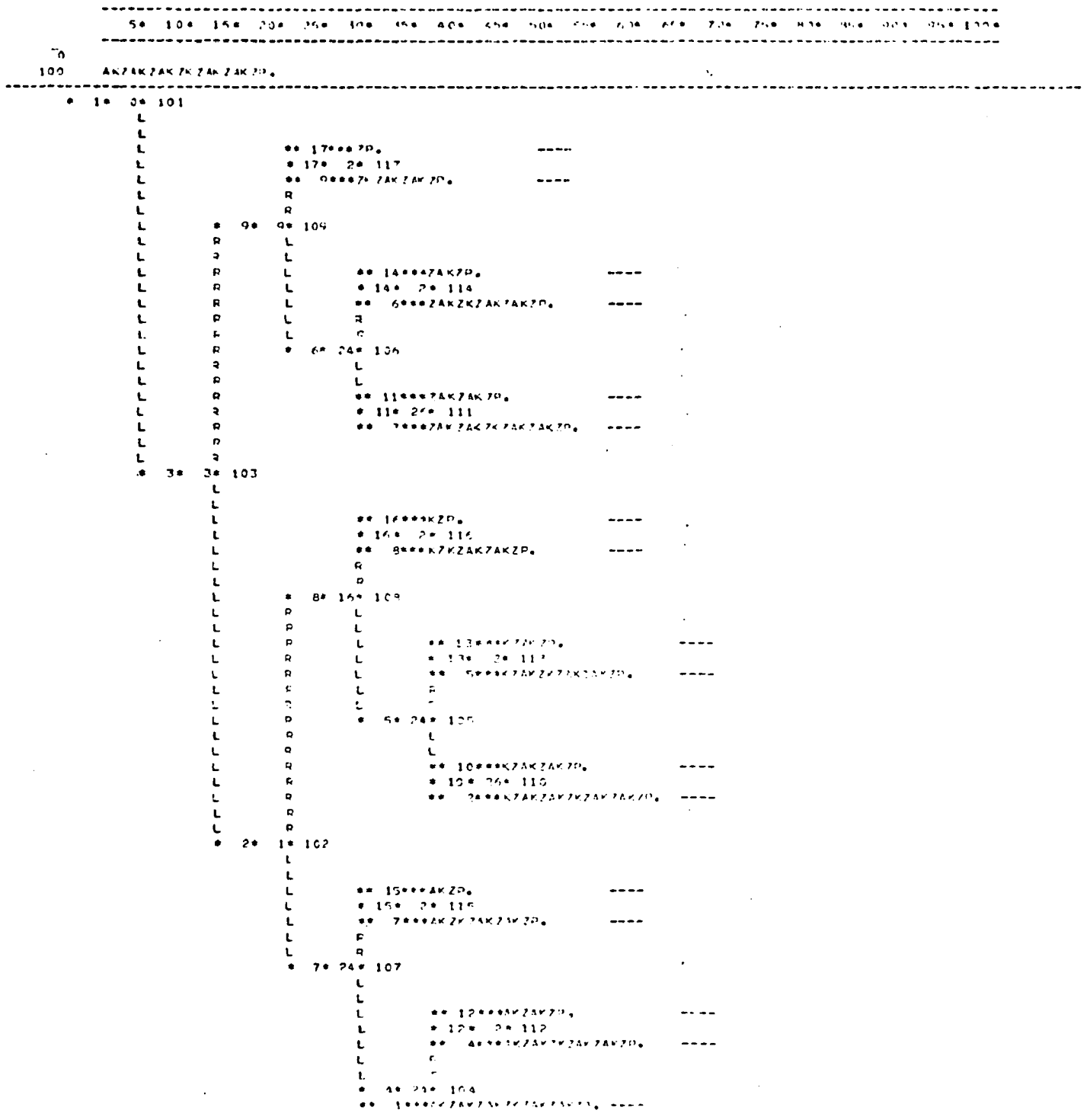


Figure 4-3. The fourth "A" has been eliminated from the text.

structure is given in Appendix B.) Most deletions will not cause such serious structural changes, but the reader should at least be aware of what could happen.

The reason that the text deletion process can cause such structural changes is that when we delete text, we also alter the bit comparison pattern for any key or keys that happen to extend over the deleted area. That is, any key not within the deleted area whose search path (as a result of successive skips) leads it into the deleted area, must be re-evaluated. Moreover, there is no way of predicting where the newly evaluated key will be positioned in the PATRICIA tree without actually re-evaluating the key. An example should help to illustrate this. Consider the rather contrived text of a figure 4-2:

AKZAKZAKZAKZAKZAKZ

where every letter is the start of a key. Notice that, in scanning from left to right (which is exactly what the search algorithm does), the first difference between the keys

AKZAKZAKZAKZAKZAKZP

and

AKZAKZAKZAKZAKZP

occurs at the comparison between "P" of the second key and the last "A" of the first. However, when we have the text shown in figure 4-3 (i.e., the fourth "A" has been deleted) then the same comparison would now be made between the keys

AKZAKZAKZKZAKZAKZP

and

AKZAKZKZAKZAKZP

Here the first difference occurs at the third "A" of the first key.

One may easily verify that all keys starting with "A", "K", or "Z" which originate ahead of the deleted "A" will be similarly affected! This amounts to an entire restructuring of the search path, which is obviously reflected in the vast difference between the PATRICIA trees of figures 4-2 and 4-3.

4.1.1 Concepts Behind Deleting Contiguous Text

Let START and END be pointers that point to character positions in the text. Assume that we are going to delete all text which occurs between (and including) the characters whose positions are given by START and END-1 (i.e., up to but not including END). We must look at every node in the PATRICIA tree and do the following. (Assume the node we are looking at is pointed to by X.)

1) Recalculate the proper address for the PTR field in the case where the PTR field points to text that was moved in order to fill in the space left by the deleted text. If X points to a node which requires such modification, then no other modification is required for the node. The recalculation is handled quite asily by the step:

   If PTR(X) $\geq$ END, set

   PTR(X) $\leftarrow$ PTR(X) - (START - END)

2) Delete any PTR field that points to a key which starts within the deleted area. This is done by the deletion algorithm, setting PD $\leftarrow$ X. (Note that in addition to PD, the deletion

algorithm must also find TD, FT, and TT.) The step to detect such a node is, quite simply:

> If START $\leq$ PTR(X) < END, then set PD ← X, call the deletion algorithm at the appropriate entry point.

Since step 1 checked for PTR(X) $\geq$ END, we can eliminate that part of the "if" statement and write simply:

> if PTR(X) $\geq$ START, then . . . . etc.

Note that such a node requires no further attention, i.e., once we have deleted it, we are done with it.

3) Fix any node whose search path has been altered as a result of the text deletion. This is done by deleting the old node, looking at the new bit pattern given by the new key, and re-inserting the new key. The step is given in simplified form as:

> if $\text{PTR(X)} + \frac{\Sigma \text{ SKIP fields leading to PTR(X)}}{\text{number of bits per character}}$
>
> is $\geq$ START, delete PTR(X); then reinsert into the tree the key which starts at PTR(X) in the text. (Remember, the text has already been concatenated.)

The rest of the nodes fall in the class where:

$$\text{PTR(X)} + \frac{\Sigma \text{ SKIP fields leading to PTR(X)}}{\text{number of bits per character}} < \text{START}$$

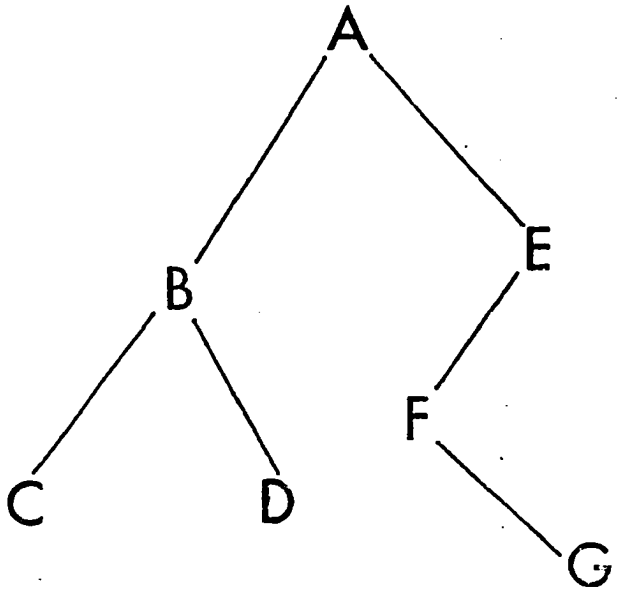
and which are therefore unaffected by the text that was deleted.

It would be nice if we could simply traverse the PATRICIA tree in endorder, altering, deleting, and reinserting nodes all with just one pass over the structure. Unfortunately, this is difficult to do, for several processes are taking place which are capable of dynamically

altering the tree in a manner that would either cause any traversal algorithm to fail, or would change the structure in such a way that the deletion algorithm wouldn't be able to function during the traversal process. For example, it is possible for the deletion algorithm to move a PTR field from a node to its ancestor. If this PTR field is one which must be modified to point to its proper text position, which was shifted by the textual concatenation (i.e., PTR $\geq$ END), then we must take care that the modification does not occur twice.

If we make the above modification during the "visit" of a preorder traversal, then we can perform deletions without worrying about accidentally altering a PTR field twice, for a preorder traversal visits all ancestors of a node before it visits the node. If any deletions to be made are made at this same visit, then any PTR fields being moved will be moved into areas already visited, and hence will not be molested any further. Unfortunately, other problems preclude making the necessary deletions during a preorder traversal.

For example, if we use algorithm 3.2.1 and encounter a type 2a deletion, node TT will not have been processed yet. The algorithm, which searches for the node by looking at the text, will not find it if node TT is a node being deleted or altered. The only way to insure that this situation will not occur is to visit all descendants first, which requires an endorder traversal. Hence, if we make our necessary deletions during the visit of an endorder traversal, and modify the necessary PTR fields during the visit of a simultaneous preorder traversal, we will avoid at least the difficulties indicated above. The traversals may be simultaneous since an endorder traversal visits a given node at either the same time or later, as illustrated in figure 4-4.



Node Access Interval	|	|	|	|	|	|	|	|	|	|	|	|	|	|
Preorder Visit	A	B	C		D			E	F	G			
Endorder Visit			C		D	B				G	F	E	A

Figure 4-4. Preorder and endorder visits. A preorder visit always occurs at the same time or before an endorder visit.

Some problems still remain, however. Any node whose search path was altered may, of course, be deleted during the traversal. However, we must be careful when we reinsert the node, since it could wind up anywhere in the structure. This obviously could be undesirable; if the node were inserted in a place that had not been visited, then it would be deleted and reinserted again! There is an easy way of avoiding this problem. We can make a list of all the nodes being altered, linking them up by their LLINK or RLINK fields as we delete them. Then, after the traversal has been completed, we can reinsert the nodes by looking at the PTR fields of the nodes we put in our list. These pointer fields give the starting text positions for the keys whose search paths must be recomputed.

4.1.2 Algorithm: Delete Contiguous Text

The complete algorithm for deleting text is now presented. It is broken up into three subalgorithms. Initially, set pointer TOPLIST $\leftarrow \lambda$; SHIFT $\leftarrow$ END - START. Assume that the text between START and END - 1 has been deleted, the remainder having been concatenated.

Subalgorithm T.0 Traversal

This subalgorithm traverses the tree in preorder and endorder. If both visits occur at the same time, the preorder visit will be done first. The algorithm uses a stack, A.

Input: START, END, SHIFT.

Output: The updated PATRICIA tree.

1) Set Σ SKIP $\leftarrow$ 0, ATOP $\leftarrow$ 0, X $\leftarrow$ pointer to root of PATRICIA tree.

2) Set Σ SKIP $\leftarrow \Sigma$ SKIP + SKIP(X)
ATOP $\leftarrow$ ATOP + 1, A(ATOP $\leftarrow$ X
(Preorder visit) Preform subalgorithm T.1.

3) IF LLINK(X) is not a thread, (i.e.: > 0)
set X $\leftarrow$ LLINK(X); go to step 2

4) Set X $\leftarrow$ A(ATOP), ATOP $\leftarrow$ ATOP-1

5) If X > 0, set ATOP $\leftarrow$ ATOP+1,
A(ATOP) $\leftarrow$ -X go to step 7, else set X $\leftarrow$ - X

6) (Endorder visit). Perform subalgorithm T.2, then set
Σ SKIP $\leftarrow \Sigma$ SKIP-SKIP(X)
(even though node X may have been deleted, its SKIP field is still intact); go to step 4.

7) If X=1 (if we are back at the root), exit.

8) If RLINK(X) is not a thread, set X $\leftarrow$ RLINK(X), go to step 2, else go to step 4

Subalgorithm T.1

This subalgorithm adjusts PTR fields when they point beyond the area of text that has been deleted. It also marks those PTR fields which are to be completely deleted. (The actual deletion must take place later.)

Input: X, START, END, SHIFT

Output: PTR(X) adjusted, or set to zero if node X is to be deleted

1) If PTR(X) $\geq$ END, set PTR(X) $\leftarrow$ PTR(X) - SHIFT, exit T.1

2) If PTR(X) $\geq$ START, set PTR(X) $\leftarrow$ 0, exit T.1

Subalgorithm T.2

This subalgorithm deletes nodes marked by T.1. It also deletes those nodes whose search path must be recomputed, and saves them in a list. If a deletion is to occur, the algorithm calls on the node deletion algorithm (3.2.1); it will not call the node deletion algorithm twice for the

same node, as this could foul up the traversal process. If a node contains 2 threads, and both are to be deleted, then the second deletion will occur at the visit to the parent of X (which will contain the other unwanted thread after the deletion of the first thread).

Input: X, START, END, SHIFT.

Output: The node at X is deleted, or the search path of its key is recomputed, or node X is left alone.

1) (The subalgorithm uses a flag, FINISHED, to force an exit after LLINK and RLINK are checked.)
Set FINISHED ← 0
If LLINK(X) is a thread, set PD ← |LLINK(X)|, go to step 4.

2) If FINISHED = 1 exit T.2, else set FINISHED = 1
(exit if we have looked at both link fields)

3) If RLINK(X) is a thread, set PD ←|(RLINK(X)| go to step 4, otherwise exit T.2.

4) (PD gives the pointer we might want to delete. First we must check it.)
If PTR (PD) $\geq$ START, go to step 2
(no modification necessary, since the key lies beyond the affected area)

5) If PTR(PD) =0, set TD ← X, FT ← |A(ATOP)|
(the top node of the stack is the father of the node being visited in endorder) call the node deletion algorithm at step 3 of algorithm 3.2.1. Then exit T.2

6) IF PTR(PD) + $\frac{\Sigma \text{ SKIP}}{\text{number of bits per character}}$
is < START, go to step 2 (the text deletion did not affect the search path to node X)

7) Set FT ← |A(ATOP)| , TD ← X,
call the deletion algorithm at step 3, algorithm 3.2.1 but instead of returning node X to free storage, save it (to be reinserted) by:

8) LLINK(X) ← TOPLIST, TOPLIST ← X. Then exit T.a.

After exiting from subalgorithm T.0 we merely reinsert all the nodes whose search path was altered; namely the nodes in the list pointed to by TOPLIST.

END

4.1.3 Algorithm: Delete a specific key from the text

The algorithms presented in the previous section may be used to perform a variety of functions. For example, to effect a"process (a)" text deletion (i.e., the key is given) we employ the brief algorithm below.

Input: The key we wish to delete.

Output: An occurrence of the key is deleted from the text.

1) Set K ← key, N ← number of bits in K, call algorithm 2.1. There will be a match at node P.

2) Set START ← PTR(P), END ← START + number of characters in K.

3) Concatenate the text by bringing together locations START - 1 and END (effectively deleting positions START through END - 1)

4) Set SHIFT ← END-START.

5) Call algorithm 4.1.2 T.0.

6) Use algorithm 2.2 to reinsert the keys whose PTR fields are given in the list created by algorithm 4.1.2 T.2 step 8.

END

A "process (b)" deletion is even easier, since we are already given START and END - simply enter algorithm 4.1.3 at step 3.

4.2 Algorithm: Insert Text

The process of inserting new text is practically identical to the process of text deletion. Keys whose search paths are changed must have their search paths recomputed, and PTR fields pointing to text which has been shifted (in this case to make room for the new text) must be altered. The only difference is that no old keys are deleted, for no text is deleted. The text insertion process is given below. Assume we are given the information TEXT, which we want to add, along with START,

the position at which the next text is to be inserted.

Input: START, TEXT, number of characters in TEXT

Output: The updated PATRICIA tree, the updated text, exit to algorithm 4.1.3, step 5.

1) Set END ← START (this guarantees that subalgorithm T.1 will not delete any keys.

2) Break apart the text at position START-1 and insert TEXT.

3) Set SHIFT (← the number of characters in TEXT)
(The net result will that subalgorithm T.1 will add SHIFT to all PTR fields that point beyond and including position START)

Now, simply enter algorithm 4.1.3 at step 5.

END

4.3 Conclusions - Altering Text

The user now has a complete set of working algorithms that enable him to update or alter either the PATRICIA tree or the text. The algorithms presented in Chapter Four can:

a) Delete any area of the text (algorithm 4.1.2).

b) Delete any specific key from the text (algorithm 4.1.3).

c) Insert new text anywhere (algorithm 4.2).

5.0 Alternate Methods for Representing the PATRICIA Tree - Compressing the Structure.

In the previous two chapters, we developed the algorithms necessary to effect most of the operations that a user might encounter when working on an interactive basis with PATRICIA. This section will be concerned with alternate methods of prepresenting the PATRICIA tree which will result in more efficient use of memory, or more rapid retrieval of information, or both.

5.1 A Right Threaded PATRICIA Tree

The backward links of the PATRICIA tree do not resemble ordinary "threads" (as explained in Knuth 1968, p. 320) in that they do not necessarily point to nodes which are postorder predecessors or successors. If we could build a PATRICIA tree in a way such that the backward pointers were "threaded," then we could compress the structure and thereby save considerable storage space. Let us focus our attention on the possibility of building a right threaded PATRICIA tree, where such a structure is defined below.

Definition 5-1.

A right threaded PATRICIA tree is a PATRICIA tree in which the backward-pointing left links point to themselves, and the backward pointing right links point to their postorder successors. An example of such a structure is given in figure 5-1, where the tree is made up of keys that

```
                    THE STRUCTURE IS RIGHT-THREADED
-------------------------------------------------------------------------------
      5*  10*  15*  20*  25*  30*  35*  40*  45*  50*  55*  60*  65*  70*
-------------------------------------------------------------------------------
  0
100    THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG.$
-------------------------------------------------------------------------------
    *  1*  0* 101
           L
           L
           L        **  1***THE QUICK BROWN FOX ----
           L        *  7* 34* 133
           L        **  7***THE LAZY DOG.$       ----
           L        R
           L        R
           *  2*  3* 105
                    L
                    L
                    L        **  2***QUICK BROWN FOX JUMP----
                    L        *  5*  1* 128
                    L        R        L
                    L        R        L
                    L        R        **  5***OVER THE LAZY DOG.$ ----
                    L        R        *  6*  1* 137
                    L        R                 L
                    L        R                 L
                    L        R                 **  6***LAZY DOG.$          ----
                    L        R                 *  8*  1* 121
                    L        R                 **  3***JUMPED OVER THE LAZY----
                    L        R
                    L        R
                    *  3*  1* 117
                             L
                             L
                             L        **  3***FOX JUMPED OVER THE ----
                             L        *  9*  1* 142
                             L        **  9***DOG.$                ----
                             L        R
                             L        R
                             *  4*  2* 111
                             **  4***BROWN FOX JUMPED OVE----

. . . . . .
```

Figure 5-1. An example of a Right Threaded PATRICIA tree. Every LLINK thread points to the originating node and every RLINK thread points to the postorder successor (the node on the line above).

begin at every word of the sentence:

THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG

Note that a thread always points to the first node encountered, moving up the page. The usual, or "non-right threaded" structure, which is built by inserting keys in the order in which they are scanned (going from left to right), is shown in figure 5-2.

Observe that when we build the structure with algorithm 2.2, the positioning of the backward pointers is determined by the order in which we insert the keys; thus, we could have attained our right threaded structure by inserting the keys in the order:

THE QUICK . . .

THE LAZY . . .

QUICK . . .

OVER . . .

LAZY . . .

JUMPED . . .

FOX . . .

DOG . . .

BROWN . . .

which is simply reverse lexicographic order. Figure 5-3 shows the structure which results when the keys are inserted in the order given above, using algorithm 2.2.

Unfortunately, we cannot always insert the keys in reverse lexicographic order; in fact, we practically never can efficiently. Thus, what we need is a new node insertion algorithm which will insure that when we insert a new node, the "right-thread" property will be preserved.

```
         ----------------------------------------------------------------------
            5*   10*   15*   20*   25*   30*   35*   40*   45*   50*   55*   60*   65*   70*
         ----------------------------------------------------------------------
    0
  100     THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG.&
---------------------------------------------------------------------------------
       *   1*   0* 101
                 L
                 L
                 L          **   1***THE QUICK BROWN FOX ----
                 L          *   7*  34*  133
                 L          **   7***THE LAZY DOG.&        ----
                 L          R
                 L          R
                 *   2*    3* 105
                            L
                            L
                            L          **   2****QUICK BROWN FOX JUMP----
                            L          *    5*    1*  121
                            L          R          L
                            L          R          L
                            L          R          **   6****OVER THE LAZY DOG.& ----
                            L          R          *    6*    1*  128
                            L          R                     L
                            L          R                     L
                            L          R                     **   8****LAZY DOG.&          ----
                            L          R                     *    8*    1*  137
                            L          R                     **   5****JUMPED OVER THE LAZY----
                            L          R
                            L          R
                            *   3*     1*  111
                                       L
                                       L
                                       L          **   4****FOX JUMPED OVER THE ----
                                       L          *    9*    1*  142
                                       L          **   9****DOG.&                ----
                                       L          R
                                       L          R
                                       *    4*    2*  117
                                       **   3****BROWN FOX JUMPED OVE----

. . . . . .
```

Figure 5-2. The tree of figure 5-1 as built by algorithm 2.2.

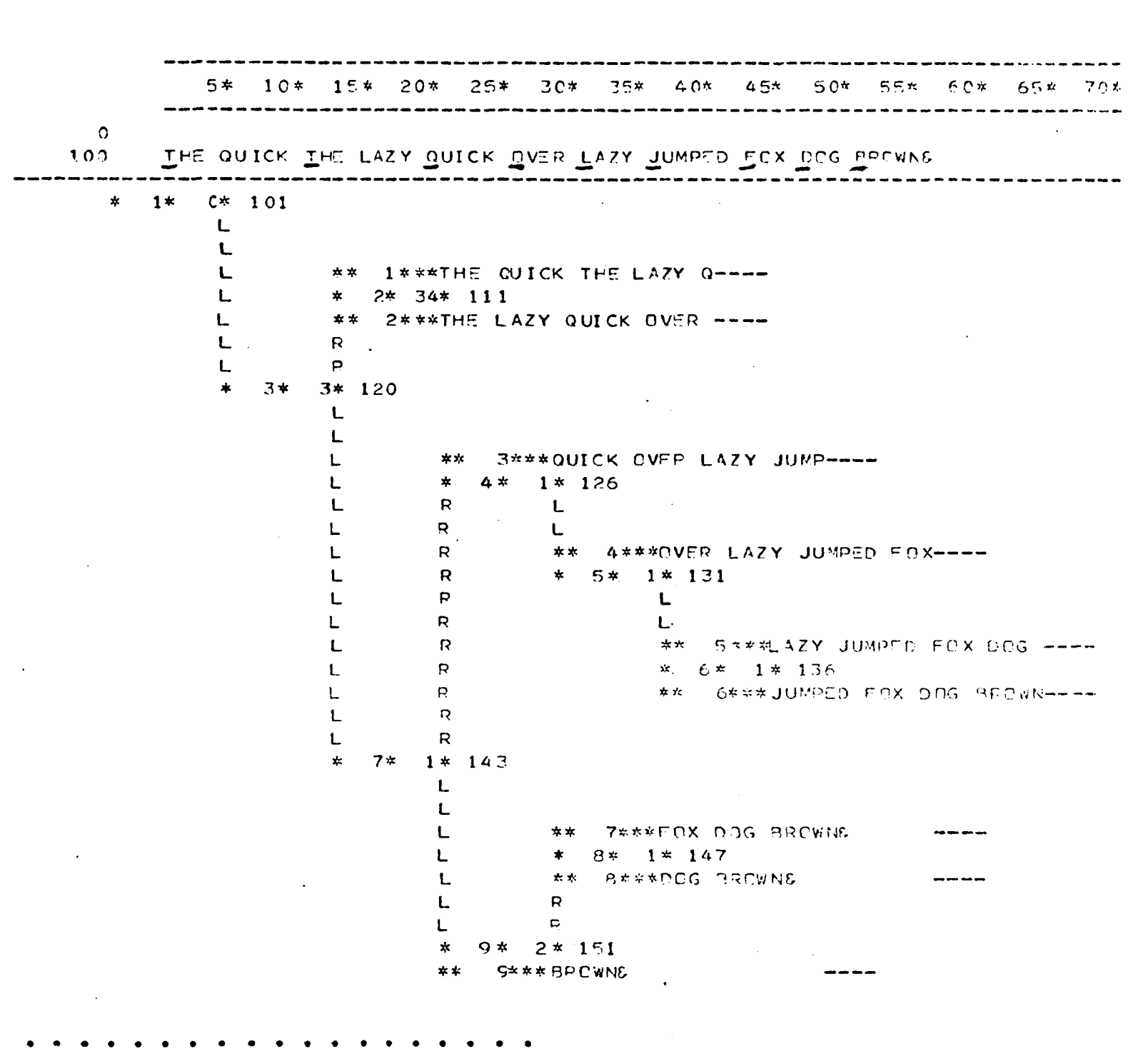


Figure 5-3. If we use algorithm 2.2 and insert the keys in reverse lexicographic order we get a Right Threaded tree. The keys have been underlined in the text.

If we look at algorithm 2.2, we see that the point where the threads are created occurs at step 5. If we insert a node, X, whose L+1st bit is zero, into a right threaded structure, then this step will cause the LLINK field to point back to X. The value of the RLINK field will depend upon which type of structure the node belongs to. The four types (where the L+1st bit = 0) are given below. Let the father of X be denoted by FX.

Type 1. X hangs from the left link of FX, and has no right subtree.

Type 2. X hangs from the right link of FX, and has no right subtree.

Type 3. X hangs from the left link of FX, and has a right subtree.

Type 4. X hangs from the right link of FX, and has a right subtree.

(Remember, for all these types, the LLINK field of X points back to N itself.) After making an insertion, the following structural changes will have taken place.

For type 1, RLINK(X) will point to the same place that LLINK(FX) previously pointed to, namely FX itself. Thus, RLINK(X) will point back to FX, which is the postorder successor of X.

For type 2, RLINK(X) will point to the same place that RLINK(FX) previously pointed to, namely its postorder successor. But now, FX's postorder successor has become X's postorder successor (X having been inserted to the right of FX).

For types 3 and 4, RLINK(X) will point to the root of X's subtree, and the "rightmost" node of this subtree (the last node traversed in postorder) will still point to its postorder successor, even after we have inserted node X (which is actually the postorder predecessor to the subtree).

Hence, types 1-4 are correctly handled automatically by algorithm 2.2 (assuming, of course, that the structure is right threaded before X is inserted). The real problem occurs when the L+1st bit of X is "1", and hence algorithm 2.2 would try to set RLINK(X) = X, which is forbidden. Again, we have four types of structures to be concerned about.

Type 5. X hangs from the left link of FX, and has no left subtree.

Type 6. X hangs from the right link of FX, and has no left subtree.

Type 7. X hangs from the left link of FX, and has a left subtree.

Type 8. X hangs from the right link of FX, and has a left subtree.

For type 5, step 5 of algorithm 2.2 would cause LLINK(X) to point to FX, and RLINK(X) to point to X. What we actually want is the reverse of this, namely, RLINK(X) should point to FX and LLINK(X) should point to X. We may alter the LLINK-RLINK fields in this manner, but then we must swap the PTR fields of X and FX.

Type 6 is handled just like type 5, except that RLINK(X) should point to the node that RLINK(FX) pointed to. Again this requires that the PTR fields of X and of the node pointed to by RLINK(FX) must be swapped.

For types 7 and 8, we must search for the node in the subtree which points to the postorder successor of the subtree. For type 7, this postorder successor would be FX, and for type 8, it would be some ancestor of FX. At any rate, this search is easily and rapidly accomplished by going down the RLINK fields of the subtree until a backward thread is encountered. Again, PTR fields must be swapped; this time between X and either FX (type 7) or the ancestor we found (type 8).

Moreover, the backward thread which we found must be altered so that it points to X, which has become the postorder successor to its subtree.

The average number of searches required to find the postorder successor of the subtree in types 7 and 8 is easily estimated for a balanced tree of $P=2^n - 1$ nodes. (A "search" in this case means simply an inspection of the node to see if it has a backward RLINK.) For such a tree, the average number of searches required to access any node is well known to be approximately $\log_2 P - 1$ (Salton, 1968, p. 72). But the maximum search path length, which is also the path length to all terminal nodes, is $\log_2 P$. Hence, the average remaining number of searches is the same as the remaining path length, or $\log_2 P - 1) = 1$. Since we are only performing this extra search when a node has an L+1st bit equal to "1", then the average number of extra searches required to build a Right Threaded PATRICIA tree is approximately:

$$\frac{\text{number of nodes in the tree}}{2}$$

which is obviously an insignificant additional cost. (This figure has been veritifed for several cases where n = 9 and 10.)

5.1.1 Algorithm: Create a Right Threaded PATRICIA Tree

The brief algorithm given below will accomplish the threading process. It is inserted in place of step 5 of algorithm 2.2.

Input: See algorithm 2.2.

Output: The updated Right Threaded PATRICIA tree

Step 5. If the L+1st bit of K = 0, set LLINK(R) ← - R, RLINK(R) ← P*SIGN(T), go to step 6, otherwise set Y ← P*SIGN(T), Z ← R.

5-1 If Y > 0 then set Z ← Y, Y ← RLINK(Y), repeat; otherwise, swap PTR(|Y|) and PTR(R), then set RLINK(Z) ← -R.

5-2 Set RLINK(R) ← Y, then if T>0 set LLINK(R) ← P, otherwise set LLINK(R) ← -R.

Step 6. Etc.

END

The algorithm is quite short, and since the additional cost is so slight, a PATRICIA tree should always be Right Threaded. For one thing the Right Threaded structure is a standardized structure. More importantly, however, is that it allows us to compress the PATRICIA structure considerably, as is seen in the following sections.

5.1.2 An Important and Immediate Consequence - Eliminating LTAG

Algorithm 5.1.1 is used with the version of Algorithm 2.2 that represents LTAG, RTAG by the sign bit of LLINK, RLINK. It would have been just as easy to construct a version that could be used with the LTAG-RTAG version of algorithm 2.2. Note however that if the tree is Right Threaded, we may immediately elliminate the LTAG field (or the sign bit of LLINK); for, if we are given a pointer, X, that points to a node in a Right Threaded PATRICIA tree, then the test

If LTAG(X) = 1

Becomes simply

if LLINK(X) = X

in which case, we know LLINK(X) is a thread.

On the other hand, we could just as easily have eliminated the LLINK field whenever LLINK is a thread. In this case, we let LTAG(X) = 1

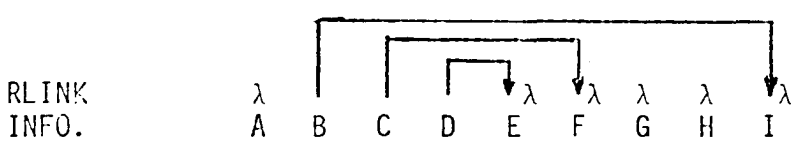
indicate that LLINK(X) = X; and when LTAG(X) = 0, then LLINK(X) points to a descendent. The real importance, however, of the right threaded structure, will be seen in the next section.

5.2 Preoder Sequential Representation

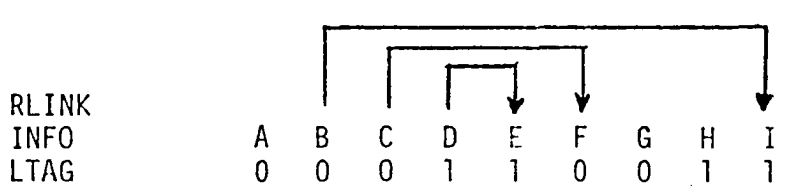
Consider the tree illustrated in figure 5-4. If we write out the nodes in preorder, we get:

A B C D E F G H I

Now, let's associate with each node a RLINK field whose value is λ if the node has no right subtree, and otherwise points to the right subtree of the node. We indicate this field with arrows and "λ"s below.



To complete the representation, note that if a node has a left subtree, its root is immediately to the right of its parent node. Obviously, every node (except the last) has another node to its right. Hence, we must differentiate between nodes with left subtrees and those without. To do this, we use the LTAG field; if LTAG(X) = 0, then X+1 points to the left subtree of X and if LTAG(X) = 1, then X has no left subtree. The complete representation, which is called Preorder Sequential form (Knuth, 1968), is shown below (the "λ"s have been omitted, and non-null RLINKS are indicated by arrows.)



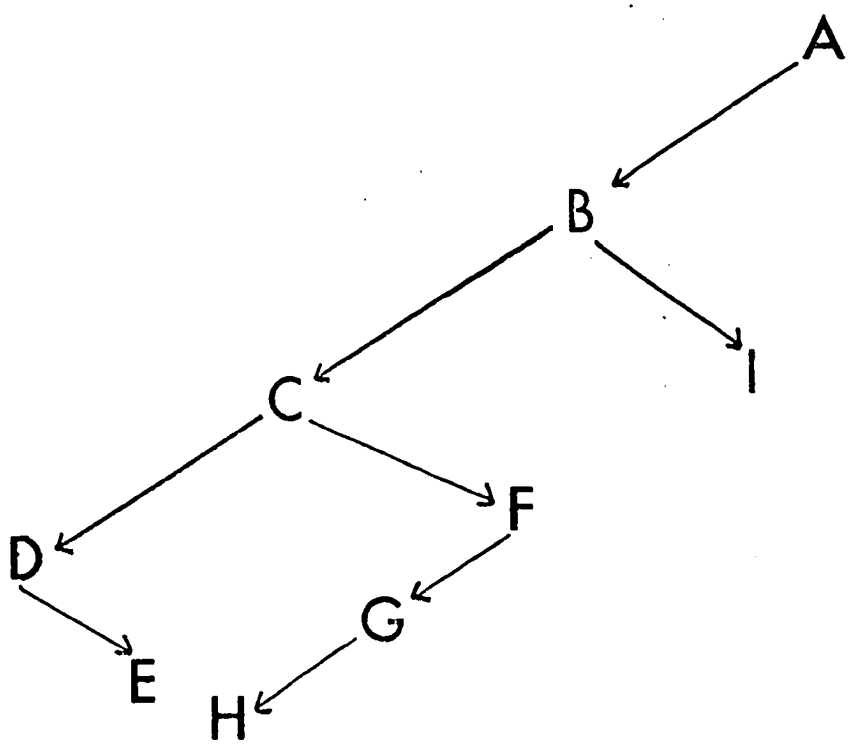


Figure 5-4. A Binary Tree

5.3 Preorder Sequential PATRICIA

The technique just illustrated may be utilized to construct a very compact yet efficient PATRICIA structure. If we assume that the tree is Right Threaded, then we may create the Preorder Sequential representation, in which an LTAG field of 1 means that the node has a left thread which, of course, must point to the node itself. Now, the problem remains: how do we indicate a right thread? We could introduce an RTAG field, and let"RTAG(X) = 1" mean that RLINK(X) was a thread. This is unnecessary, however. Recall that a preorder traversal will visit all ancestors of a given node before the node itself is visited. Hence, a right thread of node X, which always points to an ancestor of X, has the property:

$$RLINK(X) < X$$

that is, it is numerically lower than the address of node X. Thus, if RLINK(X) > X, then RLINK(X) points to the right subtree of X; otherwise RLINK(X) is a backward thread which points to the postorder successor of X.

The tree of figure 5.1 (which incidentally is identical in form to the tree of figure 5-4) has been converted to preorder sequential representation in figure 5-5. Note that the root (node #1) never has a right subtree; this is always indicated by setting RLINK(1) = 0.

5.3.1 Algorithm: Traverse a Preorder Sequential Structure in Postorder

In order to effect many of the PATRICIA algorithms, particularly to find all occurrences of a given key, it is necessary to traverse a portion of the tree in postorder. The algorithm below accomplishes this for a Preorder Sequential structure. Assume that we wish to traverse the subtree at X.

Physical Node Location	SKIP	PTR	LTAG	RLINK	TEXT (indicated by PTR)
1	0	101	0	0	THE QUICK BROWN - - -
2	3	105	0	9	QUICK BROWN FOX - - -
3	1	117	0	6	FOX JUMPED OVER - - -
4	2	111	1	5	BROWN FOX JUMPED - - -
5	1	142	1	3	DOG
6	1	128	0	2	OVER THE LAZY - - -
7	1	137	0	6	LAZY DOG
8	1	121	1	7	JUMPED OVER THE - - -
9	34	133	1	1	THE LAZY DOG

Figure 5-5. The Preorder Sequential representation for the PATRICIA tree of figure 5-1. Note that the text contains 100 leading blanks.

Input: X (a pointer to the root of the subtree we wish to traverse).

Output: A postorder visit to all nodes in the subtree whose root is at X.

1) Set Y ← X

2) (Traverse left) if LTAG(Y) = 0, set Y ← Y+1, go to step 2, else go to step 3.

3) Visit node Y.

4) Set Z ← RLINK(Y).If Z<X, exit (we are done).

5) If Z<Y, set Y ← Z, go to step 3, otherwise set Y ← Z, go to step 2.

END

This algorithm has several nice properties. For one thing, it done not require the use of a stack. Moreover, the algorithm may be used to find the postorder successor of any node. Simply set Y ← location of the node, X ← 1, and enter the algorithm at step 4; the first visit is the postorder successor to node Y.

The algorithm was applied to the structure in figure 5-5; the PTR fields of the nodes in the order they were visited are given below. (Initially, set X ← 1 since we are traversing the entire structure.)

Physical location	PTR field
4	111
5	142
3	117
8	121
7	137
6	128
2	105
9	133
1	101

5.3.2 Can the Structure be Utilized?

The apparent advantage gained by squeezing away the LLINK and RTAG fields is merely academic unless we can utilize the structure. Unfortunately, it is extremely difficult to build or alter a tree in Preorder Sequential form, for whenever we insert (or delete) a node, we must linearly shift part of the structure up (or down) and then pass over the entire structure in order to fix up the RLINK fields that refer to the shifted area.

On the other hand, postorder traversal is, as we have seen, quite nicely handled. Moreover--and much more importantly--we can effectively perform a PATRICIA search through the structure by altering algorithm 2.1. This has been done in the algorithm presented below, which is slightly faster than algorithm 2.1 since the LLINK subscript has been eliminated. If we can then find a way of efficiently handling alterations to the structure, we will have established its practicality. This process is discussed in section 5.3.4. First we present the search algorithm.

5.3.3 Algorithm: Search a Preorder Sequential Structure for a Given Key

This algorithm very closely resembles algorithm 2.1. The same explanatory remarks apply (section 2.1.1).

Input: K, the number of bits in K (see algorithm 2.1).

Output: P (a pointer to the root of a subtree containing all matches to K).

1) Set P ← 1, J ← 0, N ← number of bits in K.

2) Set Q ← P. If LTAG(P) = 1, go to step 6 else set P ← P+1 (the left subtree of P is the next sequential node)

3) Set J ← J+SKIP(P). If J > N, go to step 6,

4) If the J+1st bit of K=0, go to step 2.

5) Set Q ← P, P ← RLINK(P). If P>Q go to step 3 (otherwise, RLINK(Q) is a thread).

6) Compare K to the key in the text pointed to by PTR(P).

END

5.3.4 How to Handle Modifications

As has been pointed out, altering a Preorder Sequential structure requires at least one pass over the structure for each alteration, and hence, would be rather slow if we had several changes to make. A violent example is shown by the process of deleting the "A" from the structure illustrated in figure 4-2, thus getting the structure shown in figure 4-3. Practically every node had to be deleted and reinserted; if this were done with the Preorder Sequential form, then each node deletion could require a pass over the entire structure, the reinsertion would require another pass, and this would be repeated for every node we had to delete and reinsert--clearly an inefficient process.

A better solution is not to do any altering at all. Instead, make all alterations to the full blown Right Threaded structure; then when all alterations have been made, make one pass over the Right Threaded structure to convert it to the Preorder Sequential structure. This can be easily accomplished if we have an area in main memory large enough to hold both structures. We simply traverse the large structure in preorder and as we visit the nodes, we place them sequentially into the new structure. The algorithm below will do this.

5.3.5 Algorithm: Convert a Right Threaded PATRICIA Tree to a Preorder Sequential PATRICIA Structure

Assume storage space is available for both structures. We shall differentiate between the two with a single quote mark (') thus, LLINK(X) refers to a node in the Right Threaded tree, and LLINK'(X) refers to a different node, occupying different memory, in the Preorder Sequential structure. The algorithm does not alter the Right Threaded structure, and uses an auxillary stack.

Input: A Right Threaded PATRICIA tree whose root is at location one.

Output: A Preorder Sequential structure.

1) Set I ← 0, stack the number zero,
set RLINK(1) ← 1. Traverse the right
threaded structure in preorder and postorder.
At each preorder visit to X, perform step 2
and at each postorder visit (which, of course,
comes later) perform step 3. After completing
the traversal, exit.

2) (Preorder visit)
set I ← I+1, SKIP'(I) ← SKIP(X)
PTR'(I) ← PTR(X), stack I.
if LLINK(X) = X, set LTAG'(I) ← 1,
else set LTAG'(I) ← 0.

3) (Postorder visit)
pop stack into J. if RTAG(X) = 1,
set RLINK'(J) ← value
currently on top of stack, else,
set RLINK'(J) = I+1

END

5.3.6 Converting Over the Same Memory Space

The above algorithm works well enough; unfortunately, it is not very practical. For it we had plenty of memory, we wouldn't need to use the more compact Preorder Sequentical form to start with. Probably the right threaded PATRICIA tree will take up all available memory;

hence, if we are going to convert to Preorder Sequential form, then the conversion must be done directly over the threaded structure, which of course is then sacrificed. Unfortunately, the conversion cannot be done in one pass, for as we move a node to its sequential location we must swap it with the node formerly contained in the sequential location, and we have no idea where the father of this node is. (We must locate the father so that we can fix up the appropriate RLINK or LLINK field).

We can, however, effect the conversion in 2 passes. The first pass forms a doubly linked list of the nodes in preorder and uses the LTAG and RTAG fields to indicate whether a node has a left or right subtree.[1] The second pass then recreates the proper value for the RLINK field, and swaps nodes when they are out of physical sequence. The rationale behind using the doubly linked structure is that it allows us to move nodes around with no difficulty.

5.3.7 Algorithm: Convert to Preorder Sequential form over the Same Memory Space.

The two passes of the algorithm are presented below. Both use a stack, A, and pointers, I, J, W, X, Y, Z, ATOP, and B. This is the only storage requirement outside of the threaded structure.

Input: A Right Threaded PATRICIA tree whose root is at location one.

Output: A Preorder Sequential PATRICIA structure starting at location one.

---

[1]This is a linked version of the linear representation given in Knuth (1968) p. 359, exercise 2. It is perhaps the most compact form possible, since both RLINK and LLINK are eliminated. Unfortunately, except for the additional space saved, this form has little practical value here, for it cannot be searched efficiently. It might be useful for storing extremely compressed structures, if auxilary bulk storage were at a premimum. Usually, however, this is not the case.

Pass One

1) Set ATOP ← 0, B ← 0, X ← 1.

2) If RTAG(X) = 1 (If X has no right subtree)
go to step 3, otherwise
set ATOP ← ATOP+1,
A(ATOP) ← RLINK(X) (stack the right subtree of X)

3) Set RLINK(X) ← B, B ← X (RLINK(X) now
points to the preorder predecessor)

4) If LLINK(X) ≠ X (If X has a left subtree)
set LTAG(X) ← 0, X ← LLINK(X)
Go to step 2.

5) Set LTAG(X) ← 1 (Now get the preorder successor,
if there is one, and point LLINK(X) to it)
If ATOP = 0 then set LLINK(X) ←1, RLINK(1) ←X,
exit. Otherwise set Z ← A(ATOP), ATOP ← ATOP-1, LLINK(X) ← Z,
X ← Z, go to step 2.

END

If we apply the above algorithm to the tree of figure 5-1 we will get the structure shown in figure 5-6. The nodes are more easily recognized by the first few letters of the key pointed to by the PTR field, and the LLINK-RLINK fields are indicated by arrows.

We now make the second pass over the structure with the algorithm given below. It will recreate the proper RLINK field at the same time it is forming the preorder Sequential representation.

Pass 2

1) (initialize)

Set I←1, ATOP←1, A(ATOP)←0 Got to Step 3

2) (Linearize and go to next node)

Set J←LLINK(I), I←I+1. If I≥J

go to step 3, else

set RLINK(LLINK(I))←J

LLINK(RLINK)I))←J

swap all fields of nodes I and J

3) If LTAG(I) ≠1, set ATOP ATOP+1,
A(ATOP)←I (stack the node and continue down the left subtree) go to step 2.

4) Set J←I (We are going to find the proper value for RLINK(J)

5) If RTAG(J)≠1, go to step 7.

6) (RLINK(J) is a thread)
Set W←A(ATOP), ATOP←ATOP-1,
RLINK(J)←W, J←W (RLINK(W) might also be a thread)
if ATOP=0, exit else go to step 5.

7) (RLINK(J) points to a right subtree, which is the next node to be visited) Set RLINK(J)←I+1, go to step 2.

END

Step 2 is deceptive. It "linearizes" the doubly linked list in what, at first glance, seems to be an obvious manner. Upon further examination one will find that step 2 doesn't actually interchange nodes--it garbages up the LINK fields in many cases (for example when LLINK(J)=1). However, observe that step 2 does in fact work, as is illustrated in figures 5-7 through 5-9. One should note that the only permanently garbaged up fields are LLINK and RLINK fields of nodes which are to be encountered next; and these fields are not needed anyway. Also note that the algorithm requires the tree to be "dense" in that it occupies a contiguous physical area. This will always be the case unless some deletions have been made, which causes nodes to be returned to free storage. Thus, the algorithms for finding space and freeing space, GETNODE and FREENODE, use a doubly linked list as explained in Appendix A.2.

After exiting from the second pass of algorithm 5.3.7, the structure of figure 5-6 will have been converted to the structure of figure

Physical Location of Node	LTAG,RTAG,PTR,SKIP	TEXT (indicated by PTR)
1	0,1,101,0	THE QUICK - - -
2	0,0,105,3	QUICK - - -
3	0,0,117,1	FOX - - -
4	1,0,111,2	BROWN - - -
9	1,1,142,1	DOG
5	0,1,128,1	OVER - - -
6	0,1,137,1	LAZY - - -
8	1,1,121,1	JUMPED - - -
7	1,1,133,34	THE LAZY - - -

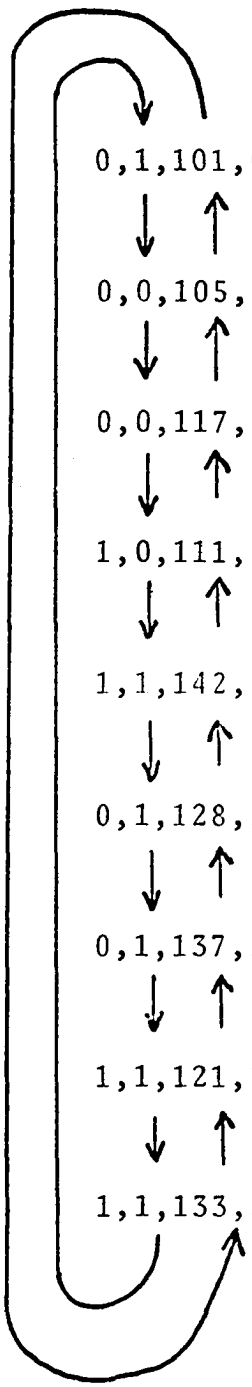


Figure 5-6. Pass 1 creates this from the structure of figure 5-1. The LLINK and RLINK fields are used to form the doubly linked list.

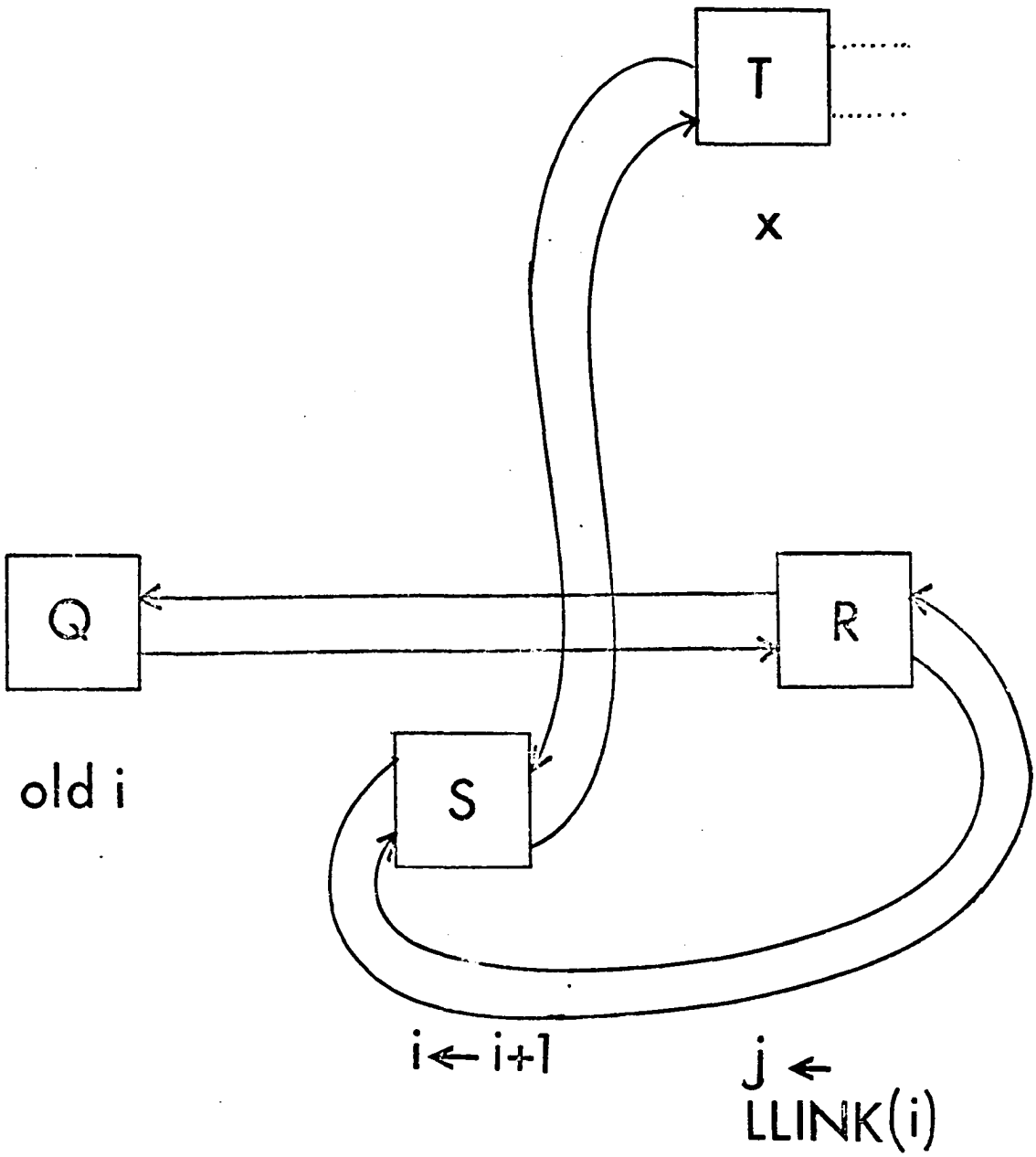


Figure 5-7. The special situation where LLINK(j) = i. The contents of the nodes are indicated by the letters Q, R, S, and T. Their physical locations are given by the letters i, j, and x. The LLINK fields emanate from the lower right side of the nodes.

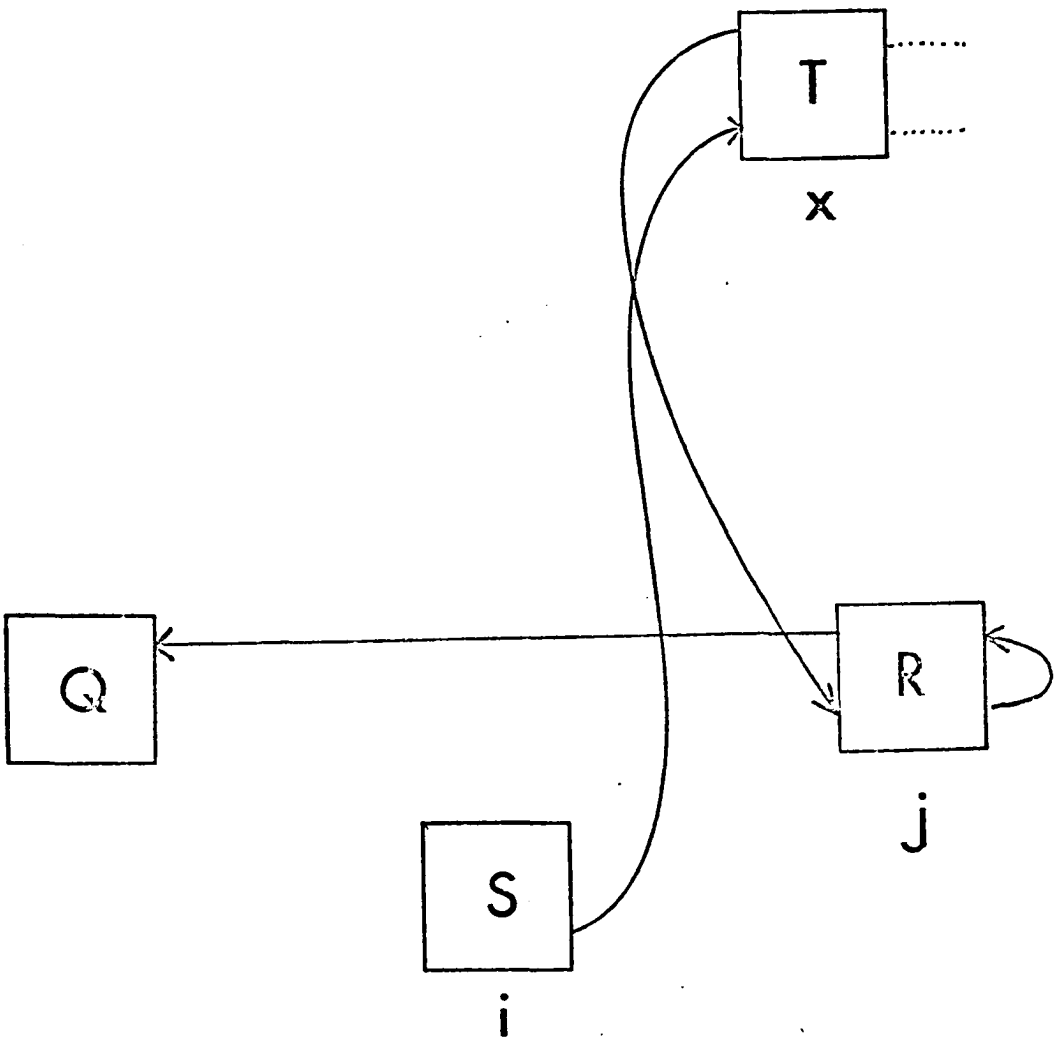


Figure 5-8. The important link fields during step 2 of algorithm 5.3.7, pass 2, just before "Swap all fields of nodes i and j." Note that node j seems to be hopelessly disoriented.

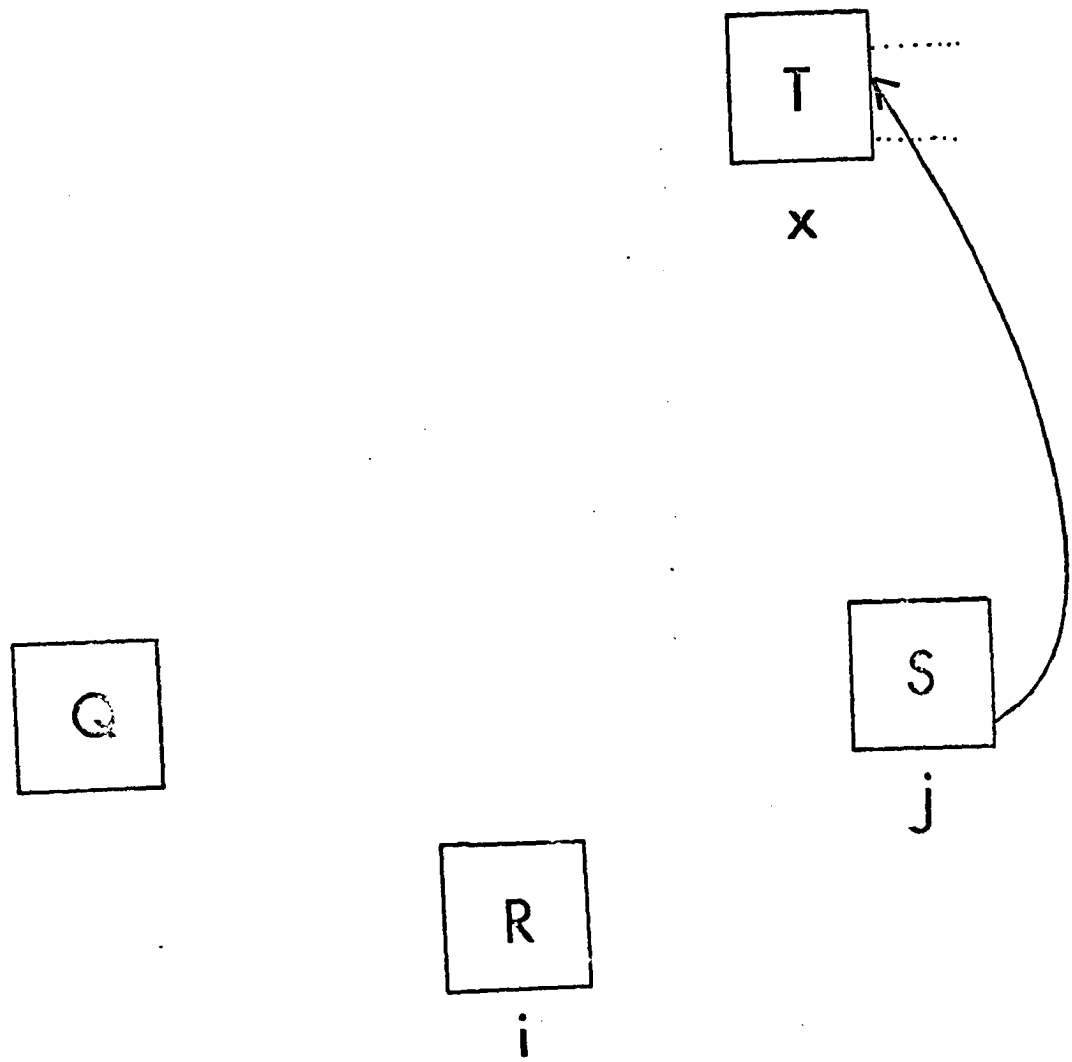


Figure 5-9. After step 2 of algorithm 5.3.7 has been completed, node i contains the proper information and LLINK(j) correctly points to node x. All other link fields are irrelevant.

5-5. The RTAG and LLINK fields are no longer needed and may be used for other purposes.

5.4 A slightly Different Version of the Preorder Sequential Representation

We may find it more convenient to represent LTAG and RLINK simply as a single signed LINK field, particularly if we were using signed RLINK-LLINK fields to start with. In this case, we may easily rewrite the search and traversal algorithms; however algorithm 5.3.7 is a little more difficult to alter. The converted algorithms are given below.

5.4.1 Algorithm: Search Through Structure with Combined RLINK-LTAG

This is a converted form of algorithm 5.3.3.

Input: K, number of bits in K.

Output: P (a pointer to the root of a subtree containing all matches to K).

1) Set P ← 1, J ← 0, N ← number of bits in K.

2) Set Q ← P. If LINK(P)< 0, go to step 6, else set P ← P+1.

3) Set J ← J+SKIP(P) if J> N, go to step 6.

4) If the J+1st bit of K is 0, go to step 2.

5) Set Q ← P, P ←|LINK(P)| . If P >^ go to step 3.

6) etc. (same as algorithm 2.1)

END

5.4.2 Algorithm: Postorder Traversal of Structure with Combined RLINK-LTAG

This is a converted form of algorithm 5.3.1.

Input: X

Output: Visit all nodes in the subtree whose root is at X.

1) Set Y ← X.

2) If LINK (Y) > 0 set Y ← Y + 1 repeat step 2.

3) Visit node Y.

4) Set Z ← LINK (Y) if Z < X exit.

5) If Z < Y set Y ← Z go to step 3, else set Y ← Z go to step 2.

END

5.4.3 Algorithm: Transform Right Threaded Structure to Preorder Sequential Form with Combined RLINK - LTAG

This is a converted form of algorithm 5.3.7. Note that we must be careful not to destroy the sign bit of the RLINK - LLINK fields while the doubly linked structure is being linearized (step 7). Also, Pass One and Pass Two have been combined into a single rather long algorithm. The RLINK field is the field which eventually becomes the LINK field, and the LLINK field is freed. Initially, assume (as before) that a negative LLINK or RLINK field indicates a thread.

Input: A Right Threaded PATRICIA tree whose root is at location 1. (See algorithm 5.3.7)

Output: A Preorder Sequential PATRICIA structure starting at location 1.

1) (Pass one) set ATOP ← 0, B ←0, X ← 1.

2) If RLINK(X)< 0 go to step 3, else set
ATOP ← ATOP + 1, A(ATOP) ← |RLINK(X)|.

3) Set RLINK(X) ← B*SIGN(RLINK(X)), B ← X.

4) If |LLINK(X)| ≠ X, set X ← LLINK(X), go to step 2.

5) If ATOP = 0, then set LLINK(X)← -1, RLINK(1) ← -X,
go to step 6, else set Z ← A(ATOP), ATOP ← ATOP - 1,
LLINK(X) ← -Z, X ← Z, go to step 2.

6) (Pass two) Set I ← 1, ATOP ← 1, A(ATOP) ← 0, go to step 8.

7) Set J ← |LLINK(I)|, I ← I + 1.
If I>J go to step 8, else set RLINK (|LLINK(I)|) ←
J*SIGN(RLINK(|LLINK(I)|)), set LLINK (|RLINK(I)|) ←
J*SIGN(LLINK(|RLINK(I)|)). Swap all fields of nodes I and J.

8) If LLINK(I) $\geq$0 set ATOP $\leftarrow$ ATOP + 1, A(ATOP) $\leftarrow$ I, go to step 7.

9) Set J $\leftarrow$ I.

10) If RLINK(J) $\geq$0 go to step 12.

11) Set K $\leftarrow$ A(ATOP), ATOP $\leftarrow$ ATOP - 1, RLINK(J) $\leftarrow$ K*SIGN(LLINK(J)), J $\leftarrow$ K. If ATOP = 0 exit, else go to step 10.

12) Set RLINK(J) $\leftarrow$ (I+1)*SIGN(LLINK(J)), go to step 7.

END

The above algorithm will transform the tree of figure 5-1 into the structure of figure 5-10 instead of the structure of figure 5-5.

5.5 Further Compression

At the expense of an additional disk access when we are looking at a specific position in the text, we may eliminate the PTR field from the node and place a table of PTR fields on disk along with the text. If the table gives the PTR fields in the order corresponding to the location of the nodes, it is then a simple matter to use the address of a node to access the proper PTR field; then, the PTR field is utilized in the usual manner to pick up the proper text from the disk. Moreover, note that all the nodes of a subtree follow immediately below the root. For example, the node in location 3 (Text = FOX...) is the root of the subtree containing the nodes in locations 4, 5, and 6.

Thus, if we have a match with several keys, and have a buffer area large enough to hold, say, N PTR fields from disk, then we may access the disk once to pick up the PTR fields, and as long as the subtree contains no more than N nodes, we only need to access the disk whenever we pick up the actual text. Hence, if we have N matches for a key, we need only access the disk N+1 times. (The usual method would have required N accesses).

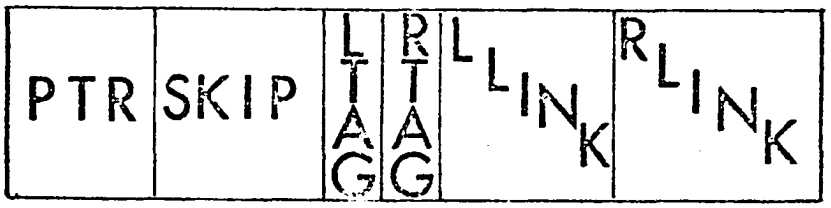
Physical Node Location	SKIP	PTR	LINK	TEXT
1	0	101	0	THE QUICK - - -
2	3	105	9	QUICK BROWN - - -
3	1	117	6	FOX JUMPED - - -
4	2	111	-5	BROWN FOX - - -
5	1	142	-3	DOG
6	1	128	2	OVER THE - - -
7	1	137	6	LAZY DOG
8	1	121	-7	JUMPED OVER - - -
9	34	133	-1	THE LAZY - - -

Figure 5-10. Preorder Sequential form with LTAG and RLINK combined.

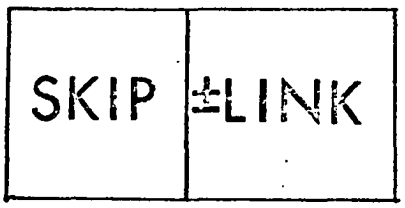
5.6 Conclusions - Advantages of the Compressed Form

The main points of this chapter are summarized below.

1) We have reduced the PATRICIA node in size from the following structure:



to this one:



2) We have speeded up the search process by eliminating one of the subscripts.

3) We have not significantly slowed down the text accessing process, particularly when there exist multiple matches to the same key.

6.0 Practical Applications

Some real and potential applications are now given which employ the algorithms presented in the previous chapters. Note that the table of contents for this dissertation, as well as the list of illustrations, list of algorithms, and index, were all prepared using PATRICIA: specifically algorithms 2.1, 2.2, 2.3, 3.1, 3.2.1, and 3.3.2. First, a deck of cards was punched where each card contained either a chapter or subchapter heading, or a figure caption, along with the page number containing the particular heading or figure. Selected words were flagged as keys; for example every chapter or subchapter number, every occurrence of the word "ALGORITHM" in algorithm subheadings, every occurrence of words such as "TREE", "NODE", "LTAG", etc. To create the list of illustrations, the key "FIGURE" was searched for. The table of contents was listed by searching for the keys "1.", "2.", etc. Then the subtrees containing the keys "FIGURE", "1.", "2.", etc. were deleted from the structure, and a search was made for the null key, which caused all remaining keys to be listed. This forms the index, which does not duplicate chapter and subchapter headings, or the lists of figures and algorithms.

6.1 A Hypothetical Medium - Scale System

Let us assume that we have a file of 20,000 documents, where each document represents an abstract, a dossier, a student record, or some similar thing. Let us further assume that each document contains up to

500 words of about 10 characters each, and contains an average of 5 keys. Thus, our structure would consist of a text of 100,000,000 characters, (which will be stored on bulk storage devices such as IBM 2314 disks), and a PATRICIA tree of 100,000 nodes. The individual node in the PATRICIA tree would have the following bit requirements:

PTR	field -	27 bits	(PTR < 100,000,000)
LLINK	field -	17 bits	(LLINK < 100,000)
RLINK	field -	17 bits	(RLINK < 100,000)
LTAG, RTAG	-	2 bits	
SKIP	field -	12 bits	(assume a string of no more than 500 identical characters in any two keys, or 4000 bits)

This amounts to 75 bits, or 10 bytes (rounded to the next byte), or one million bytes for the PATRICIA tree. The node for the Preorder Sequential form of section 5.5 would have the bit requirements given below:

LINK	field -	18 bits	(|LINK| < 100,000)
SKIP	field -	12 bits	

which is 30 bits, or 4 bytes, or 400K bytes for the entire Preorder Sequential structure. In addition, the PTR field would require 27 bits, or 4 bytes, or 400K bytes of disk storage for the entire PTR table.

Thus, PATRICIA would use (400 + 300 + 1000)K bytes of disk storage which amounts to 1.7% of the entire file. Building the PATRICIA structures would require an amount of time not much greater than $2 \cdot 100{,}000 \cdot X$, where X represents the average access time to the disk (see Appendix C). Two disk accesses are required for each key inserted into the tree. If we assume that X = 75 milliseconds (the average access time for a 2314 disk), we note that none of the algorithms involved in the building

process (2.1, 2.2, 5.1.1, 5.4.3) require an amount of time per key that is anywhere near the time used by disk accesses. Thus, the PATRICIA structures would be built in about 72·2·100000 milliseconds = 15,000 seconds or about 4 1/2 hours.

The user would run in either "update" mode or "query" mode. In update mode, alterations are made to the PATRICIA tree or to the actual text. After all alterations have been completed, the new version of the Preorder Sequential form is built; this structure, along with the tree, is then written out on the disk. In query mode the user works with the Preorder Sequential structure, which could run in a multi-programming environment since it requires so much less space than the tree.

6.1.1 Boolean Operations

For most applications the user wants the capability of asking for specific combinations of keys; for example, he might want to find all keys that start with "ANT" except "ANTLER" and "ANTECEDENT." Or he might want to retrieve all documents that contain both of the keys "CHEMISTRY" and "CRYSTAL."

The construction of a particular Boolean query editor is easily accomplished for PATRICIA, since we can quickly find all the subtrees for the keys contained within a Boolean expression. For example, if we wish to eliminate "ANTLER" and "ANTECEDENT" from our query for "ANT", we simply traverse the subtree of keys that start with "ANT", but we do not visit either of the subtrees of keys starting with "ANTLER" or "ANTECEDENT", which are both contained in the larger subtree for "ANT." To find only those documents containing both of the keys "CHEMISTRY" and "CRYSTAL" we look at the PTR fields of nodes in the two subtrees, and

only keep entries which point to documents that are present in both.

6.2 Another Application - CALL FOR ACTION

The text program of Appendix B was used to create a key word index to referral files for the Oklahoma branch of CALL FOR ACTION, which is based in Oklahoma City at television station KWTV, channel 9. The data was gathered by a graduate student in Library Science at the University of Oklahoma, who prepared the cards in a rather unrestrictive format that was most comfortable for her. She tagged key words with asterisks, and indicated the end of a particular "abstract" by punching a "1"; then she continued immediately with the next record (on the same card if she wished). At the time of this application, the program maintained all text in core, which imposed a 32767 character restriction. Nevertheless, this was sufficient to permit the run. Two runs were made, each consisting of about 1000 keys and 25000 characters of text. In the process of preparing the data, a mistake in one of the texts was corrected by punching a text alteration card, which was entered after the entire structure had been built.

To list all the keys, a search was made for the null key. The entire run took about seven minutes on a 360/50, with 15 copies of the printout being produced. Some of the output is illustrated in figures 6-1 and 6-2.

6.3 Some Useful Dirty Tricks Involving Comparison Strings

We can guarantee that the number of identical characters in a comparison between two keys is held to a reasonable limit. The usual way to do this is to give each document a unique terminating symbol in the form of some catalogue code number. This insures that the maximum

THE COMPLETE TEXT IS LISTED BELOW

```
                                                
```

```
WEEDS AND TRASH - OKLAHOMA CITY COUNCIL                                  23261
                                  ENVIRONMENT -   WEEDS AND TRASH - OKLAHOMA
                                CITY COUNCIL
.. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
WEEDS AND TRASH - OKLAHOMA CITY-COUNTY HEALTH DEPARTMEN                  23317
                                  ENVIRONMENT -   WEEDS AND TRASH - OKLAHOMA
                                CITY-COUNTY HEALTH DEPARTMENT
.. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
WELFARE - DEPARTMENT OF INSTITUTIONS, SOCIAL AND REHABI                   7969
                                  CHILDREN -   WELFARE - DEPARTMENT OF INSTITUTIONS,
                                SOCIAL AND REHABILITATIVE SERVICES (DISRS)

.. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
WESLEYAN YOUTH INC., OKC                                                 8382
                                  CHILDREN AND JUVENILES - GUIDANCE AND COUNSELING
                                - BOYS -   WESLEYAN YOUTH INC., OKC
.. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
WILDLIFE CONSERVATION                                                    2506
                                  ANIMALS - WILDLIFE CONSERVATION - OKLAHOMA
                                STATE DEPARTMENT OF   WILDLIFE CONSERVATION

.. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
WILDLIFE CONSERVATION                                                   20500
                                  ENVIRONMENT - AGENCIES AND ASSOCIATIONS
                                - OKLAHOMA STATE DEPARTMENT OF   WILDLIFE
                                CONSERVATION
.. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
WILDLIFE CONSERVATION - OKLAHOMA CITY ZOO                                2399
                                  ANIMALS -   WILDLIFE CONSERVATION - OKLAHOMA
                                CITY ZOO
.. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
WILDLIFE CONSERVATION - OKLAHOMA STATE DEPARTMENT OF WI                  2453
                                  ANIMALS -   WILDLIFE CONSERVATION - OKLAHOMA
                                STATE DEPARTMENT OF WILDLIFE CONSERVATION

.. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
WONDER HOUSE DAY CARE CENTER - NORMAN                                    5475
                                  CHILDREN - DAY CARE SERVICES -   WONDER
                                HOUSE DAY CARE CENTER - NORMAN
.. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
```

Figure 6-2. Some of the output produced for Call for Action. The keys appear at the left. Every key is printed, along with the entire abstract for that key. The 4 or 5 digit number at the right gives the PTR field for the particular key; thus we can refer back to the text (figure 6-1) for updating purposes.

length of identical keys is limited to the size of the document. Of course we would have little use for identical documents, so in actuality the identical character strings would be much smaller. Thus, in the example of section 6.1, we assumed no more than 500 identical comparison characters in each document of (no more than) 5000 characters.

If this limit is exceeded, we may under certain circumstances employ a different method to lower the number of identical characters in two or more keys. The method requires that we insert a non-printing character somewhere in one of the identical strings. This character should be placed so that it conforms to the following criteria:

1) It is placed in an inoffensive spot, such as between two words.
2) It is far enough ahead of the start of any key so that it will not interfere with a query. (30 or 40 characters ahead should be sufficient.) The choice of where to put such a symbol should be made during the building phase.

This special character could also be used to alter the overall structure of the PATRICIA tree in an attempt to make it more balanced, although most practical applications involve text that either produces reasonably well-balanced structures, or does not have the sufficient 30 or 40 characters between the start of any key and the spot at which we might want to insert our special symbol. Nevertheless, it is intriguing to recall the example of figures 4-1 through 4-3, where a single well placed character drastically restructured the tree. Unfortunately, it also caused the keys ahead of the alteration to be changed.

6.3.1 Using a Terminating Symbol to Limit the Scope of Text Alterations.

The special terminating symbol mentioned in section 6.3 also

serves to partition the text into small units, or "books" (Morrison 1968). This is very practical, for it means that a given text alteration will affect only the particular book where the alteration is being made. If we allocate on disk extra vacant space for each book, then we can change the affected area without having to push other unaffected books around. Moreover, if we can hold an entire book in a memory buffer, then only one access to the disk is required for any alteration (or group of alterations) made to a book.

A.0 APPENDIX - OTHER ALGORITHMS

This appendix contains algorithms which, though not specifically germaine to PATRICIA, are useful to (or are required by) some of the algorithms presented in the preceding chapters.

A.1 Algorithm: Get a node from an available list

This algorithm gets a PATRICIA node from an available list of nodes (Knuth 1968, p. 254). Initially the list is sequential, consisting of M available nodes. Also, initially let MARKER = 0, TOPFREE = λ. The algorithm will exit with pointer X pointing to the next available node. (The nodes are linked together by their LLINK fields when they are returned by A.1.1 below.)

Input: X (A pointer as yet undefined).

Output: X points to an available node.

1) If TOPFREE $\neq \lambda$, set X $\leftarrow$ TOPFREE, TOPFREE $\leftarrow$ LLINK(TOPFREE), exit.

2) If MARKER = M, no space is available, else set MARKER $\leftarrow$ MARKER + 1, X $\leftarrow$ MARKER, exit.

END

A.1.1 Algorithm: Return a node to an available list.

This algorithm returns a PATRICIA node to an available list. The nodes will be linked by their LLINK fields. Assume the node being released is pointed to by X.

Input: X (a pointer).

Output: Node X has been freed.

1) Set LLINK(X) ← TOPFREE, set TOPFREE ← X, exit

END

A.2 Algorithms for a doubly linked available list.

The algorithms of section A.1 have been rewritten so that they utilize a doubly linked list. A list head is set up at location M, the high order memory location. The list head is chosen to be there so that it won't conflict with the PATRICIA tree. The list is unlinked to start with, and is only linked up as nodes are returned to free storage. This guarantees a "dense" structure in that all nodes are members of doubly linked lists. In other words, if X equals the location of the highest physical node in the PATRICIA tree, then for $1 \leq I \leq X$, node (I) is a member of either the doubly linked available list or of the tree (which, after the application of algorithm 5.3.7, pass 2, is in the form of a doubly linked list.) If this were not the case, then algorithm 5.4.3 could fail in step 7 whenever node (I) contained irrelevant or random RLINK or LLINK fields.

The algorithms are presented below. Assume that MARKER initially is set to 2 (because location 1 is reserved for the root of the PATRICIA tree). Again, let M be the highest available memory location as well as the list head. Thus, initially, we set RLINK(M)=LLINK(M)=M.

A.2.1 Algorithm: Get a Node from a Doubly Linked Available List.

Input: X (a pointer as yet undefined.

Output: X points to an available node.

1) If LLINK(M)=M, go to step 4.

2) Set X←RLINK(X).

3) Set LLINK(RLINK(X))←M
RLINK(LLINK(X))←RLINK(X), exit.

4) If MARKER = M-1, overflow, else
Set MARKER MARKER+1, X MARKER, exit.

END

A.2.2 Algorithm: Return a Node to a Doubly Linked Available list.

Input: X (a pointer).

Output: Node X has been returned to free storage.

1) Set LLINK(X)←M, RLINK(X)←RLINK(M)

2) Set LLINK(RLINK(M))←X,
RLINK(M)←X, exit.

END

A.3 Algorithm: Print a Tree.

This algorithm first finds the level of a node. The tree will be printed on its side, with the root at the left. Let N horizontal spaces exist between each level on the printout. The level, L, of a node will be given by the number of entries in the sequential stack used for the traversal. Only one node will be printed on a given line. We shall have P vertical spaces between nodes.

The rules for drawing the tree are given below. At each "visit" to a node, space out P-1 lines, printing "*" and "ß". Then (where required as defined later):

1) Write out the INFO field starting at line position L*N+1 (assume INFO is the information we wish to see, and occupies at least N positions).

2) If the node is a right subtree, start printing a vertical line ("*") at position L * N, otherwise, start printing spaces at position L * N. (Assume the root is not a right subtree)

3) If the node has a left subtree, start printing a vertical line at position (L + 1) * N, otherwise start printing spaces. (INFO fields take precedence over any lines; i.e., don't print an "*" if it obliterates information.)

These rules are implemented as follows:

Assume ATOP points to the top of the stack, A, which is used by the algorithm below; then, at a visit to node (X):

1) ATOP gives the level of node X.

2) If ATOP $\neq$ 0 (If we are not looking at the root) and if, for node (X), RLINK(A(ATOP)) = X (The node at the next numerically lower level points to node (X)), then node (X) is a right subtree; print "*" at line position L*N.

3) If LLINK(X) $\neq \lambda$, then node (X) has a left subtree. Print "*" at (L + 1)*N.

The entire algorithm is given below:

Input: TOP, the pointer to the root.

Output: The printed representation of the tree.

1) Set X $\leftarrow$ TOP
   ATOP $\leftarrow$ 0.

2) If X = λ, go to step 4.

3) Set ATOP $\leftarrow$ ATOP + 1,
   A(ATOP) $\leftarrow$ X
   X $\leftarrow$ RLINK(X), got to step 2.

4) If ATOP = 0, exit, we are done
   else set X $\leftarrow$ A(ATOP), ATOP $\leftarrow$ ATOP-1

5) If X < 0 go to step 4

6) "visit" node X

7) Set ATOP ← ATOP+1, A(ATOP) ← X,
X ← LLINK(X), go to step 2.

Step 6 is now expanded:

(Initially, set skeleton line, S, = all "Ø")

6-1) Write out S(1) through S(N*ATOP), then all node information, starting at N*ATOP+1.

6-2) If ATOP ≠ 0 and RLINK(A(ATOP)) = X, move "Ø" to S(L*N) (node X is a right subtree) else move "Ø" to S(L*N).

6-3) If LLINK(X) ≠ λ, move "*" to S((L+1)*N) (node X has a left subtree) else move "Ø" to S((L+1)*N).

6-4 Write out S a total of P-1 times.

END

B.0 APPENDIX - THE TEST PROGRAM

The algorithms described in chapters 2-5 have been tested by writing a PL/I program in which there is a close correspondence between PL/I code and the algorithms wherever possible. Details of the program's operation are given in section B.1, and the source listing, as well as some sample input and output, is shown in section B.2.

B.1 The Facilities of the Program

The test program has the capability of performing the functions listed below.

1) Read text and create a PATRICIA tree (either Right Threaded or unthreaded).
2) Display a PATRICIA tree and/or the text.
3) Search for a key and list all matches.
4) Delete a node from the tree, (or delete an entire subtree).
5) Delete or insert text.
6) Convert from a Right Threaded tree structure to Preorder sequential representation.

B.1.1 Read Text and Create a Tree.

The first data to follow the //GO·SYSIN card is of the self explanatory form given below.

*HERE *IS *SOME SAMPLE *TEXT. *THE ASTERISKS INDICATE *THE *STARTING POINTS FOR *KEYS AND ARE NOT STORED. AT THE CONCLUSION OF THE TEXT,

*PUNCH *AN AMPERSAND IF YOU WANT *THE *STRUCTURE TO BE UNTHREADED, OR A DOLLAR SIGN IF YOU *WANT IT TO BE *THREADED. *THIS *TREE WILL BE RIGHT *THREADED. THE TEXT CONTINUES FROM COLUMN 80 OF ONE CARD TO COLUMN ONE OF THE NEXT.$

After reading the text and creating the tree, the program will allow one to perform any or all of the functions B.1.2 through B.1.6. For each of these functions, leading blanks are ignored.

B.1.2 Display the Tree and/or the Text.

To display both the tree and the text, type a "!". To display only the text. type a "|". The tree is drawn on its side with the root at the left. LLINK fields are indicated by "L"s connecting nodes and going down the page; RLINK fields are indicated by "R"s similarly going up the page.

The rest of the node structure is illustrated below, and is in one of four possible forms.

1) If the node has no backward pointers:

```
*JJJ*SSS*PPPP
```

2) If the node has a right backward pointer:

```
**QQQ***TEXT-----
*JJJ*SSS*PPPP
```

3) If the node has a left backward pointer:

```
*JJJ*SSS*PPPP
**QQQ***TEXT-------
```

4) If the node has left and right backward pointers:

```
**QQQ***TEXT----------
*JJJ*SSS*PPPP
**QQQ***TEXT----------
```

where:

JJJ = physical location of the node

SSS = SKIP field

PPPP = PTR field

QQQ = value of LLINK or RLINK when one (or both) is a backward pointer

TEXT = the first 20 characters of the text pointed to by PPPP(QQQ).

B.1.3 Search for a Key and List All its Matches.

To do this, punch the key followed by a "?" Examples:

THE? - searches for all keys starting with THE.

THEb? - searches for all keys starting with THEb.

S?ISb?bbbbT? - searches for all keys starting with S, ISb, and T.

SOMEbSAMPLEb? - searches for all keys starting with SOMEbSAMPLEb.

B.1.4 Delete a node from the tree.

Punch the key, followed by a "/". Examples:

HEREbISbSOMEb/	Deletes any key starting with "HEREbISbSOME"
HEREb/ISb/SOMEb/	Deletes any keys starting with "HEREb" or "ISb" or "SOMEb".
S/ T/	Deletes all keys starting with S or T.

Only the PATRICIA tree is altered. The text is not affected in any way.

B.1.5 Delete or insert text.

The different ways of accomplishing this are given below:

(XXX-YYY)	Delete all text between (and including) positions XXX and YYY. XXX and YYY are integers.

LLL - - -)	Delete an occurrence of the characters "LLL - - -" from the text. Note that "LLL - - -" must be a key; moreover, only one occurrence of "LLL - - -" as a key will be found. To delete other occurrences, repeat.
(XXX*LLL - - -)	Insert the letters "LLL - - - " into the text starting in position XXX + 1.

Some examples are given below. They all refer to the text given in B.1.1.

(2-9)	This would cause the text "ERE¢IS¢S" to be deleted, along with the keys starting at IS and SOME. The first position of the text will still start at "H", but the second will be "O", and the text will read "HOME SAMPLE---." Also, "HOME---" will be the starting point for a key.
(4*SY)	This would insert the letters "SY after the second "E" of "HERE", forming "HERESY". Also, there will be a key created starting at "SY¢IS---". The former key "HERE IS ---- will still exist, but it will now read "HERESY IS ----".
THREA)	This will cause "THREA" to be deleted from the text at some occurence of a key starting with "THREA". Either of the two keys starting with "THREADED.---" could be affected. The key itself will be deleted, and the text will be altered to read "---¢BE¢DED---" or "---RIGHT¢DED---" depending upon which key is deleted.

B.1.6 Convert to Preorder sequential Form.

After the user has specified all the functions that he may wish to perform for a particular text, he punches a "$" (dollar sign). This will cause the tree to be converted to Preorder Sequential form if it was created as a Right Threaded tree. The nodes will then be traversed in postorder, and listed. The program will then accept a new text (i.e. return to B.1).

B.2 Sample Output.

The next few pages contain a listing of the test program, along with a sample run. Note that in the program listing, the algorithms are indicated by enclosing them in appropriately numbered "/* - - - - - */" cards.

```
1                 PATRICA:        PROC  OPTIONS (MAIN);
                  /* THIS PROGRAM IS MEANT TO BE A TEST PROGRAM ONLY.
                  THEREFORE MUCH OF THE CODE IS STILL IN RATHER ROUGH FORM. */
2                (STRINGRANGE):   BEGIN;
3                  (SUBSCRIPTRANGE):   BEGIN;
4                   NKEYS = 200:   NCHARS  = 2000:
6                      NKEYS = 550:  NCHARS =        20000:
8                        BEGIN;
9                       DCL  (RLINK(NKEYS),LLINK(NKEYS),SKIPS(NKEYS)) FIXED BIN (15,0);
10                           DCL PTR(NKEYS) FIXED BIN (31,0),KEY_LIST(NKEYS  )
                        FIXED BIN   (15,0);
11                     DCL C2 CHAR(NCHARS+250);
12                       DCL CHARS CHAR(NCHARS)DEF C2 POS (250);
13                      SUBSTR(C2,1) = ' ';
14                        DCL  LINK(NKEYS)  FIXED BIN(15,0)  DEF RLINK;
15                           DCL COMPARE CHAR(1000), BITS BIT(8000) DEF COMPARE;
16                           DCL KEY CHAR(1000), KEY_BITS BIT(8000) DEF KEY;
17                          DCL TEMP CHAR(1);
18                           DCL WHAT_BITS(200)  CHAR(1);
19                      DCL (THREAD_SEARCHES,TOT_SEARCHES) FIXED BIN (31,0);
20                           DCL SKELTON CHAR(200);
21                           DCL (P,Q,J,N,R,T,L,ATOP,X,PP,FLAG,MATCH,POSITION,KK,
                    TOT_KEYS,QQ,TD,FT,TT,EMPTY,AA,D,S,W,Y,V,Z,I) FIXED BIN (31,0);
22                     DCL (TOPFREE,MARKER,M,STARTING_TEXT_POSITION,VERT_SPACE,
                          HORIZ_SPACE, A(100),PRINTPAGE_WIDTH)   FIXED BIN  (31,0);
23                           DCL   II  FIXED BIN (31,0);
24                          DCL PRINT_TREE ENTRY (FIXED BIN (31,0));
                       /*  ************************************************    */
25                    STARTING_TEXT_POSITION  = 1;
26                           PRINTPAGE_WIDTH = 120:
27                            PRINTPAGE_WIDTH =         132:
28                       M=NKEYS:
29                           OPEN FILE(SYSPRINT)PAGESIZE(52)LINESIZE(PRINTPAGE_WIDTH):
30                          ON ENDFILE(SYSIN) GO TO NODATA;
32           READIN:
33                           LLINK(M) = M; RLINK(M) = M; MARKER = 0;
35                           THREAD_SEARCHES = 0; TOT_SEARCHES = 0;
37                           TOT_KEYS = 0; I= STARTING_TEXT_POSITION - 1; KK=1;
40                            SUBSTR(CHARS,1,141) = ' ';
41                           DO WHILE (I<999999);
42                        GET EDIT (TEMP) (A(1));
43                            I=I+1; IF TEMP = '*'
45                               THEN DO;
46                                    TOT_KEYS = TOT_KEYS + 1;
47                             KEY_LIST(TOT_KEYS) =.I;
48                             GET EDIT (TEMP)(A(1));
49                                    END;
```

```
50                      SUBSTR(CHARS,I,1) = TEMP;
51                     IF TEMP = '$' | TEMP = '&' THEN DO;
53                       DO L=I+1 TO I+100  ;
54                           SUBSTR(CHARS,L,5) = '     '; END;
56                       GET SKIP;  GO TO NOREAD;  END;       END;
60              NOREAD:    /*  SET UP ROOT OF ENTIRE STRUCTURE    */
61                       IF TEMP = '$' THEN THREAD = 1; ELSE THREAD = 0;
63                       TOPFREE=0;  MARKER=0;
65                       CALL GETNODE(R); LLINK(R) = -R; PTR(R)=KEY_LIST(1);
68                        RLINK(R) =  -R;
69                       SKIPS(R) = 0;
70                      POSITION = 1;
71                      FLAG = 0;
72                       CALL INSERT_KEY;
73                     PUT PAGE;
74                       PUT EDIT
                         (' STATISTICS FOR BUILDING TREE',
                          '   NUMBER OF EXTRA NODE ACCESSES REQUIRED TO THE END=',
                         THREAD_SEARCHES,'   NUMBER OF SEARCHES=',TOT_SEARCHES,
                         '   NUMBER OF KEYS IN THE TREE=',TOT_KEYS)
                         (SKIP,A,SKIP,A,F(4),SKIP,A,F(4),SKIP,A,F(4));
                  /* THIS ROUTINE READS IN KEYS FOR MATCHING OR DELETING.
                             IT ALSO PROVIDES THE OPTION OF DISPLAYING THE CURRENT
                              PATRICIA STRUCTURE       */
75              READMORE:    POSITION = 0;
76                            TEXT_POSITION_FLAG = 0;
                             /* A DEBUGGING AID    CAUSES THE COMPARISON BITS TO PRINT
                            FLAG=1;
                           ********************************************************* */
77                           DO II = 1 TO 500;
78                           GET EDIT (TEMP)(A(1));
79                            IF POSITION = 0  &  TEMP = ' '  THEN GO TO READMORE;
81                               IF TEMP = '('   THEN DO;
83                        TEXT_POSITION_FLAG = 1;   POSITION = 0;  END;
86                           IF TEMP = '/' THEN DO; PUT SKIP; PUT SKIP LIST
                          ('*****DELETING ALL KEYS STARTING WITH   ');
90                              GO TO SEARCH1;  END;
92                           IF TEMP = '?' THEN DO;
94                     PUT PAGE LIST ('*****SEARCHING FOR ALL KEYS STARTING WITH    ');
95                              GO TO SEARCH1;  END;
97                        IF TEMP = '|'  THEN DO;
99                       PUT PAGE EDIT (' THE COMPLETE TEXT IS LISTED BELOW')
                           (A,SKIP);
100                         CALL PRINT_TEXT;  PUT PAGE;  GO TO READMORE;  END;
104                          IF TEMP = '$' THEN DO;
106                     IF THREAD = 1 THEN DO;
108                  CALL TRANSFORM;
```

```
109                                 X=1;   CALL SEQUENTIAL_TRAVERSE(X);
111                          END;
112                                 GET SKIP; GO TO READIN; END;
115                         IF TEMP =    ' '  THEN DO;
117                           PUT PAGE; IF THREAD = 1 THEN PUT SKIP LIST ('
                              THE STRUCTURE IS RIGHT-THREADED  '):
120                                    CALL PRINT_TEXT;  CALL PRINT_TREE(1R); PUT
123                              PAGE;   GO TO READMORE;        END;
125                           IF TEMP = ')' THEN DO;
127                  TEXT_EDIT:    IF TEXT_POSITION_FLAG = 0  THEN GO TO LOOP;
129                  ALTER_TEXT_FROM_POSITION: BEGIN; DCL(I,J) FIXED BIN(31,0);
131                           I=INDEX(KEY,'*'); IF I ¬= 0 THEN DO;
134                           J=SUBSTR(KEY,2,I-2); CHARS=SUBSTR(CHARS,1,J)||
                              SUBSTR(KEY,I+1,POSITION-I) || SUBSTR(CHARS,J+1);
136                           PUT SKIP;
137                           PUT SKIP EDIT('***** INSERTING AFTER POSITION ',J,
                              ' THE TEXT ***',SUBSTR(KEY,I+1,POSITION-I))(A,F(4),A,A);
138                           START=J+1; END=J+1; SHIFT=I-POSITION;
141                           CALL DELETE_TEXT; TOT_KEYS = 1; KEY_LIST(1) = START;
144                           KK=0;
145                            CALL INSERT_KEY; GO TO READMORE; END;
148                           ELSE DO;
149                          I=INDEX(KEY,'-'); IF I=0 THEN DO; PUT EDIT('BAD QUERY')(A);
153                           GO TO READMORE;  END;      START=SUBSTR(KEY,2,I-2);
156                                END = SUBSTR(KEY,I+1,POSITION-I)+1; SHIFT=END-START;
158                           PUT SKIP EDIT('*****DELETING ALL TEXT BETWEEN ',START,
                              ' AND ',END-1,' NAMELY ***',SUBSTR(CHARS,START,SHIFT))
                              (SKIP,A,F(4),A,F(4),A,A);
159                           CHARS=SUBSTR(CHARS,1,START-1)|| SUBSTR(CHARS,END);
160                             CALL DELETE_TEXT;  GO TO READMORE;   END;
163                                   END      ALTER_TEXT_FROM_POSITION;
164                  LOOP:          PUT SKIP EDIT('*****DELETING THE TEXT    ',
165                        SUBSTR(KEY,1,POSITION))(SKIP,A,A); N=POSITION*8;
166                                CALL SEARCH; IF MATCH = 0 THEN DO;
169                           PUT EDIT ('*** BUT IT ISN'T THERE')(A); GO TO READMORE;
171                           END; ELSE PUT EDIT ('*** FROM POSITION ',PTR(P))
173                              (A,F(5)); START      =PTR(P); END=START+POSITION;
175                           CHARS = SUBSTR(CHARS,1,START-1)||SUBSTR(CHARS,END);
176                          SHIFT = POSITION;  CALL DELETE_TEXT; GO TO READMORE; END;
180                           ELSE DO;
181                          POSITION = POSITION + 1;  SUBSTR(KEY,POSITION,1) = TEMP;
183                                   END;          END;
185                  SEARCH1:
                              PUT EDIT (SUBSTR(KEY,1,POSITION))  (A);
186                                  N=POSITION * 8;   CALL SEARCH;
188                           IF MATCH = 1 THEN GO TO CHECK_MORE;
190                           PUT EDIT('*** BUT NO MATCH WAS FOUND')(A);
```

```
191                     GO TO READMORE;
192             CHECK_MORE:  IF TEMP ¬= '/' THEN DO; CALL TRAVERSE; GO TO READMORE;
196                                                  END;
197                       IF PP<0 THEN DO; CALL DELETE_ENTRY; GO TO READMORE;
201                                    END;
202             DELETE_SUBTREE:
                        /*  ANOTHER DEBUGGING AID
                          PUT EDIT ('*** ITS SUBTREE IS DRAWN BELOW')(A);
                             CALL PRINT_TREE(PP);
                        ***************************************************     */
                  /* # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # */
                  /*  3.3.2  3.3.2  3.3.2  3.3.2  3.3.2  3.3.2  3.3.2  3.3.2  3.3.2  */
203             A_3_3_2__123:       TD=0; FT=W; R=P; PD=0; CALL TRAVERSE;
207             A_3_3_2__4:         IF RLINK(TD) = R THEN RLINK(TD) = -PD; ELSE
209                                    LLINK(TD) = -PD;
210             A_3_3_2__5:   CALL FIND_IT; GO TO READMORE;
                  /*  3.3.2  3.3.2  3.3.2  3.3.2  3.3.2  3.3.2  3.3.2  3.3.2  */
                  /* # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # */
212             TRAVERSE:       PROC;
213                           DCL  (II,JJ)  FIXED BIN(31,0);
214                           DCL CHARS_TO_LEFT_OF_KEY  FIXED BIN(31,0);
215                                DCL   LINE  CHAR(500);
216                           DCL KEY_COLUMN  FIXED BIN(31,0);
217                         DCL CODE CHAR (8);
218                            DCL CODELENGTH  FIXED BIN (31,0);
219                         CODELENGTH  = 0;
                    /*      CODELENGTH MUST BE 0 FOR CALL FOR ACTION FORMAT  */
220                         CODELENGTH   = 8;
221                           DCL    (I,J        )   FIXED BIN (31,0);
222                           CHARS_TO_LEFT_OF_KEY = 20;
223                       CHARS_TO_LEFT_OF_KEY = 34;
224             FOUND_ONE:    ATOP=0          ;X = PP;
226             FOUND_THREE: IF X > 0 THEN GO TO FOUND_SIX;
228                           IF TEMP = '?'  THEN DO;
230                           II=ABS(X); O=PTR(II);
                      /* O GIVES TEXT POSITION  */
232                     LINE=SUBSTR(C2,0,249) || '  ' ||
                              SUBSTR(CHARS,0)       ;
233                           DO I = 250 TO 2 BY -1 WHILE (SUBSTR(LINE,I,1) ¬='|');
234                            END;
235                             SUBSTR(LINE,1,I+CODELENGTH) = ' ';
236                         J=INDEX(SUBSTR(LINE,253),'|');
237                           SUBSTR(CODE,1) = ' ';
238                           IF J ¬= 0 THEN DO;
240                       SUBSTR(CODE,1) = SUBSTR(LINE,253+J,CODELENGTH);
241                           SUBSTR(LINE,252+J      )= ' '; END;
                        /*   SPECIAL FORMAT FOR  CALL FOR ACTION  */
```

```
                /*   ************************************************************
                       PUT SKIP EDIT (SUBSTR(LINE,252,56),0)
                         (A,X(50),F(7));
                       II=I+CODELENGTH   DO WHILE (II<= 253+J);
                         JJ=INDEX(SUBSTR(LINE,II+40),' ');
                          PUT EDIT (SUBSTR(LINE,II,40+JJ))
                      (SKIP,X(52),A);
                          II=II+40+JJ;      END;
                      PUT SKIP EDIT (('.. .. .. ' DO II=1 TO 3))((100)A);
                     /*   END OF SPECIAL STUFF      */
                   /*  SPECIAL FORMAT FOR DISSERTATION INDEX, ETC.    */
                  /* ***************************************************
                       PUT SKIP EDIT(SUBSTR(LINE,250-CHARS_TO_LEFT_OF_KEY,
                         PRINTPAGE_WIDTH-10),CODE)
                         (A,X(1),A);
                   /*    END OF DISSERTATION FORMAT   */
                  /*   SPECIAL FORMAT FOR DISSERTATION EXAMPLE IN APPENDIX B    */
243                 PUT SKIP EDIT (SUBSTR(LINE,250-30,PRINTPAGE_WIDTH-10),0)
                         (A,X(1),F(6));
                  /*   ***************************************
                  /*   END OF SPECIAL STUFF     */
244                        END;
245                   IF TEMP = '/' THEN DO; CALL CHECK_FOR_ANCESTOR;
248                   IF AA=0 THEN DO; PD=ABS(X); GO TO FOUND_EXIT; END; END;
254                    IF ATOP = 0 THEN GO TO FOUND_EXIT;ELSE GO TO FOUND_NINE;
257           FOUND_SIX:     ATOP = ATOP + 1; A(ATOP)=X; X = LLINK(X)
260                            ;GO TO FOUND_THREE;
261           FOUND_NINE:  X = A(ATOP); ATOP = ATOP - 1; X = RLINK(X);
264                           GO TO FOUND_THREE;
265            CHECK_FOR_ANCESTOR:  PROC;
266            CH_2:          AA=ABS(X);  CALL SEARCH;
268                                       END  CHECK_FOR_ANCESTOR;
269             FOUND_EXIT:          END  TRAVERSE;
270           NODATA:       PUT PAGE EDIT ('RAN OUT OF DATA!')(A);
271                         STOP;
272           DELETE_ENTRY:  PROCEDURE          ;
                /* # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # */
                /*  3.2.1  3.2.1  3.2.1  3.2.1  3.2.1  3.2.1  3.2.1  3.2.1  */
273            FIND_NODES_2:  PD=ABS(P); TD=ABS(Q); FT=W; /* W WAS FOUND
                                BY THE SEARCH ALGORITHM  */
276            FIND_IT:       ENTRY;
277            FIND_NODE_3:
278                         IF LLINK(TD)*RLINK(TD)>0|TD=PD THEN DO;
279                         TT=TD;  GO TO F_N_4;  END;
282            FIND_NODES_4:  KEY = SUBSTR(CHARS,BTR(TD),1000) ; N=8000;
284                              CALL SEARCH;  TT=0;
286            F_N_4:
```

```
                  /*  3.2.1  3.2.1  3.2.1  3.2.1  3.2.1  3.2.1  3.2.1  3.2.1  */
                  /* # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # */
                        /*  ANOTHER DEBUGGING AID
                             PUT SKIP EDIT ('DELETING THE KEY',
                             SUBSTR(CHARS,PTR(ABS(PD)),30),'-----  PD,TD,TT,FT ARE',
                                 PD,TD,TT,FT)(A,A,A,(4)F(5))        ;
                   */
                  /*  ************************************************************ */
                  /* # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # */
                  /*  3.1  3.1  3.1  3.1  3.1  3.1  3.1  3.1  3.1  3.1  3.1   */
                DELETE_1:    PTR(PD)=PTR(TD);
287             DELETE_2:    IF ABS(LLINK(TT))=TD THEN LLINK(TT)=-PD; ELSE
289                              RLINK(TT)=-PD;
290             DELETE_3:    IF ABS(LLINK(TD))=PD THEN W=RLINK(TD);
292                               ELSE W=LLINK(TD);
293             DELETE_4:
294                          IF W>0 THEN SKIPS(W) = SKIPS(W)
                                 + SKIPS(TD);
295             DELETE_5:    IF LLINK(FT) = TD THEN LLINK(FT)=W;
297                             ELSE RLINK(FT) = W;
298                          CALL FREENODE(TD);
                  /*  3.1  3.1  3.1  3.1  3.1  3.1  3.1  3.1  3.1  3.1  3.1   */
                  /* # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # */
299                           END     DELETE_ENTRY;
300              DELETE_TEXT:  PROC;
301                           DCL (A(500),ATOP,SSKIP,X) FIXED BIN (31,0);
302                           TOT_KEYS = 0;
                  /* # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # */
                  /*  4.1.2  4.1.2  4.1.2  4.1.2  4.1.2  4.1.2  4.1.2  4.1.2  */
303             D_1:         SSKIP = 0; ATOP=0; X=1;
306             D_2:         SSKIP=SSKIP+SKIPS(X);  ATOP=ATOP+1; A(ATOP)=X;
                     /*      ************************************************** */
                       PUT EDIT ('PREORDER VISIT ***',X,'***          ') (A,F(4),A);
                         /*       */
309                           CALL DELETE_TEXT_T1;
310             D_3:          IF LLINK(X) > 0 THEN DO; X=LLINK(X); GO TO D_2; END;
315             D_4:
316                                                  X=A(ATOP); ATOP=ATOP-1;
317             D_5:          IF X<0 THEN GO TO D_6; ATOP=ATOP+1; A(ATOP)=-X;GO TO D_7;
322             D_6:          X=ABS(X);
                       /* ************************************************
                      PUT EDIT ('ENDORDER VISIT ***',X,'***          ') (A,F(4),A);
                        ************************************************  */
                         /*       */
323                           CALL DELETE_TEXT_T2;  SSKIP=SSKIP-SKIPS(X);
                     /*  ************************************************************
                          PUT LIST (' --- TREE AFTER ENDORDER VISIT --');
```

```
                                CALL PRINT_TREE(18);
                        ************************************************    */
                                /*      */
325                                GO TO D_4;
326                 D_7:
327                       IF X = 1 THEN GO TO D_EXIT;
328                          IF RLINK(X) <0 THEN GO TO D_4;
330                           X=RLINK(X); GO TO D_2;
332                 DELETE_TEXT_T1:   PROC;
333                  T1_1:            IF PTR(X) >= END THEN DO;
335                                   PTR(X) = PTR(X)-SHIFT; GO TO EXIT_T1; END;
338                 T1_2:            IF PTR(X) >= START THEN PTR(X) = 0;
340                 EXIT_T1:         END  DELETE_TEXT_T1;
341                 DELETE_TEXT_T2:   PROC;
342                                  DCL FINISHED  FIXED BIN (31,0);
343                                   FINISHED = 0;
344                 T2_1:            IF LLINK(X) <0 THEN DO; PD=ABS(LLINK(X)); GO TO T2_4;END;
349                 T2_2:            IF FINISHED = 1 THEN GO TO T2_EXIT;  FINISHED = 1;
352                 T2_3:            IF RLINK(X)<0 THEN DO; PD=ABS(RLINK(X)); GO TO T2_4; END;
357                                 GO TO T2_EXIT;
358                 T2_4:
359                                  IF PTR(PD) >= START THEN GO TO T2_2;
360                 T2_5:            IF PTR(PD) = 0 THEN DO;
362                                  TD=X; FT=ABS(A(ATOP)); CALL FIND_IT; GO TO T2_EXIT;END;
367                 T2_6:            IF PTR(PD) + SSKIP/8 <  START THEN GO TO T2_2;
369                 T2_7:            FT= ABS(A(ATOP)); TD=X; TOT_KEYS= TOT_KEYS + 1;
372                                    KEY_LIST(TOT_KEYS) = PTR(PD);
373                                    CALL FIND_IT;
374                 T2_EXIT:         END  DELETE_TEXT_T2;
375                   D_EXIT:
                                     /*  REINSERT THE MODIFIED KEYS  */
376                                 KK=0;   CALL INSERT_KEY;
                      /*  4.1.2  4.1.2  4.1.2  4.1.2  4.1.2  4.1.2  4.1.2  4.1.2  */
                      /* # # # # # # # # # # # # # # # # # # # # # # # # # # # # # */
377                                  END  DELETE_TEXT;
378                 SEARCH:          PROC;
379                       DCL  LL FIXED BIN (31,0) STATIC;
380                                  IF FLAG ¬= 0 THEN
381                                 WHAT_BITS = '-';
                      /* # # # # # # # # # # # # # # # # # # # # # # # # # # # # # */
                      /*  2.1  2.1  2.1  2.1  2.1  2.1  2.1  2.1  2.1  2.1  2.1  */
382                 ONE:             P = 1; J = 0;
384                  TWO:                 IF P = AA THEN AA = 0;
386                                   W=Q; Q=P;  PP=LLINK(P); P=ABS(PP);
390                                    IF LLINK(Q) <  0 THEN GO TO SIX;
392                  THREE:
                                     TOT_SEARCHES = TOT_SEARCHES + 1;
```

```
393                     J = J + SKIPS(P); IF J > N THEN GO TO SIX;
396          FOUR:       IF FLAG ¬= 0 THEN IF J<200 THEN
398                      WHAT_BITS(J) = SUBSTR(KEY_BITS,J,1);
399                     IF SUBSTR(KEY_BITS,J,1) = '0'B THEN GO TO TWO;
401          FIVE:
402                           IF P = AA THEN AA = 0;
403                      W=Q; Q=P;  PP=RLINK(P); P=ABS(PP);
407                      IF RLINK(Q) < 0 THEN GO TO SIX;
409                      GO TO THREE;
410        SIX: IF FLAG = 1 THEN DO;
412                     PUT EDIT ((WHAT_BITS(II) DO II=1 TO J WHILE (II<193)))
                              (SKIP(2), (12)(X(1),(8)A(1))):
413                     END;
414                      COMPARE = SUBSTR(CHARS,PTR(P),N/8+2)        :
415                      MATCH = 1;
416                    L=N;
417                      DO LL=0 TO (N-1)/8;
418                         IF SUBSTR(COMPARE,LL+1,1) ¬= SUBSTR(KEY,LL+1,1)
                         THEN
419                      DO LL=LL+8 TO N-1;
420                      IF SUBSTR(KEY_BITS,LL+1,1) ¬= SUBSTR(BITS,LL+1,1)
421                      THEN DO ; MATCH = 0; L=LL; GO TO SETL; END;
426                       END;   END;
                    /*   ************************    */
428                 SETL:
                      /*  2.1  2.1  2.1  2.1  2.1  2.1  2.1  2.1  2.1  2.1  2.1  */
                      /* # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # */
                          RETURN;
429                     END SEARCH;
430                 INSERT_KEY:    PROC;
431                       DCL  (TP,Y,Z)   FIXED BIN (31,0);
432                  LOOP:
433                      KK=KK+1; IF KK> TOT_KEYS THEN GO TO ALLBUILT;
435                      POSITION = KEY_LIST(KK);
                      /* # # # # # # # # # # # # # # # # # # # # # # # # # # # # # */
                      /*  2.2  2.2  2.2  2.2  2.2  2.2  2.2  2.2  2.2  2.2  2.2  */
436                   S1:          KEY = SUBSTR(CHARS,POSITION,1000);
437                   S2:          CALL GETNODE(P);
438                   S3:          PTR(P) = POSITION; N=(I-POSITION+1)*8;
440                                IF N>8000 THEN N=8000;
442                                   CALL SEARCH;
443                                 IF L < J THEN DO; N = L; CALL SEARCH;  END;
448                   S4:          IF ABS(LLINK(Q)) = P THEN DO;
450                                     T = LLINK(Q);  LLINK(Q) = R; END;
453                                ELSE DO;  T = RLINK(Q);  RLINK(Q) = R; END;
457                                 IF THREAD = 1 THEN GO TO S5_THREAD;
459                   S5:          IF SUBSTR(KEY_BITS,L+1,1) = '0'B THEN DO;
```

```
461                          LLINK(R) = -R;  RLINK(P) = P*SIGN(T);  END;
464                       ELSE DO;  RLINK(R) = -P;  LLINK(R) = P*SIGN(T);  END;
468                        GO TO S6;
                        /*   THE RIGHT THREADING INSERTION    */
                       /*  5.1.1  5.1.1  5.1.1  5.1.1  5.1.1  5.1.1  5.1.1  5.1.1  */
469                    S5_THREAD:    IF SUBSTR(KEY_BITS,L+1,1) = '0'B THEN DO;
471                                    LLINK(R) = -R; RLINK(R) =  P*SIGN(T) ; GO TO S6; END;
475                                  ELSE DO ; Y= P*SIGN(T); Z=R;
478                                     END;
479                    S5_1:          IF Y>0 THEN DO;Z=Y; Y=RLINK(Y);
483                                   THREAD_SEARCHES = THREAD_SEARCHES + 1; GO TO S5_1; END;
486                                   TP=PTR(ABS(Y)); PTR(ABS(Y))= PTR(R); PTR(R) = TP;
489                                    RLINK(Z) = -P;
490                    S5_2:          RLINK(R) = Y ;
491                                   IF T>0 THEN LLINK(R) = P; ELSE LLINK(R) = -R;
                       /*  5.1.1  5.1.1  5.1.1  5.1.1  5.1.1  5.1.1  5.1.1  5.1.1  */
                          /*   END OF THE RIGHT THREADING INSERTION    */
494                    S6:           IF T < 0 THEN SKIPS(R) = L+1-J;
496                                  ELSE DO;  SKIPS(R) = L+1-J+SKIPS(P);
498                                     SKIPS(P) = J-(L+1);  END;
500                       GO TO LOOP;
                       /*  2.2  2.2  2.2  2.2  2.2  2.2  2.2  2.2  2.2  2.2  2.2  */
                       /* # # # # # # # # # # # # # # # # # # # # # # # # # # # # # */
501                    ALLBUILT:       END   INSERT_KEY;
502                     GETNODE:       PROC  (X);
503                                     DCL   X   FIXED BIN (31,0);
504                    G1:            IF LLINK(M) = M THEN GO TO G4;
506                    G2:            X=RLINK(M);
507                    G3:            LLINK(RLINK(X))=M;
508                                   RLINK(LLINK(X))=RLINK(X);
509                                    RETURN;
510                    G4:            IF MARKER = M-1 THEN PUT LIST(' *****NO ROOM *****');
512                                    ELSE DO ; MARKER = MARKER + 1;X= MARKER; RETURN;
516                                  END;
517                    FREENODE:       ENTRY(X)
                                     ;
518                    F1:            LLINK(X) = M; RLINK(X) = RLINK(M);
520                    F2:            LLINK(RLINK(M)) = X;  RLINK(M) = X; RETURN;
523                                     END     GETNODE;
524                    TRANSFORM:      PROC;
525                                   DCL (A(100),ATOP,X,I,J,K,B)  FIXED BIN (31,0);
                       /* # # # # # # # # # # # # # # # # # # # # # # # # # # # # # */
                       /*  5.4.3  5.4.3  5.4.3  5.4.3  5.4.3  5.4.3  5.4.3  5.4.3  */
526                    T1:            ATOP = 0; B=0; X=1;
529                    T2:
530                                   IF RLINK(X) < 0 THEN GO TO T3; ELSE DO;
532                                     ATOP = ATOP+1; A(ATOP) = ABS(RLINK(X)); END;
```

```
535                T3:          RLINK(X) = B*SIGN(RLINK(X));      B=X;
537                T4:          IF ABS(LLINK(X)) ¬= X THEN DO; X=LLINK(X);GO TO T2;END;
542                T5:          IF ATOP = 0 THEN DO; LLINK(X) = -1; RLINK(1) = -X;
546                                   GO TO T6;   END              ; ELSE DO;
549                             Z=A(ATOP); ATOP=ATOP-1; LLINK(X) = -Z; X=Z;GO TO T2; END;
555                T6:          I=1; ATOP=1; A(ATOP) = 0; GO TO T8;
559                T7:
560                             J=ABS(LLINK(I)); I=I+1; IF I>=J THEN GO TO T8; ELSE DO;
564                             X=ABS(RLINK(I));LLINK(X) = J*SIGN(LLINK(X));
566                             X=ABS(LLINK(I)); RLINK(X)=J*SIGN(RLINK(X));
568                               CALL SWAP(RLINK(I),RLINK(J),LLINK(I),LLINK(J),
569                                 SKIPS(I),SKIPS(J),PTR(I),PTR(J));  END;
570                T8:
571                             IF LLINK(I) >= 0 THEN DO; ATOP=ATOP+1; A(ATOP) = I   ;
574                             GO TO T7; END;
576                T9:          J=I;
577                T10:         IF RLINK(J) >=0 THEN GO TO T12;
579                T11:         K=A(ATOP); ATOP = ATOP-1; RLINK(J) = K*SIGN(LLINK(J));
582                             J=K; IF ATOP = 0 THEN GO TO V;ELSE GO TO T10;
586                T12:         RLINK(J) = (I+1)*SIGN(LLINK(J)); GO TO T7;
                    /*  5.4.3  5.4.3  5.4.3  5.4.3  5.4.3  5.4.3  5.4.3  5.4.3  */
                    /* # # # # # # # # # # # # # # # # # # # # # # # # # # # # # */
588                V:
                                  PUT PAGE;
589                       PUT SKIP;
590                       PUT SKIP LIST (' THE PREORDER SEQUENTIAL FORM IS GIVEN BELOW. T
                  HE FIELDS LISTED ARE');
591                    PUT SKIP LIST  ('        LOCATION','        LINK','        SKIP',
                       '   START OF KEY IN TEXT');
592                       PUT SKIP LIST(('--' DO J=1 TO 5));
593                            DO J=1 TO I;
594                  PUT SKIP LIST (J,LINK(J),SKIPS(J),SUBSTR(CHARS,PTR(J),30));   END;
596                            RETURN;
597                                 END TRANSFORM;
598                SWAP:    PROC (A,B,C,D,E,F,G,H);
599                     DCL(A,B,C,D,E,F)   FIXED BIN (15,0), (G,H) FIXED BIN (21,0);
600                                X=A;A=B;B=X;X=C;C=D;D=X;X=E;E=F;F=X;X=G;G=H;H=X;
612                          RETURN;      END  SWAP;
614                SEQUENTIAL_TRAVERSE:  PROC(X);
615                          DCL  (X,Y,Z)  FIXED BIN (31,0);
616                               PUT PAGE;
617                       PUT SKIP LIST(' THE STRUCTURE IS NOW TRAVERSED IN POSTORDER.',
                               ' ONLY THE TEXT HAS BEEN LISTED.');
618                               PUT SKIP EDIT ('- - - - - - - - - - - - - - -')(A);
619                       PUT SKIP;
                    /* # # # # # # # # # # # # # # # # # # # # # # # # # # # # # */
                    /*  5.4.2  5.4.2  5.4.2  5.4.2  5.4.2  5.4.2  5.4.2  5.4.2  */
```

```
620                 S1:           Y=X;
621                 S2:           IF LINK(Y) >=0 THEN DO ; Y=Y+1; GO TO S2; END;
626                 S3:           PUT SKIP EDIT (SUBSTR(CHARS,PTR(Y),30))(A);
627                 S4:           Z=ABS(LINK(Y));IF Z<X THEN RETURN     ;
630                 S5:           IF Z<Y THEN DO; Y=Z; GO TO S3; END; ELSE DO;
636                               Y=Z; GO TO S2; END;
                      /*  5.4.2  5.4.2  5.4.2  5.4.2  5.4.2  5.4.2  5.4.2  5.4.2  */
                      /* # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # */
639                               END  SEQUENTIAL_TRAVERSE;
640                 PRINT_TEXT:  PROC;
641                           DO II = 0 TO I BY 100;
642                              IF((II/1000)*1000-II) = 0 THEN
643                            DO;
644                            PUT SKIP EDIT(('-' DO L=1 TO 100),' APPROX.')
                                (X(11),(100)A(1),A);
645                            PUT SKIP EDIT ((L,'*' DO L= 5 TO 100 BY 5),'  CARD  ')
                              (X(11),(20)(F(4),A(1)),A);
646                           PUT SKIP EDIT(('-' DO L=1 TO 100),' NUMBER ')
                             (X(11),(100)A(1),A);
647                            . END;
648                           PUT SKIP EDIT (II,SUBSTR(CHARS,II+1,100),II/80+1)
                                 (X(3),F(5),X(3),A,F(6));
649                             END;
650                       PUT SKIP EDIT (('-' DO L = 1 TO 120))(A)     ;
651                          END     PRINT_TEXT;
652                 PRINT_TREE:  PROCEDURE (Y);
653                           DCL (A(100),ATOP,J,X,Y,Q,R ) FIXED BIN (31,0);  X=Y;
655                           SKELTON = ' '; VERT_SPACE =
                            2;
657                           HORIZ_SPACE =
                             8;
658                           EMPTY = 1;
659                 PRINT_1:  ATOP = EMPTY;
                      /*  SINCE THE ROOT OF THE PATRICIA STRUCTURE HAS NO BLINK, WE
                          MUST NOT ATTEMPT TO TRAVERSE ITS RIGHT SUBTREE */
660                            IF X=1  THEN GO TO PRINT_611;
662               PRINT_2:     IF X<0 THEN GO TO PRINT_4;
664                            ATOP = ATOP + 1; A(ATOP) = Y; X=RLINK(X);
667                            GO TO PRINT_2;
668               PRINT_4:     IF ATOP = EMPTY THEN GO TO PRINT_EXIT;
670                            X=A(ATOP);ATOP=ATOP-1; IF X<0 THEN GO TO PRINT_4;
674               PRINT_61: CALL TEXT_OUT(BLINK(X));
675               PRINT_611:
                           PUT SKIP EDIT (SUBSTR(SKELTON,1,ATOP*HORIZ_SPACE-1),
                                '*',X,'*',SKIPS(X),'*',PTR(X))
                           (A,A(1),F(3),A,F(3),A,F(4));
676                        CALL TEXT_OUT(LLINK(X));
```

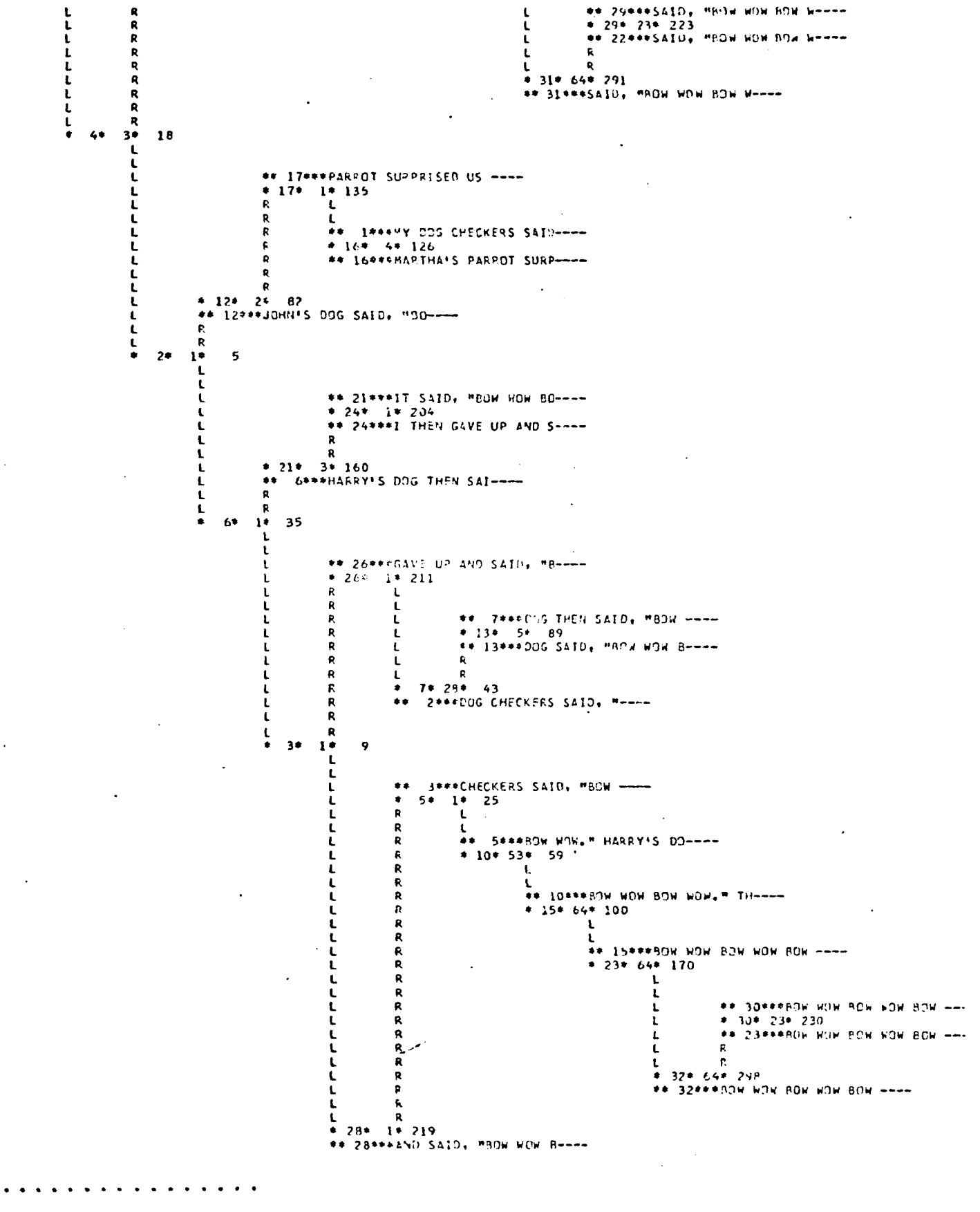
```
677                TEXT_OUT:     PROC   (LINK);
678                                 DCL   LINK   FIXED  BIN(15,0);
679                              IF LINK < 0 THEN DO;Q=PTR(ABS(LINK));
682                              PUT SKIP EDIT (SUBSTR(SKELTON,1,ATOP*HORIZ_SPACE-1),
                                     '**',ABS(LINK),'***',SUBSTR(CHARS,Q,20),'----')
                                  (A,A,F(3),A,A,A);
683                                        END;
684                              END TEXT_OUT;
685                PRINT_62:     IF ATOP ¬=EMPTY THEN IF  RLINK(ABS(A(ATOP)))= X THEN
687                               SUBSTR(SKELTON,ATOP*HORIZ_SPACE,1) = 'R';
688                                    ELSE
688                              SUBSTR(SKELTON,ATOP*HORIZ_SPACE,1) = ' ';
689                PRINT_63:   IF LLINK(X)¬< 0 THEN
690                               SUBSTR(SKELTON,(ATOP+1)*HORIZ_SPACE,1) = 'L';
691                                    ELSE
691                              SUBSTR(SKELTON,(ATOP+1)*HORIZ_SPACE,1) = ' ';
692                PRINT_64:      DO II=1 TO  VERT_SPACE; PUT SKIP EDIT
                               (SUBSTR(SKELTON,1,PRINTPAGE_WIDTH))(A);
694                                    END;
695                PRINT_7:      ATOP = ATOP+1; A(ATOP) = -X; X=LLINK(X);GO TO PRINT_2;
699                  PRINT_EXIT:
                      /*  ************************************************************ */
                                 PUT SKIP LIST(' . . . . . .');
700                               END PRINT_TREE;
701                              END PATRICA;
```

```
                (200*AZ)(200*B)(200*T)A? SURPRISED US WHEN IT )!BOW?BOW/WOW  H?H/H?D/T/M/BOW?!?
        ! ?     (100*THIS SENTENCE WILL START A NEW KEY. )  T?  (101-103)T?
    *SAID, "*BOW WOW BOW WOW BOW WOW BOW WOW BOW WOW."      &
*UP *AND *SAID, "*BOW WOW BOW WOW BOW WOW BOW WOW." ONE MORE FOR GOOD MEASURE:
 JRPRISED *US *WHEN *IT *SAID, "*BOW WOW BOW WOW BOW WOW BOW WOW." *I *THEN *GAVE
WOW." *THEN *JOHN'S *DOG *SAID, "*BOW WOW BOW WOW BOW WOW." *MARTHA'S *PARROT *S
*MY *DOG *CHECKERS *SAID, "*BOW WOW." *HARRY'S *DOG *THEN *SAID, "*BOW WOW BOW
```

```
STATISTICS FOR BUILDING TREE
  NUMBER OF EXTRA NODE ACCESSES REQUIRED TO THREAD=   0
  NUMBER OF SEARCHES= 167
  NUMBER OF KEYS IN THE TREE=  32
```

```
        ----------------------------------------------------------------------------------------------------- APPROX.
           5*  10*  15*  20*  25*  30*  35*  40*  45*  50*  55*  60*  65*  70*  75*  80*  85*  90*  95* 100*  CARD
        ----------------------------------------------------------------------------------------------------- NUMBER
      0   MY DOG CHECKERS SAID, "BOW WOW." HARRY'S DOG THEN SAID, "BOW WOW WOW WOW." THEN JOHN'S DOG SAID, "B      1
    100  OW WOW BOW WOW BOW WOW." MARTHA'S PARROT SURPRISED US WHEN IT SAID, "BOW WOW BOW WOW BOW WOW BOW WOW      2
    200  ." I THEN GAVE UP AND SAID, "BOW WOW BOW WOW BOW WOW BOW WOW." ONE MORE FOR GOOD MEASURE: SAID, "BOW      3
    300   WOW BOW WOW BOW WOW BOW WOW BOW WOW."                                                                  4
```

```
    *  1*  0*   2
          L
          L
          L              ** 20***WHEN IT SAID, "BOW W----
          L              * 20*  1* 155
          L              R       L
          L              R       L
          L              R       ** 19***US WHEN IT SAID, "BO----
          L              R       * 27*  4* 216
          L              R       ** 27***UP AND SAID, "BOW WO----
          L              R
          L              R
          L       * 19*  3* 152
          L       R      L
          L       R      L
          L       R      L       **  8***THEN SAID, "BOW WOW ----
          L       R      L       * 11* 35*  77
          L       R      L       R       L
          L       R      L       R       L
          L       R      L       R       ** 11***THEN JOHN'S DOG SAID----
          L       R      L       R       * 25*  1* 206
          L       R      L       R       ** 25***THEN GAVE UP AND SAI----
          L       R      L       R
          L       R      L       R
          L       R      *  8*  2*  47
          L       R              L
          L       R              L
          L       R              ** 18***SURPRISED US WHEN IT-----
          L       R              * 18*  3* 142
          L       R                      L
          L       R                      L
          L       R                      **  4***SAID, "BOW WOW." HAR----
          L       R                      *  9*106*  52
          L       R                             L
          L       R                             L
          L       R                             **  9**4SAID, "BOW WOW BOW W-----
          L       R                             * 14* 64*  93
          L       R                                    L
          L       R                                    L
          L       R                                    ** 14***SAID, "BOW WOW BOW W----
          L       R                                    * 22* 64* 163
          L       R                                           L
          L       R                                           L
```


L R L ** 29***SAID, "BOW WOW BOW W----
L R L * 29* 23* 223
L R L ** 22***SAID, "BOW WOW BOW W----
L R L R
L R L R
L R * 31* 64* 291
L R ** 31***SAID, "BOW WOW BOW W----
L R
L R
* 4* 3* 18
L
L
L ** 17***PARROT SURPRISED US ----
L * 17* 1* 135
L R L
L R L
L R ** 1***MY DOG CHECKERS SAID----
L R * 16* 4* 126
L R ** 16***MARTHA'S PARROT SURP----
L R
L R
L * 12* 2* 87
L ** 12***JOHN'S DOG SAID, "BO----
L R
L R
* 2* 1* 5
L
L
L ** 21***IT SAID, "BOW WOW BO----
L * 24* 1* 204
L ** 24***I THEN GAVE UP AND S----
L R
L R
L * 21* 3* 160
L ** 6***HARRY'S DOG THEN SAI----
L R
L R
* 6* 1* 35
L
L
L ** 26***GAVE UP AND SAID, "B----
L * 26* 1* 211
L R L
L R L
L R L ** 7***DOG THEN SAID, "BOW ----
L R L * 13* 5* 89
L R L ** 13***DOG SAID, "BOW WOW B----
L R L R
L R L R
L R * 7* 28* 43
L R ** 2***DOG CHECKERS SAID, "----
L R
L R
* 3* 1* 9
L
L
L ** 3***CHECKERS SAID, "BOW ----
L * 5* 1* 25
L R L
L R L
L R ** 5***BOW WOW," HARRY'S DO----
L R * 10* 53* 59
L R L
L R L
L R ** 10***BOW WOW BOW WOW," TH----
L R * 15* 64* 100
L R L
L R L
L R ** 15***BOW WOW BOW WOW BOW ----
L R * 23* 64* 170
L R L
L R L
L R L ** 30***BOW WOW BOW WOW BOW ---
L R L * 30* 23* 230
L R L ** 23***BOW WOW BOW WOW BOW ---
L R L R
L R L R
L R * 32* 64* 298
L R ** 32***BOW WOW BOW WOW BOW ----
L R
L R
* 28* 1* 219
** 28***AND SAID, "BOW WOW B----

. . . . . .

```
*****SEARCHING FOR ALL KEYS STARTING WITH
 WOW BOW WOW." I THEN GAVE UP    AND SAID, "BOW WOW BOW WOW BOW WOW BOW WOW." ONE MORE FOR GOOD MEASURE: SAID, "BOW WOW BOW   219
MORE FOR GOOD MEASURE: SAID, "   BOW WOW BOW WOW BOW WOW BOW WOW."       E                                                  298
T SURPRISED US WHEN IT SAID, "   BOW WOW BOW WOW BOW WOW BOW WOW." I THEN GAVE UP AND SAID, "BOW WOW BOW WOW BOW WOW BOW WO  170
W." I THEN GAVE UP AND SAID, "   BOW WOW BOW WOW BOW WOW BOW WOW." ONE MORE FOR GOOD MEASURE: SAID, "BOW WOW BOW WOW BOW WO  230
 WOW." THEN JOHN'S DOG SAID, "   BOW WOW BOW WOW BOW WOW." MARTHA'S PARROT SURPRISED US WHEN IT SAID, "BOW WOW BOW WOW BOW   100
WOW." HARRY'S DOG THEN SAID, "   BOW WOW BOW WOW." THEN JOHN'S DOG SAID, "BOW WOW BOW WOW BOW WOW." MARTHA'S PARROT SURPRIS    99
          MY DOG CHECKERS SAID, "   BOW WOW." HARRY'S DOG THEN SAID, "BOW WOW BOW WOW." THEN JOHN'S DOG SAID, "BOW WOW BOW WOW   25
                  MY DOG           CHECKERS SAID, "BOW WOW." HARRY'S DOG THEN SAID, "BOW WOW BOW WOW." THEN JOHN'S DOG SAID,     9
                     MY            DOG CHECKERS SAID, "BOW WOW." HARRY'S DOG THEN SAID, "BOW WOW BOW WOW." THEN JOHN'S DOG SA   5
"BOW WOW BOW WOW." THEN JOHN'S   DOG SAID, "BOW WOW BOW WOW BOW WOW." MARTHA'S PARROT SURPRISED US WHEN IT SAID, "BOW WOW B   89
KERS SAID, "BOW WOW." HARRY'S    DOG THEN SAID, "BOW WOW BOW WOW." THEN JOHN'S DOG SAID, "BOW WOW BOW WOW BOW WOW." MARTHA'   43
 WOW BOW WOW BOW WOW." I THEN    GAVE UP AND SAID, "BOW WOW BOW WOW BOW WOW BOW WOW." ONE MORE FOR GOOD MEASURE: SAID, "BOW   211
DOG CHECKERS SAID, "BOW WOW."    HARRY'S DOG THEN SAID, "BOW WOW BOW WOW." THEN JOHN'S DOG SAID, "BOW WOW BOW WOW BOW WOW."   35
WOW BOW WOW BOW WOW BOW WOW."    I THEN GAVE UP AND SAID, "BOW WOW BOW WOW BOW WOW BOW WOW." ONE MORE FOR GOOD MEASURE: SAI   204
HA'S PARROT SURPRISED US WHEN    IT SAID, "BOW WOW BOW WOW BOW WOW BOW WOW." I THEN GAVE UP AND SAID, "BOW WOW BOW WOW BOW   160
SAID, "BOW WOW BOW WOW." THEN    JOHN'S DOG SAID, "BOW WOW BOW WOW BOW WOW." MARTHA'S PARROT SURPRISED US WHEN IT SAID, "BO   92
D, "BOW WOW BOW WOW BOW WOW."    MARTHA'S PARROT SURPRISED US WHEN IT SAID, "BOW WOW BOW WOW BOW WOW BOW WOW." I THEN GAVE    126
                                 MY DOG CHECKERS SAID, "BOW WOW." HARRY'S DOG THEN SAID, "BOW WOW BOW WOW." THEN JOHN'S DOG   2
OW BOW WOW BOW WOW." MARTHA'S    PARROT SURPRISED US WHEN IT SAID, "BOW WOW BOW WOW BOW WOW BOW WOW." I THEN GAVE UP AND SA   135
." ONE MORE FOR GOOD MEASURE:    SAID, "BOW WOW BOW WOW BOW WOW BOW WOW BOW WOW."       E                                   291
S PARROT SURPRISED US WHEN IT    SAID, "BOW WOW BOW WOW BOW WOW BOW WOW." I THEN GAVE UP AND SAID, "BOW WOW BOW WOW BOW WOW   163
 BOW WOW." I THEN GAVE UP AND    SAID, "BOW WOW BOW WOW BOW WOW BOW WOW." ONE MORE FOR GOOD MEASURE: SAID, "BOW WOW BOW WOW   223
WOW BOW WOW." THEN JOHN'S DOG    SAID, "BOW WOW BOW WOW BOW WOW." MARTHA'S PARROT SURPRISED US WHEN IT SAID, "BOW WOW BOW W   93
. "BOW WOW." HARRY'S DOG THEN    SAID, "BOW WOW BOW WOW." THEN JOHN'S DOG SAID, "BOW WOW BOW WOW BOW WOW." MARTHA'S PARROT    52
                  MY DOG CHECKERS SAID, "BOW WOW." HARRY'S DOG THEN SAID, "BOW WOW BOW WOW." THEN JOHN'S DOG SAID, "BOW WOW     18
WOW BOW WOW." MARTHA'S PARROT    SURPRISED US WHEN IT SAID, "BOW WOW BOW WOW BOW WOW BOW WOW." I THEN GAVE UP AND SAID, "BO   142
W BOW WOW BOW WOW BOW WOW." I    THEN GAVE UP AND SAID, "BOW WOW BOW WOW BOW WOW BOW WOW." ONE MORE FOR GOOD MEASURE: SAID,   206
THEN SAID, "BOW WOW BOW WOW."    THEN JOHN'S DOG SAID, "BOW WOW BOW WOW BOW WOW." MARTHA'S PARROT SURPRISED US WHEN IT SAID   77
 SAID, "BOW WOW." HARRY'S DOG    THEN SAID, "BOW WOW BOW WOW." THEN JOHN'S DOG SAID, "BOW WOW BOW WOW BOW WOW." MARTHA'S PA   47
BOW WOW BOW WOW." I THEN GAVE    UP AND SAID, "BOW WOW BOW WOW BOW WOW BOW WOW." ONE MORE FOR GOOD MEASURE: SAID, "BOW WOW   216
W." MARTHA'S PARROT SURPRISED    US WHEN IT SAID, "BOW WOW BOW WOW BOW WOW BOW WOW." I THEN GAVE UP AND SAID, "BOW WOW BOW   152
 MARTHA'S PARROT SURPRISED US    WHEN IT SAID, "BOW WOW BOW WOW BOW WOW BOW WOW." I THEN GAVE UP AND SAID, "BOW WOW BOW WOW   155

***** INSERTING AFTER POSITION  100 THE TEXT ***THIS SENTENCE WILL START A NEW KEY.


*****SEARCHING FOR ALL KEYS STARTING WITH     T
W BOW WOW BOW WOW BOW WOW." I    THEN GAVE UP AND SAID, "BOW WOW BOW WOW BOW WOW BOW WOW." ONE MORE FOR GOOD MEASURE: SAID,   242
THEN SAID, "BOW WOW BOW WOW."    THEN JOHN'S DOG SAID, "BTHIS SENTENCE WILL START A NEW KEY. OW WOW BOW WOW BOW WOW." MARTH   77
 SAID, "BOW WOW." HARRY'S DOG    THEN SAID, "BOW WOW BOW WOW." THEN JOHN'S DOG SAID, "BTHIS SENTENCE WILL START A NEW KEY.      47
WOW." THEN JOHN'S DOG SAID, "B   THIS SENTENCE WILL START A NEW KEY. OW WOW BOW WOW BOW WOW." MARTHA'S PARROT SURPRISED US      101

*****DELETING ALL TEXT BETWEEN  101 AND  103 NAMELY ***THI
```

```
*****SEARCHING FOR ALL KEYS STARTING WITH      T
W BOW WOW BOW WOW BOW WOW." I    THEN GAVE UP AND SAID, "BOW WOW BOW WOW BOW WOW BOW WOW." ONE MORE FOR GOOD MEASURE: SAID,     239
THEN SAID, "BOW WOW BOW WOW."    THEN JOHN'S DOG SAID, "BS SENTENCE WILL START A NEW KEY. OW WOW BOW WOW BOW WOW." MARTHA'S      77
 SAID, "BOW WOW." HARRY'S DOG    THEN SAID, "BOW WOW BOW WOW." THEN JOHN'S DOG SAID, "BS SENTENCE WILL START A NEW KEY. OW       47

***** INSERTING AFTER POSITION   200 THE TEXT ***AZ

***** INSERTING AFTER POSITION   200 THE TEXT ***B

***** INSERTING AFTER POSITION   200 THE TEXT ***T


*****SEARCHING FOR ALL KEYS STARTING WITH      A
 WOW BOW WOW." I THEN GAVE UP    AND SAID, "BOW WOW BOW WOW BOW WOW BOW WOW." ONE MORE FOR GOOD MEASURE: SAID, "BOW WOW BOW     256
T SURPRISED US WHEN IT SAID,TB   AZ "BOW WOW BOW WOW BOW WOW BOW WOW." I THEN GAVE UP AND SAID, "BOW WOW BOW WOW BOW WOW BO     203

*****DELETING THE TEXT     SURPRISED US WHEN IT *** FROM POSITION    175


 THE COMPLETE TEXT IS LISTED BELOW
                ---------------------------------------------------------------------------------------------------- APPROX.
                  5*  10*  15*  20*  25*  30*  35*  40*  45*  50*  55*  60*  65*  70*  75*  80*  85*  90*  95* 100*  CARD
                ---------------------------------------------------------------------------------------------------- NUMBER
       0         MY DOG CHECKERS SAID, "BOW WOW." HARRY'S DOG THEN SAID, "BOW WOW BOW WOW." THEN JOHN'S DOG SAID, "B      1
     100       S SENTENCE WILL START A NEW KEY. OW WOW BOW WOW BOW WOW." MARTHA'S PARROT SAID,TBAZ "BOW WOW BOW WOW      2
     200        BOW WOW BOW WOW." I THEN GAVE UP AND SAID, "BOW WOW BOW WOW BOW WOW BOW WOW." ONE MORE FOR GOOD MEA      3
     300       SURE: SAID, "BOW WOW WOW WOW BOW WOW BOW WOW BOW WOW."       S                                          4
-----------------------------------------------------------------------------------------------------------------------


*****SEARCHING FOR ALL KEYS STARTING WITH      BOW
MORE FOR GOOD MEASURE: SAID, "  BOW WOW BOW WOW BOW WOW BOW WOW BOW WOW."       E                                          314
." MARTHA'S PARROT SAID,TBAZ "  BOW WOW BOW WOW BOW WOW BOW WOW." I THEN GAVE UP AND SAID, "BOW WOW BOW WOW BOW WOW BOW WO     186
W." I THEN GAVE UP AND SAID, "  BOW WOW BOW WOW BOW WOW BOW WOW." ONE MORE FOR GOOD MEASURE: SAID, "BOW WOW BOW WOW BOW WO     246
WOW." HARRY'S DOG THEN SAID, "  BOW WOW BOW WOW." THEN JOHN'S DOG SAID, "BS SENTENCE WILL START A NEW KEY. OW WOW BOW WOW      59
       MY DOG CHECKERS SAID, "  BOW WOW." HARRY'S DOG THEN SAID, "BOW WOW BOW WOW." THEN JOHN'S DOG SAID, "BS SENTENCE WIL      25

*****DELETING ALL KEYS STARTING WITH     BOW


*****SEARCHING FOR ALL KEYS STARTING WITH      WOW  H*** BUT NO MATCH WAS FOUND

*****DELETING ALL KEYS STARTING WITH     H


*****SEARCHING FOR ALL KEYS STARTING WITH      BOW*** BUT NO MATCH WAS FOUND
```

```
                                                                                                                    APPROX.
      5*  10*  15*  20*  25*  30*  35*  40*  45*  50*  55*  60*  65*  70*  75*  80*  85*  90*  95* 100*          CHAR
                                                                                                                    NUMBER
    0  MY DOG CHECKERS SAID, "BOW WOW." HARRY'S DOG THEN SAID, "BOW WOW BOW WOW." THEN JOHN'S DOG SAID, "BO          1
  100  S SENTENCE WILL START A NEW KEY. OW WOW BOW WOW "OW WOW." MARTHA'S PARROT SAID,TBAZ "BOW WOW BOW WOW          2
  200  BOW WOW BOW WOW." I THEN GAVE UP AND SAID, "BOW WOW BOW WOW BOW WOW BOW WOW." ONE MORE FOR GOOD MEA          3
  300  SURE: SAID, "BOW WOW BOW WOW BOW WOW BOW WOW BOW WOW."      E                                               4
```

```
* 1*  0* 168
     L
     L
     L    ** 19***UP AND SAID, "BOW WO----
     L    * 19*  3* 232
     L    R       L
     L    R       L
     L    R       ** 29***SAID,TBAZ "BOW WOW B----
     L    R       * 29* 35* 175
     L    R               L
     L    R               L
     L    R               ** 23***SAID, "BS SENTENCE W----
     L    R               * 23* 26*  93
     L    R                       L
     L    R                       L
     L    R                       **  4***SAID, "BOW WOW." HAR----
     L    R                       *  9* 50*  52
     L    R                               L
     L    R                               L
     L    R                               **  9***SAID, "BOW WOW BOW W----
     L    R                               * 14* 64* 239
     L    R                                       L
     L    R                                       L
     L    R                                       ** 14***SAID, "POW WOW BOW W----
     L    R                                       * 31*128* 307
     L    R                                       ** 31***SAID, "BOW WOW BOW W----
     L    R
     L    R
     *  4*  3*  18
          L
          L
          L       **  1***PARROT SAID,TBAZ "BO----
          L       * 12*  2*  82
          L       ** 12***JOHN'S DOG SAID, "BS----
          L       R
          L       R
          *  2*  1* 227
                  L
                  L
                  **  6***I THEN GAVE UP AND S----
                  *  6*  1* 220
                          L
                          L
                          **  2***GAVE UP AND SAID, "B----
                          *  3*  1*   9
                                  L
                                  L
                                  L       **  3***CHECKERS SAID, "BOW ----
                                  L       *  5*  1* 181
                                  L       R       L
                                  L       R       L
                                  L       R       ** 33***BS SENTENCE WILL STA----
                                  L       R       * 33*  3* 100
                                  L       R       **  5***BAZ "BOW WOW BOW WOW----
                                  L       R
                                  L       R
                                  * 28*  1* 235
                                          L
                                          L
                                          ** 22***AZ "BOW WOW BOW WOW ----
                                          * 22*  4* 182
                                          ** 28***AND SAID, "BOW WOW B----
```

. . . . . .

```
*****SEARCHING FOR ALL KEYS STARTING WITH
 WOW BOW WOW." I THEN GAVE UP   AND SAID, "BOW WOW BOW WOW BOW WOW BOW WOW." ONE MORE FOR GOOD MEASURE: SAID, "BOW WOW WOW
 WOW." MARTHA'S PARROT SAID,TB  AZ "BOW WOW BOW WOW BOW WOW BOW WOW." I THEN GAVE UP AND SAID, "BOW WOW BOW WOW BOW WOW BO
W WOW." MARTHA'S PARROT SAID,T  OZ "BOW WOW BOW WOW BOW WOW BOW WOW." I THEN GAVE UP AND SAID, "BOW WOW BOW WOW WOW WOW B
 WOW." THEN JOHN'S DOG SAID, "  BS SENTENCE WILL START A NEW KEY. OW WOW BOW WOW WOW WOW." MARTHA'S PARROT SAID,TBAZ "BOW
                    MY DOG      CHECKERS SAID, "BOW WOW." HARRY'S DOG THEN SAID, "BOW WOW BOW WOW." THEN JOHN'S DOG SAID,
 WOW BOW WOW BOW WOW." I THEN   GAVE UP AND SAID, "BOW WOW BOW WOW BOW WOW BOW WOW." ONE MORE FOR GOOD MEASURE: SAID, "BOW
WOW BOW WOW BOW WOW BOW WOW."  I THEN GAVE UP AND SAID, "BOW WOW WOW WOW BOW WOW BOW WOW." ONE MORE FOR GOOD MEASURE: SAI
 SAID, "BOW WOW BOW WOW." THEN  JOHN'S DOG SAID, "BS SENTENCE WILL START A NEW KEY. OW WOW BOW WOW BOW WOW." MARTHA'S PARR
OW BOW WOW BOW WOW." MARTHA'S   PARROT SAID,TBAZ "BOW WOW BOW WOW BOW WOW BOW WOW." I THEN GAVE UP AND SAID, "BOW WOW BOW
 ." ONE MORE FOR GOOD MEASURE:  SAID, "BOW WOW BOW WOW BOW WOW BOW WOW."      E
 BOW WOW." I THEN GAVE UP AND   SAID, "BOW WOW BOW WOW BOW WOW WOW WOW." ONE MORE FOR GOOD MEASURE: SAID, "BOW WOW BOW WOW
. "BOW WOW." HARRY'S DOG THEN   SAID, "BOW WOW BOW WOW." THEN JOHN'S DOG SAID, "BS SENTENCE WILL START A NEW KEY. OW WOW B
                MY DOG CHECKERS SAID, "BOW WOW." HARRY'S DOG THEN SAID, "BOW WOW BOW WOW." THEN JOHN'S DOG SAID, "BS SENTE
WOW BOW WOW." THEN JOHN'S DOG   SAID, "BS SENTENCE WILL START A NEW KEY. OW WOW WOW WOW BOW WOW." MARTHA'S PARROT SAID,TBA
WOW BOW WOW." MARTHA'S PARROT   SAID,TBAZ "BOW WOW BOW WOW BOW WOW BOW WOW." I THEN GAVE UP AND SAID, "BOW WOW BOW WOW BOW
BOW WOW BOW WOW." I THEN GAVE   UP AND SAID, "BOW WOW BOW WOW BOW WOW BOW WOW." ONE MORE FOR GOOD MEASURE: SAID, "BOW WOW
```

C.0 APPENDIX - TIMING CONSIDERATIONS

This appendix gives estimates of execution times for most of the important PATRICIA algorithms. To obtain these estimates several algorithms have been analyzed in terms of 360/50 operations. The estimates are close, but are not to be considered exact, since some of the 360 operations are variable, depending upon the operands involved. If the user wishes to convert to another machine, consider the average instruction execution time for the 360/50 to be 5 microseconds, and the instructions to be ADD, SHIFT, AND, COMPARE, LOAD, STORE, and BRANCH, which are the predominant operations of all PATRICIA algorithms.

C.1 Timing for the Search Loop of Algorithm 2.1.

The 360/50 instructions required for each bit comparison cycle, along with their times, are given in figure C-1. The comparison cycle includes steps 2-5 of the algorithm. Step 1 and Step 6 are each executed only once. A search is found to take 63 microseconds on a 360/50. Thus, for example, in a relatively balanced tree of 100,000 nodes, we would be able to access any node in less than 1 millisecond.

C.2 Timing for a Traversal Iteration

The timing for a traversal loop of a postorder traversal is given in figure C-2, which shows the 360/50 machine code necessary to effect an iteration using algorithm 2.3. The main loop of the algorithm was used, except for steps which are concerned with backward pointers to

the structure and the actual "visit." The time in figure C-2 of 36 microseconds means that we could traverse a PATRICIA tree of 100,000 nodes in 3.6 seconds on a 360/50.

C.3 Timing for Other Algorithms

Using the above technique, we may obtain time estimates for other important PATRICIA algorithms. A table of algorithm execution times (including an estimate of the time required on a 360/50 with a 2314 disk) is given in figure C-3.

'STEP IN ALGORITHM 2.1	360 OPERATION	360/50 TIME MICROSECONDS	COMMENTS
3	L	4	fetch SKIP field
3	AR	3	accumulate sum of SKIP fields
3	C	4	compare to see if we are beyond the
3	BC	4	number of bits in the key
4	LR	3	save accumulated SKIP (we need it twice)
4	SRDL	5	divide by 8 to get byte position
4	N	6	mask out all but remainder, which we use as an index (this is the second use of the accumulated SKIP)
4	L	4	get proper word of key
4	N	6	mask out all but proper bit (the mask is determined by the index we obtained above)
4	C	4	compare with a '1'
4	BC	4	branch to right or left
2,5	L	4	get RLINK or LLINK
2,5	C	4	compare sign to see if we are at the
2,5	BC	4	end of the search
2,5	B	4	loop
	total =	63	

Figure C-1. Timing for the loop in algorithm 2.1.
On a 360/50, a comparison can be made every 63 microseconds.

STEP IN ALGORITHM 2.3	360 OPERATION	360/50 TIME MICROSECONDS	COMMENTS
5	A	4	ATOP ← ATOP + 1
5	ST	4	put X on the stack
2	C	4	check to see if
2	BC	4	we are at a terminal node
4	BC	4	see if stack is empty
6	L	4	unstack X
6	S	4	ATOP ← ATOP - 1
5,6	L	4	X ← LLINK(X) or X ← RLINK(X)
5,6	B	4	loop
	total =	36	

Figure C-2. Timing for a traversal loop using algorithm 2.3.
Omitting the visit, a node can be traversed every 36 microseconds.

Algorithm	Function	Symbol	Estimated Formula for Time	Estimated Actual Time (seconds) for 360/50, 2314 disk, 100,000 keys
2.1	Search (one iteration)	L		.000063
2.3	Traversal (one iteration)	T		.000036
2.3	Traverse tree (no visits)		pT	3.6
	Disk Access Time	X		.075
2.1	Search for key, do not Retrieve		(n-1) L	.001
2.1	Search for, Retrieve a key		X+(n-1) L	.076
2.2	Build PATRICIA tree		$\frac{n}{2}$pL+2pX	15,000
3.1	Delete a Node	D3		.00005
3.2.1	Find PD, TD, etc.	F3	T $\frac{n-2}{2}$+(n-1)L	.001
	Total for a node deletion		F3+D3	.001
3.3.2	Prepare for Subtree Deletion	A3	$\frac{\#keys}{2}$ T+(n-1)L	.0013 average (5-50 keys)
	Total for deleting a subtree of keys		A3+D3	.002
4.1.2	One iteration only	I4		.0002 ave
4.1.3	An entire text alteration operation		p I4+αX (α depends upon "book" size and can be made equal to 1. See section 6.3.1)	20
5.4.3	One iteration	A5		.0003
5.4.3	Convert entire structure to preorder sequential		A5p	30

Figure C-3. Timing of most PATRICIA algorithms. Assume $p=2^n-1$ keys in the tree.

D.0 APPENDIX - THEOREMS AND PROOFS

This appendix presents some useful theorems and their proofs.

Given: 1) α, β, γ, δ, are PATRICIA keys.

2) T is a PATRICIA tree.

3) All the keys α, β, γ, δ, are unique.

Let $\phi(\alpha, \delta)$ express the number of identical leading bits in the two keys δ and α .

Theorem For any $\alpha \notin T$, algorithm 2.1 finds a unique $\gamma \in T$ such that

$$\phi(\alpha, \gamma) \geq \phi(\alpha, \beta) \quad \text{for all } \beta \in T$$

Proof (induction)

1) obviously true for a tree of one node

2) assume true for an arbitrary tree -

we will show that if α is inserted,

the property is preserved.

let B_1, B_2, $B_3 \ldots B_n$ be the bit positions looked at by algorithm 2.1 in our search path to γ; N_{B_1}, N_{B_2}, $\ldots N_{B_n}$ are the nodes encountered.

a) (contradiction)

Assume $\exists$ a β s.t. $\phi(\alpha, \beta) > \phi(\alpha, \gamma)$

$\Rightarrow \exists$ a search path, N_{B_1}, N_{B_2}, $\ldots N_{B_i}$, N_{Q_1}, N_{Q_2}, $\ldots N_{Q_n}$

leading to β which first differs from the path to

γ in node $N_{Q_1} \Rightarrow$ bit comparison at B_i

got us to N_{Q_1} instead of $N_{B_{i+1}}$ $\Rightarrow$ B_ith bit

of $\beta \neq B_i$th bit of γ.

If $B_i < \phi(\alpha, \gamma)$

$\Rightarrow$ $\phi(\alpha, \beta) \leq B_i < \phi(\alpha, \gamma)$ which contradicts

assumption a)

Now, note that algorithms 2.1 and 2.2, when creating the search path for a new key, insert the comparison at the first bit where this key differs from all others on the search path. But our inductive assumption states that this is the longest sequence of identical bits for the two keys.

We thus have:

$\delta_1 = \delta_2$ through the first B_i bits, for all keys

δ_1, δ_2 in the subtree at N_{P_i}. Thus, if $\phi(\alpha, \gamma) \leq B_i$ then $\phi(\alpha,\gamma) = \phi(\alpha, \beta)$, which contradicts assumption a).

Corollary For a given node, N_{B_i}, the first B_i bits of all keys in the subtree at N_{B_i} are identical.

Corollary For a given set of unique PATRICIA keys there exists one and only one PATRICIA tree.

Proof The comparisons that separate all keys are at the longest leading identical bit strings. Since all keys are unique, only one structure of comparisons exists.

Corollary Algorithms 2.1 and 2.2 build the PATRICIA tree in such a way that a postorder traversal presents the keys in ascending numeric order.

Proof Given a PATRICIA tree. Since algorithm 2.1 goes to the left if a "0" is encountered and to the right for a "1" then for a given node, N_{B_i}, any node, $N_{B_{i+j}}$, farther down the search path and to the left of N_{B_i} is numerically less than any node farther down the search path and to the right. If we apply this recursively for all nodes, we have defined the "visit" of a postorder traversal to mean "is less than."

Theorem In any PATRICIA subtree, there exists exactly one backward pointer to a node outside the subtree. (Assume the header node is not contained in any subtree)

Proof (induction)

For a tree of one node, R, (along with header node, H) algorithm 2.2 constructs the link fields such that one pointer of R points to R, the other points to H.

Assume theorem is true for a PATRICIA tree of n nodes. Use algorithm 2.2 to insert a new node, R. Call R's father"Q"and R's son (if any)"P". Then step 5 will set one of R's link fields equal to R (as a backward pointer). Following this:

a) If R is a terminal node.

The other link is pointed back to the node where the replaced link of Q pointed. This preserves the structure specificed by

the theorem for subtrees at Q and R. No other subtrees are affected.

b) If R is a non-terminal node.

The other link is pointed to P, which was the successor of Q. Again, the structure is preserved.

Corollary All backward pointers in any PATRICIA subtree, except for the backward pointer referred to above, are contained within the subtree.

Corollary The single backward pointer that points outside the subtree points to an ancestor of the subtree.

Theorem Algorithm 3.1 properly deletes a PTR field and a bit comparison.

Proof First define " → " as "is the father of," and " → → " as "is the ancestor of (but not the father of)"
Then, enumerate all the node structures and their corresponding types in terms of the notation introduced section 3.0.

PD	→ →	TD	→ →	TT	-	Type 2a
PD	→ →	TD	→	TT	-	2a
PD	→ →	TD	=	TT	-	2c
PD	→	TD	→ →	TT	-	2b
PD	→	TD	→	TT	-	2b
PD	→	TD	=	TT	-	2d
PD	=	TD	→	TT	-	cannot construct
PD	=	TD	→	TT	-	cannot construct
PD	=	TD	=	TT	-	2e 2f

No other relationships between PD, TD, and TT exist (e.g. TD → PD) due to their definitions. Also, note that FT is defined by the position of TT. Now, simply run each configuration through algorithm 3.1.

Corollary Algorithm 3.1 preserves a Right Threaded PATRICIA tree.

E.0. APPENDIX - GLOSSARY

The terms described in Section 1.1 are reproduced here for convenience.

AVAILABLE LIST--A list of empty nodes. (A process which requires space for a new node can always get one by picking the top or bottom node from an available list.)

AVAIL LIST--Identical to an available list.

ANCESTOR--Within a tree, an ancestor of node X is on a path between node X and the root of the tree.

BACKWARD POINTER--A link field in a PATRICIA node, X, that points to X or to some ancestor of X.

BINARY TREE--A data structure in which each node has no more than two nodes hanging from it. These two nodes are commonly called "ROOTS of LEFT and RIGHT SUBTREES."

EBCDIC--A specific internal code where 8 bits represent one character within the computer. For example, the EBCDIC value of of "A" is binary "11000001."

ENDORDER TRAVERSAL--A method of looking at all the nodes of a binary tree in which we first look at all the nodes in the left subtree of a node, then all the nodes in the right subtree of the node, and finally, the node itself. Each node is "looked at" exactly once, although the algorithm for effecting an endorder traversal may actually pass by the node more than once.

FIELD--The smallest entity of information contained in a node. A field may be one or more binary bits in size.

KEY--A contiguous string of characters constituting a word or phrase that we wish to search for and, hence, use in some comparison scheme.

LAMBDA ("λ")--See NULL POINTER.

LEFT LINK--In a binary tree, the link field pointing to the left subtree of the node.

LINK--The specific field of a node that points to the next node in a list. (Actually, a given node can point to more than one node: for example, a node in a binary tree can point to two other nodes.)

LIST--A series of nodes which are physically stored at random, but which have an order that is specified by the LINK fields.

NODE--An entity of information. It will consist of one or more fields. (An example--a node could be likened to a single library catalogue card, and a field to an individual entry on the card, such as the author's name.)

NULL POINTER--(Sc[illegible]A" or "λ"). A specifically valued link field that[illegible] in a list. When any link field points to no ot[illegible]ue called "λ" (frequently zero). We sometimes[illegible]o λ."

POINTER--Has[illegible] except sometimes a pointer is not conta[illegible]

POSTORDER TRA[illegible].t all the nodes of a binary tree in which [illegible]des in the left subtree of a given node; then [illegible], we look at all the nodes in the right subtree of [illegible]

PREORDER TRAVERSAL--Still another method of looking at all the nodes in a binary tree, in which we first look at the node, then the nodes in its left subtree, and finally the nodes in its right subtree.

RIGHT LINK--In a binary tree, the link field pointing to the left subtree of the node.

RIGHT THREADED BINARY TREE--A binary tree in which the right links of terminal nodes point to the next node that would be visited if we were traversing the tree in postorder.

ROOT--In a tree, the node from which all other nodes hang. (Thus, computer trees are usually upsidedown.)

SUBTREE--A branch of a tree. Pick any node in a tree--it is the root of a subtree.

TERMINAL NODE--A node in a binary tree that has no left and/or right subtree. In a PATRICIA tree, the affected right or left link then becomes a backward pointer.

THREAD--The same as a backward pointer.

VISIT--A term for what we do when we "look at" a node during a preorder, postorder, or endorder traversal. Usually a visit involves performing an algorithm, or printing out information.

E.O. APPENDIX - GLOSSARY

The terms described in Section 1.1 are reproduced here for convenience.

AVAILABLE LIST--A list of empty nodes. (A process which requires space for a new node can always get one by picking the top or bottom node from an available list.)

AVAIL LIST--Identical to an available list.

ANCESTOR--Within a tree, an ancestor of node X is on a path between node X and the root of the tree.

BACKWARD POINTER--A link field in a PATRICIA node, X, that points to X or to some ancestor of X.

BINARY TREE--A data structure in which each node has no more than two nodes hanging from it. These two nodes are commonly called "ROOTS of LEFT and RIGHT SUBTREES."

EBCDIC--A specific internal code where 8 bits represent one character within the computer. For example, the EBCDIC value of of "A" is binary "11000001."

ENDORDER TRAVERSAL--A method of looking at all the nodes of a binary tree in which we first look at all the nodes in the left subtree of a node, then all the nodes in the right subtree of the node, and finally, the node itself. Each node is "looked at" exactly once, although the algorithm for effecting an endorder traversal may actually pass by the node more than once.

FIELD--The smallest entity of information contained in a node. A field may be one or more binary bits in size.

KEY--A contiguous string of characters constituting a word or phrase that we wish to search for and, hence, use in some comparison scheme.

LAMBDA ("λ")--See NULL POINTER.

LEFT LINK--In a binary tree, the link field pointing to the left subtree of the node.

LINK--The specific field of a node that points to the next node in a list. (Actually, a given node can point to more than one node: for example, a node in a binary tree can point to two other nodes.)

LIST--A series of nodes which are physically stored at random, but which have an order that is specified by the LINK fields.

NODE--An entity of information. It will consist of one or more fields. (An example--a node could be likened to a single library catalogue card, and a field to an individual entry on the card, such as the author's name.)

NULL POINTER--(Sometimes called "LAMBDA" or "λ"). A specifically valued link field that indicates the last node in a list. When any link field points to no other node, it is given a value called "λ" (frequently zero). We sometimes say that this link "points to λ."

POINTER--Has the same function as a link, except sometimes a pointer is not contained in any node.

POSTORDER TRAVERSAL--A method of looking at all the nodes of a binary tree in which we first look at all the nodes in the left subtree of a given node; then we look at the node; then we look at all the nodes in the right subtree of the node.

PREORDER TRAVERSAL--Still another method of looking at all the nodes in a binary tree, in which we first look at the node, then the nodes in its left subtree, and finally the nodes in its right subtree.

RIGHT LINK--In a binary tree, the link field pointing to the left subtree of the node.

RIGHT THREADED BINARY TREE--A binary tree in which the right links of terminal nodes point to the next node that would be visited if we were traversing the tree in postorder.

ROOT--In a tree, the node from which all other nodes hang. (Thus, computer trees are usually upsidedown.)

SUBTREE--A branch of a tree. Pick any node in a tree--it is the root of a subtree.

TERMINAL NODE--A node in a binary tree that has no left and/or right subtree. In a PATRICIA tree, the affected right or left link then becomes a backward pointer.

THREAD--The same as a backward pointer.

VISIT--A term for what we do when we "look at" a node during a preorder, postorder, or endorder traversal. Usually a visit involves performing an algorithm, or printing out information.

BIBLIOGRAPHY

Fredkin, E. (1960) "Trie Memory." Comm ACM 3,9 (September) 490-99.

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

Knuth, D. E. (1968) The Art of Computer Programming, Vol. 1 Fundamental Algorithms.

Martin, W. A. (1971) "Sorting." Computer Surveys. 3,4 (December) 147-174.

Morrison, D. R. (1968) "PATRICIA - Practical Algorithm to Retrieve Information Coded in Alphanumeric." JACM 15, 4 (October) 514-534.

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

INDEX

•••••SEARCHING FOR ALL KEYS STARTING WITH
FIGURE 2-3 THE ACTUAL REPRESENTATION OF THE PATRICIA TREE OF FIGURE 2-1 PAGE 21
FIGURE 2-2 IF WE ADD A KEY TO THE TREE OF FIGURE 2-1 THE TREE IS CHANGED PAGE 18
5.6 CONCLUSIONS - ADVANTAGES OF THE COMPRESSED FORM PAGE 105
2.1.1 COMMENTS ON ALGORITHM 2.1 PAGE 26
IGURE C-1 TIMING FOR THE LOOP IN ALGORITHM 2.1 PAGE 144
TIMING FOR THE SEARCH LOOP OF ALGORITHM 2.1 PAGE 142
2.2.1 COMMENTS ON ALGORITHM 2.2 PAGE 28
RE TREE OF FIGURE 5-1 AS BUILT BY ALGORITHM 2.2 PAGE 80
FIGURE 5-3 IF WE USE ALGORITHM 2.2 AND INSERT THE KEYS IN REVERSE ORDER PAGE 81
TIMING FOR A TRAVERSAL LOOP USING ALGORITHM 2.3 PAGE 145
FIGURE 5-4 AFTER STEP 2 OF ALGORITHM 5.3.7 PAGE 100
TAG LINK FIELDS DURING STEP 2 OF ALGORITHM 5.3.7 PAGE 99
A.0 APPENDIX - OTHER ALGORITHMS PAGE 114
C.3 TIMING FOR OTHER ALGORITHMS PAGE 143
GURE C-3 TIMING OF MOST PATRICIA ALGORITHMS PAGE 146
A.2 ALGORITHMS FOR A DOUBLY LINKED AVAILABLE LIST PAGE 115
.3.1 SOME CHARACTERISTICS OF THE ALGORITHMS IN THIS DISSERTATION PAGE 9
URE 2-4 A PATRICIA TREE BUILT BY ALGORITHMS 2.2 AND 2.1 PAGE 23
4.4 CONCLUSIONS - ALTERING TEXT PAGE 76
3.0 ALTERING THE PATRICIA TREE - NODE DELETION PAGE 32
5.0 ALTERNATE METHODS FOR REPRESENTING THE PATRICIA TREE PAGE 77
6.2 ANOTHER APPLICATION - CALL FOR ACTION PAGE 109
F.0 APPENDIX - GLOSSARY PAGE 152
A.0 APPENDIX - OTHER ALGORITHMS PAGE 114
H.0 APPENDIX - THE TEST PROGRAM PAGE 119
D.0 APPENDIX - THEOREMS AND PROOFS PAGE 147
C.0 APPENDIX - TIMING CONSIDERATIONS PAGE 142
6.2 ANOTHER APPLICATION - CALL FOR ACTION PAGE 109
6.0 PRACTICAL APPLICATIONS PAGE 106
ALGORITHM: GET A NODE FROM AN AVAILABLE LIST PAGE 114
1 ALGORITHM: RETURN A NODE TO AN AVAILABLE LIST PAGE 114
ALGORITHMS FOR A DOUBLY LINKED AVAILABLE LIST PAGE 115
RETURN A NODE TO A DOUBLY LINKED AVAILABLE LIST PAGE 116
: GET A NODE FROM A DOUBLY LINKED AVAILABLE LIST PAGE 115
IGURE 3-2 TYPE 1 NODE STRUCTURE. MACHINE POINTERS ARE INDICATED PAGE 34
2.4 HOW WELL BALANCED IS A PATRICIA TREE? PAGE 30
BIBLIOGRAPHY PAGE 154
FIGURE 5-4 A BINARY TREE PAGE 87
FIGURE 1-2 A SORTED BINARY TREE PAGE 11
FIGURE 2-5 THE BIT COMPARISONS THAT ARE ACTUALLY MADE PAGE 24
6.1.1 BOOLEAN OPERATIONS PAGE 108
RE 5-2 THE TREE OF FIGURE 5-1 AS BUILT BY ALGORITHM 2.2 PAGE 80
6.2 ANOTHER APPLICATION - CALL FOR ACTION PAGE 109
SOME OF THE OUTPUT PRODUCED FOR CALL FOR ACTION PAGE 111
FIGURE 6-1 SOME TEXT OF THE CALL FOR ACTION FILES PAGE 110
FIGURE 3.11 SPECIAL CASE WHERE THETA PAGE 57
RE TREE OF FIGURE 2-1 THE TREE IS CHANGED PAGE 18
1.3.1 SOME CHARACTERISTICS OF THE ALGORITHMS IN THIS DISSERTATION PAGE 9
QUENTIAL FORM WITH LTAG AND RLINK COMBINED PAGE 104
ORDER TRAVERSAL OF STRUCTURE WITH COMBINED RLINK-LTAG PAGE 101
HM: SEARCH THROUGH STRUCTURE WITH COMBINED RLINK-LTAG PAGE 101

T THREADED TO PREORDER SEQUENTIAL — COMBINING RLINK-LTAG — PAGE 102
2.1.1 — COMMENTS ON ALGORITHM 2.1 — PAGE 26
2.2.1 — COMMENTS ON ALGORITHM 2.2 — PAGE 28
3 SOME USEFUL TRICKS INVOLVING — COMPARISON STRINGS — PAGE 109
FIGURE 2-5 THE HIT — COMPARISONS THAT ARE ACTUALLY MADE — PAGE 24
CONCLUSIONS - ADVANTAGES OF THE — COMPRESSED FORM — PAGE 105
5.5 FURTHER — COMPRESSION — PAGE 103
4.1.1 — CONCEPTS BEHIND DELETING CONTIGUOUS TEXT — PAGE 68
3.4 — CONCLUSION - DELETING NODES — PAGE 61
5.5 — CONCLUSIONS - ADVANTAGES OF THE COMPRESSED FORM — PAGE 105
4.3 — CONCLUSIONS - ALTERING TEXT — PAGE 76
2.5 — CONCLUSIONS - SUMMARY OF CHAPTER TWO — PAGE 31
FIGURE 3B10 GENERAL — CONFIGURATION FOR A SUBTREE DELETION — PAGE 56
5.1.2 AN IMPORTANT AND IMMEDIATE — CONSEQUENCE - ELIMINATING LTAG — PAGE 85
C.0 APPENDIX - TIMING — CONSIDERATIONS — PAGE 142
4.1 DELETING — CONTIGUOUS TEXT — PAGE 63
#4.1.2 ALGORITHM: DELETE — CONTIGUOUS TEXT — PAGE 72
4.1.1 CONCEPTS BEHIND DELETING — CONTIGUOUS TEXT — PAGE 68
5.3.5 ALGORITHM: — CONVERT A RIGHT THREADED TREE TO PREORDER SEQUENTIAL — PAGE 93
4.1.5 — CONVERT TO PREORDER SEQUENTIAL FORM — PAGE 122
#5.3.7 ALGORITHM: — CONVERT TO PREORDER SEQUENTIAL FORM OVER THE SAME SPACE — PAGE 94
5.1.6 — CONVERTING OVER THE SAME MEMORY SPACE — PAGE 93
##5.1.1 ALGORITHM: — CREATE A RIGHT THREADED PATRICIA TREE — PAGE 84
4.1.1 READ TEXT AND — CREATE A TREE — PAGE 119
FIGURE 5-6 PASS 1 — CREATES THIS FROM THE STRUCTURE OF FIGURE 5-1 — PAGE 97
4.1.4 — DELETE A NODE FROM THE TREE — PAGE 121
#3.1 ALGORITHM: — DELETE A PATRICIA NODE FROM THE TREE — PAGE 38
4.1.3 ALGORITHM: — DELETE A SPECIFIC KEY FROM THE TEXT — PAGE 75
#4.1.2 ALGORITHM: — DELETE CONTIGUOUS TEXT — PAGE 72
4.1.5 — DELETE ON INSERT TEXT — PAGE 121
4.1 — DELETING CONTIGUOUS TEXT — PAGE 63
4.1.1 CONCEPTS BEHIND — DELETING CONTIGUOUS TEXT — PAGE 68
3.4 CONCLUSION - — DELETING NODES — PAGE 61
ALTERING THE PATRICIA TREE - NODE — DELETION — PAGE 32
EPARING THE STRUCTURE FOR SUBTREE — DELETION — PAGE 59
NERAL CONFIGURATION FOR A SUBTREE — DELETION — PAGE 56
DELETION CORRESPONDS TO A TYPE 2 — DELETION — PAGE 58
PREPARE THE STRUCTURE FOR SUBTREE — DELETION — PAGE 60
FIGURE 3B12 SUBTREE — DELETION CORRESPONDS TO A TYPE 2 DELETION — PAGE 58
3.3 SUBTREE PRUNING - — DELETION OF PREFIXES — PAGE 54
4.0 — DELETION OF TEXT MATERIAL — PAGE 62
GURE 1-4 THE NUMBERS INSERTED IN — DESCENDING ORDER — PAGE 14
3.2 — DETERMINING TO, FT, TT — PAGE 52
5.4 A SLIGHTLY — DIFFERENT VERSION OF PREORDER SEQUENTIAL REPRESENTATION — PAGE 101
1.4 THE — DIGITAL TREE — PAGE 10
FIGURE 1-3 A — DIGITAL TREE — PAGE 14
I INSERT - SEARCH FOR A NODE IN A — DIGITAL TREE — PAGE 12
4.1.2 — DISPLAY THE TREE AND/OR THE TEXT — PAGE 120
THAT WILL BE USED THROUGHOUT THE — DISSERTATION — PAGE 33
RISTICS OF THE ALGORITHMS IN THIS — DISSERTATION — PAGE 9
A.2 ALGORITHMS FOR A — DOUBLY LINKED AVAILABLE LIST — PAGE 115
.2 ALGORITHM: RETURN A NODE TO A — DOUBLY LINKED AVAILABLE LIST — PAGE 116

2.1 ALGORITHM: GET A NODE FROM A DOUBLY LINKED AVALIABLE LIST PAGE 115
FIGURE 4-3 THE FOURTH A HAS BEEN ELIMINATED FROM THE TEXT PAGE 66
FIGURE 4-2 THE SPACE HAS BEEN ELIMINATED FROM THE TEXT OF FIGURE 4-1 PAGE 65
RTANT AND IMMEDIATE CONSEQUENCE - ELIMINATING LTAG PAGE 85
FIGURE 4-4 PREORDER AND ENDORDER VISITS PAGE 71
2.0 THE ESSENTIALS OF PATRICIA PAGE 15
FIGURE 5-1 AN EXAMPLE OF A RIGHT THREADED PATRICIA TREE PAGE 78
1.3 EXAMPLE OF A STRUCTURE DESIGNED FOR FAST KEY RETRIEVAL PAGE 7
8.1 THE FACILITIES OF THE PROGRAM PAGE 119
AMPLE OF A STRUCTURE DESIGNED FOR FAST KEY RETRIEVAL PAGE 7
SOME TEXT OF THE CALL FOR ACTION FILES PAGE 110
3.2.1 ALGORITHM: FIND PD, TD, PT, TT PAGE 54
6 CONVERT TO PREORDER SEQUENTIAL FORM PAGE 122
NS - ADVANTAGES OF THE COMPRESSED FORM PAGE 105
FIGURE 5&10 PREORDER SEQUENTIAL FORM WITH LTAG AND RLINK COMBINED PAGE 104
3.2 DETERMINING TD, PT, TT PAGE 52
3.2.1 ALGORITHM: FIND PD, TD, PT, TT PAGE 54
5.5 FURTHER COMPRESSION PAGE 103
A.2.1 ALGORITHM: GET A NODE FROM A DOUBLY LINKED AVALIABLE LIST PAGE 115
A.1 ALGORITHM: GET A NODE FROM AN AVAILABLE LIST PAGE 114
E.0 APPENDIX - GLOSSARY PAGE 152
5.1 A HYPOTHETICAL MEDIUM-SCALE SYSTEM PAGE 100
5.1.2 AN IMPORTANT AND IMMEDIATE CONSEQUENCE - ELIMINATING LTAG PAGE 85
5.1.2 AN IMPORTANT AND IMMEDIATE CONSEQUENCE - ELIMINATING LTAG PAGE 85
FIGURE 5-4 THE IMPORTANT LINK FIELDS DURING STEP 2 OF ALGORITHM 5.3.7 PAGE 99
1.4.1 ALGORITHM: INSERT - SEARCH FOR A NODE IN A DIGITAL TREE PAGE 12
#2.2 ALGORITHM: INSERT A NEW NODE INTO A PATRICIA TREE PAGE 27
8.1.5 DELETE OR INSERT TEXT PAGE 121
4.2 ALGORITHM: INSERT TEXT PAGE 75
5-3 IF WE USE ALGORITHM 2.2 AND INSERT THE KEYS IN REVERSE ORDER PAGE 81
1.0 INTRODUCTION PAGE 1
C.2 TIMING FOR A TRAVERSAL ITERATION PAGE 142
ENDORDER SEQUENTIAL STRUCTURE FOR A KEY PAGE 91
8.1.3 SEARCH FOR A KEY AND LIST ALL ITS MATCHES PAGE 121
1.3 ALGORITHM: DELETE A SPECIFIC KEY FROM THE TEXT PAGE 75
1.3.2 ALGORITHM: SEARCH FOR A KEY IN A TRIE MEMORY PAGE 9
OF A STRUCTURE DESIGNED FOR FAST KEY RETRIEVAL PAGE 7
FIGURE 2-2 IF WE ADD A KEY TO THE TREE OF FIGURE 2-1 THE TREE IS CHANGED PAGE 18
1.2 SEARCHING FOR KEYS PAGE 6
USE ALGORITHM 2.2 AND INSERT THE KEYS IN REVERSE ORDER PAGE 81
FIGURE 5-5 THE IMPORTANT LINK FIELDS DURING STEP 2 OF ALGORITHM 5.3.7 PAGE 99
A.2 ALGORITHMS FOR A DOUBLY LINKED AVAILABLE LIST PAGE 115
ORITHM: RETURN A NODE TO A DOUBLY LINKED AVAILABLE LIST PAGE 116
ORITHM: GET A NODE FROM A DOUBLY LINKED AVALIABLE LIST PAGE 115
THM: GET A NODE FROM AN AVAILABLE LIST PAGE 114
HM: RETURN A NODE TO AN AVAILABLE LIST PAGE 114
MS FOR A DOUBLY LINKED AVAILABLE LIST PAGE 115
OF FROM A DOUBLY LINKED AVALIABLE LIST PAGE 115
NODE TO A DOUBLY LINKED AVAILABLE LIST PAGE 116
8.1.3 SEARCH FOR A KEY AND LIST ALL ITS MATCHES PAGE 121
#2.3 ALGORITHM: LIST ALL MATCHES IN A PATRICIA TREE PAGE 29
5-7 THE SPECIAL SITUATION WHERE LLINK(J) = I PAGE 98

FIGURE C-1 TIMING FOR THE LOOP IN ALGORITHM 2.1 PAGE 144
C.1 TIMING FOR THE SEARCH LOOP OF ALGORITHM 2.1 PAGE 142
IGURE C-2 TIMING FOR A TRAVERSAL LOOP USING ALGORITHM 2.3 PAGE 145
OF STRUCTURE WITH COMBINED RLINK- LTAG PAGE 101
EORDER SEQUENTIAL COMBINING RLINK- LTAG PAGE 102
MEDIATE CONSEQUENCE - ELIMINATING LTAG PAGE 85
UCH STRUCTURE WITH COMBINED RLINK- LTAG PAGE 101
810 PREORDER SEQUENTIAL FORM WITH LTAG AND RLINK COMBINED PAGE 104
SEARCH FOR A KEY AND LIST ALL ITS MATCHES PAGE 121
#2.3 ALGORITHM: LIST ALL MATCHES IN A PATRICIA TREE PAGE 29
4.1 DELETION OF TEXT MATERIAL PAGE 62
5.1 A HYPOTHETICAL MEDIUM-SCALE SYSTEM PAGE 106
FIGURE 1-1 A SAMPLE TRIE MEMORY PAGE 8
RITHM: SEARCH FOR A KEY IN A TRIE MEMORY PAGE 9
5.1.6 CONVERTING OVER THE SAME MEMORY SPACE PAGE 93
5.0 ALTERNATE METHODS FOR REPRESENTING THE PATRICIA TREE PAGE 77
5.3.4 HOW TO HANDLE MODIFICATIONS PAGE 92
FIGURE C-3 TIMING OF MOST PATRICIA ALGORITHMS PAGE 146
#2.2 ALGORITHM: INSERT A NEW NODE INTO A PATRICIA TREE PAGE 27
0 ALTERING THE PATRICIA TREE - NODE DELETION PAGE 32
A.2.1 ALGORITHM: GET A NODE FROM A DOUBLY LINKED AVALIABLE LIST PAGE 115
A.1 ALGORITHM: GET A NODE FROM AN AVAILABLE LIST PAGE 114
5.1.4 DELETE A NODE FROM THE TREE PAGE 121
1 ALGORITHM: DELETE A PATRICIA NODE FROM THE TREE PAGE 38
ALGORITHM: INSERT - SEARCH FOR A NODE IN A DIGITAL TREE PAGE 12
#2.1 ALGORITHM: SEARCH FOR A NODE IN A PATRICIA TREE PAGE 25
#2.2 ALGORITHM: INSERT A NEW NODE INTO A PATRICIA TREE PAGE 27
FIGURE 3-3 TYPE 2 NODE STRUCTURE PAGE 35
FIGURE 3-4 TYPE 2A NODE STRUCTURE PAGE 36
FIGURE 3-2 TYPE 1 NODE STRUCTURE. BACKWARD POINTERS ARE INDICATED PAGE 34
A.2.2 ALGORITHM: RETURN A NODE TO A DOUBLY LINKED AVAILABLE LIST PAGE 116
A.1.1 ALGORITHM: RETURN A NODE TO AN AVAILABLE LIST PAGE 114
3.4 CONCLUSION - DELETING NODES PAGE 61
FIGURE 3-1 NOTATION THAT WILL BE USED THROUGHOUT THE DISSERTATION PAGE 33
A.1.1 BOOLEAN OPERATIONS PAGE 108
ME NUMBERS INSERTED IN DESCENDING ORDER PAGE 14
.2 AND INSERT THE KEYS IN REVERSE ORDER PAGE 81
C.3 TIMING FOR OTHER ALGORITHMS PAGE 143
B.2 SAMPLE OUTPUT PAGE 123
FIGURE 6-2 SOME OF THE OUTPUT PRODUCED FOR CALL FOR ACTION PAGE 111
FIGURE 5-6 PASS 1 CREATES THIS FROM THE STRUCTURE OF FIGURE 5-1 PAGE 97
2.0 THE ESSENTIALS OF PATRICIA PAGE 15
5.3 PREORDER SEQUENTIAL PATRICIA PAGE 88
FIGURE C-3 TIMING OF MOST PATRICIA ALGORITHMS PAGE 146
#3.1 ALGORITHM: DELETE A PATRICIA NODE FROM THE TREE PAGE 38
FIGURE 2-1 A SIMPLE PATRICIA TREE PAGE 16
5.1 A RIGHT THREADED PATRICIA TREE PAGE 77
ALGORITHM: LIST ALL MATCHES IN A PATRICIA TREE PAGE 29
ALGORITHM: SEARCH FOR A NODE IN A PATRICIA TREE PAGE 25
1 AN EXAMPLE OF A RIGHT THREADED PATRICIA TREE PAGE 78
GORITHM: INSERT A NEW NODE INTO A PATRICIA TREE PAGE 27
LGORITHM: CREATE A RIGHT THREADED PATRICIA TREE PAGE 84

NATE METHODS FOR REPRESENTING THE PATRICIA TREE PAGE 77
3.0 ALTERING THE PATRICIA TREE - NODE DELETION PAGE 32
FIGURE 2-4 A PATRICIA TREE BUILT BY ALGORITHMS 2.2 AND 2.1 PAGE 23
THE ACTUAL REPRESENTATION OF THE PATRICIA TREE OF FIGURE 2-1 PAGE 21
2.4 HOW WELL BALANCED IS A PATRICIA TREE? PAGE 30
3.2.1 ALGORITHM: FIND PU. TU. FT. TT PAGE 54
TYPE 1 NODE STRUCTURE. BACKWARD POINTERS ARE INDICATED PAGE 34
PREORDER SEQUENTIAL STRUCTURE IN POSTORDER PAGE 88
5.4.2 ALGORITHM: POSTORDER TRAVERSAL OF STRUCTURE WITH COMBINED RLINK-LTAG PAGE 101
6.0 PRACTICAL APPLICATIONS PAGE 106
SUBTREE PRUNING - DELETION OF PREFIXES PAGE 54
FIGURE 4-4 PREORDER AND ENDORDER VISITS PAGE 71
CONVERT A RIGHT THREADED TREE TO PREORDER SEQUENTIAL PAGE 93
ITHM: TRANSFORM RIGHT THREADED TO PREORDER SEQUENTIAL COMBINING RLINK-LTAG PAGE 102
5.1.6 CONVERT TO PREORDER SEQUENTIAL FORM PAGE 122
#5.3.7 ALGORITHM: CONVERT TO PREORDER SEQUENTIAL FORM OVER THE SAME SPACE PAGE 94
FIGURE 5610 PREORDER SEQUENTIAL FORM WITH LTAG AND RLINK COMBINED PAGE 104
5.3 PREORDER SEQUENTIAL PATRICIA PAGE 88
5.2 PREORDER SEQUENTIAL REPRESENTATION PAGE 86
FIGURE 5-5 THE PREORDER SEQUENTIAL REPRESENTATION PAGE 89
A SLIGHTLY DIFFERENT VERSION OF PREORDER SEQUENTIAL REPRESENTATION PAGE 101
5.3.3 ALGORITHM: SEARCH THE PREORDER SEQUENTIAL STRUCTURE FOR A KEY PAGE 91
#5.3.1 ALGORITHM: TRAVERSE A PREORDER SEQUENTIAL STRUCTURE IN POSTORDER PAGE 88
3.3.2 ALGORITHM: PREPARE THE STRUCTURE FOR SUBTREE DELETION PAGE 60
3.3.1 PREPARING THE STRUCTURE FOR SUBTREE DELETION PAGE 59
A.3 ALGORITHM: PRINT A TREE PAGE 116
H.0 APPENDIX - THE TEST PROGRAM PAGE 119
H.1 THE FACILITIES OF THE PROGRAM PAGE 119
D.0 APPENDIX - THEOREMS AND PROOFS PAGE 147
3.3 SUBTREE PRUNING - DELETION OF PREFIXES PAGE 54
H.1.1 READ TEXT AND CREATE A TREE PAGE 119
5.2 PREORDER SEQUENTIAL REPRESENTATION PAGE 86
GURE 5-5 THE PREORDER SEQUENTIAL REPRESENTATION PAGE 89
NT VERSION OF PREORDER SEQUENTIAL REPRESENTATION PAGE 101
FIGURE 2-3 THE ACTUAL REPRESENTATION OF THE PATRICIA TREE OF FIGURE 2-1 PAGE 21
5.0 ALTERNATE METHODS FOR REPRESENTING THE PATRICIA TREE PAGE 77
A STRUCTURE DESIGNED FOR FAST KEY RETRIEVAL PAGE 7
A.2.2 ALGORITHM: RETURN A NODE TO A DOUBLY LINKED AVAILABLE LIST PAGE 116
A.1.1 ALGORITHM: RETURN A NODE TO AN AVAILABLE LIST PAGE 114
ORITHM 2.2 AND INSERT THE KEYS IN REVERSE ORDER PAGE 81
5.1 A RIGHT THREADED PATRICIA TREE PAGE 77
FIGURE 5-1 AN EXAMPLE OF A RIGHT THREADED PATRICIA TREE PAGE 78
#5.1.1 ALGORITHM: CREATE A RIGHT THREADED PATRICIA TREE PAGE 84
5.4.3 ALGORITHM: TRANSFORM RIGHT THREADED TO PREORDER SEQUENTIAL COMBINING RLINK-LTAG PAGE 102
5.1.5 ALGORITHM: CONVERT A RIGHT THREADED TREE TO PREORDER SEQUENTIAL PAGE 93
DER SEQUENTIAL FORM WITH LTAG AND RLINK COMBINED PAGE 104
VERSAL OF STRUCTURE WITH COMBINED RLINK-LTAG PAGE 101
TO PREORDER SEQUENTIAL COMBINING RLINK-LTAG PAGE 102
H THROUGH STRUCTURE WITH COMBINED RLINK-LTAG PAGE 101
5.3.6 CONVERTING OVER THE SAME MEMORY SPACE PAGE 93
H.2 SAMPLE OUTPUT PAGE 123
FIGURE 1-1 A SAMPLE TRIE MEMORY PAGE 8

R.1.3	SEARCH FOR A KEY AND LIST ALL ITS MATCHES	PAGE 121
1.3.2 ALGORITHM:	SEARCH FOR A KEY IN A TRIE MEMORY	PAGE 9
1.4.1 ALGORITHM: INSERT -	SEARCH FOR A NODE IN A DIGITAL TREE	PAGE 12
**2.1 ALGORITHM:	SEARCH FOR A NODE IN A PATRICIA TREE	PAGE 25
C.1 TIMING FOR THE	SEARCH LOOP OF ALGORITHM 2.1	PAGE 142
5.3.3 ALGORITHM:	SEARCH THE PREORDER SEQUENTIAL STRUCTURE FOR A KEY	PAGE 91
5.4.1 ALGORITHM:	SEARCH THROUGH STRUCTURE WITH COMBINED RLINK-LTAG	PAGE 101
1.2	SEARCHING FOR KEYS	PAGE 6
A RIGHT THREADED TREE TO PREORDER	SEQUENTIAL	PAGE 93
NSFORM RIGHT THREADED TO PREORDER	SEQUENTIAL COMBINING RLINK-LTAG	PAGE 102
5.1.4 CONVERT TO PREORDER	SEQUENTIAL FORM	PAGE 122
7 ALGORITHM: CONVERT TO PREORDER	SEQUENTIAL FORM OVER THE SAME SPACE	PAGE 94
FIGURE 5-10 PREORDER	SEQUENTIAL FORM WITH LTAG AND RLINK COMBINED	PAGE 104
5.3 PREORDER	SEQUENTIAL PATRICIA	PAGE 88
5.2 PREORDER	SEQUENTIAL REPRESENTATION	PAGE 86
FIGURE 5-5 THE PREORDER	SEQUENTIAL REPRESENTATION	PAGE 89
TLY DIFFERENT VERSION OF PREORDER	SEQUENTIAL REPRESENTATION	PAGE 101
3 ALGORITHM: SEARCH THE PREORDER	SEQUENTIAL STRUCTURE FOR A KEY	PAGE 91
1 ALGORITHM: TRAVERSE A PREORDER	SEQUENTIAL STRUCTURE IN POSTORDER	PAGE 88
FIGURE 2-1 A	SIMPLE PATRICIA TREE	PAGE 16
FIGURE 5-7 THE SPECIAL	SITUATION WHERE LLINK(J) = I	PAGE 98
5.4 A	SLIGHTLY DIFFERENT VERSION OF PREORDER SEQUENTIAL REPRESENTATION	PAGE 101
FIGURE 1-2 A	SORTED BINARY TREE	PAGE 11
CONVERTING OVER THE SAME MEMORY	SPACE	PAGE 93
FIGURE 4-2 THE	SPACE HAS BEEN ELIMINATED FROM THE TEXT OF FIGURE 4-1	PAGE 65
DER SEQUENTIAL FORM OVER THE SAME	SPACE	PAGE 94
FIGURE 3.11	SPECIAL CASE WHERE TEM=TA	PAGE 57
FIGURE 5.7 THE	SPECIAL SITUATION WHERE LLINK(J) = I	PAGE 98
4.1.3 ALGORITHM: DELETE A	SPECIFIC KEY FROM THE TEXT	PAGE 75
FIGURE 4-1 A	STRANGE TREE	PAGE 64
SEFUL TRICKS INVOLVING COMPARISON	STRINGS	PAGE 109
FIGURE 3-3 TYPE 2 NODE	STRUCTURE	PAGE 35
FIGURE 3-4 TYPE 2A NODE	STRUCTURE	PAGE 36
5.3.2 CAN THE	STRUCTURE BE UTILIZED?	PAGE 91
1.3 EXAMPLE OF A	STRUCTURE DESIGNED FOR FAST KEY RETRIEVAL	PAGE 7
M: SEARCH THE PREORDER SEQUENTIAL	STRUCTURE FOR A KEY	PAGE 91
3.3.1 PREPARING THE	STRUCTURE FOR SUBTREE DELETION	PAGE 59
3.3.2 ALGORITHM: PREPARE THE	STRUCTURE FOR SUBTREE DELETION	PAGE 60
M: TRAVERSE A PREORDER SEQUENTIAL	STRUCTURE IN POSTORDER	PAGE 88
5-6 PASS 1 CREATES THIS FROM THE	STRUCTURE OF FIGURE 5-1	PAGE 97
ALGORITHM: POSTORDER TRAVERSAL OF	STRUCTURE WITH COMBINED RLINK-LTAG	PAGE 101
5.4.1 ALGORITHM: SEARCH THROUGH	STRUCTURE WITH COMBINED RLINK-LTAG	PAGE 101
FIGURE 3-2 TYPE 1 NODE	STRUCTURE, BACKWARD POINTERS ARE INDICATED	PAGE 34
.3.1 PREPARING THE STRUCTURE FOR	SUBTREE DELETION	PAGE 59
3.10 GENERAL CONFIGURATION FOR A	SUBTREE DELETION	PAGE 56
ORITHM: PREPARE THE STRUCTURE FOR	SUBTREE DELETION	PAGE 60
FIGURE 3.12	SUBTREE DELETION CORRESPONDS TO A TYPE 2 DELETION	PAGE 58
3.3	SUBTREE PRUNING - DELETION OF PREFIXES	PAGE 54
2.5 CONCLUSIONS -	SUMMARY OF CHAPTER TWO	PAGE 31
.1 A HYPOTHETICAL MEDIUM-SCALE	SYSTEM	PAGE 106
3.2 DETERMINING	TO. FT. TT	PAGE 52
3.2.1 ALGORITHM: FIND PO.	TO. FT. TT	PAGE 54

1.1	TERMINOLOGY	PAGE 4
8.0 APPENDIX - THE	TEST PROGRAM	PAGE 119
8.1.5 DELETE OR INSERT	TEXT	PAGE 121
4.2 ALGORITHM: INSERT	TEXT	PAGE 75
4.1 DELETING CONTIGUOUS	TEXT	PAGE 63
4.3 CONCLUSIONS - ALTERING	TEXT	PAGE 76
.1.2 DISPLAY THE TREE AND/OR THE	TEXT	PAGE 120
1.2 ALGORITHM: DELETE CONTIGUOUS	TEXT	PAGE 72
NCEPTS BEHIND DELETING CONTIGUOUS	TEXT	PAGE 68
M: DELETE A SPECIFIC KEY FROM THE	TEXT	PAGE 75
TH A HAS BEEN ELIMINATED FROM THE	TEXT	PAGE 66
8.1.1 READ	TEXT AND CREATE A TREE	PAGE 119
4.0 DELETION OF	TEXT MATERIAL	PAGE 62
PACE HAS BEEN ELIMINATED FROM THE	TEXT OF FIGURE 4-1	PAGE 65
FIGURE 6-1 SOME	TEXT OF THE CALL FOR ACTION FILES	PAGE 110
D.0 APPENDIX -	THEOREMS AND PROOFS	PAGE 147
5.1 A RIGHT	THREADED PATRICIA TREE	PAGE 77
FIGURE 5-1 AN EXAMPLE OF A RIGHT	THREADED PATRICIA TREE	PAGE 78
#5.1.1 ALGORITHM: CREATE A RIGHT	THREADED PATRICIA TREE	PAGE 84
5.4.3 ALGORITHM: TRANSFORM RIGHT	THREADED TO PREORDER SEQUENTIAL COMBINING RLINK-LTAG	PAGE 102
5.3.5 ALGORITHM: CONVERT A RIGHT	THREADED TREE TO PREORDER SEQUENTIAL	PAGE 93
C.0 APPENDIX -	TIMING CONSIDERATIONS	PAGE 142
C.2	TIMING FOR A TRAVERSAL ITERATION	PAGE 142
FIGURE C-2	TIMING FOR A TRAVERSAL LOOP USING ALGORITHM 2.3	PAGE 145
C.3	TIMING FOR OTHER ALGORITHMS	PAGE 143
FIGURE C-1	TIMING FOR THE LOOP IN ALGORITHM 2.1	PAGE 144
C.1	TIMING FOR THE SEARCH LOOP OF ALGORITHM 2.1	PAGE 142
FIGURE C-3	TIMING OF MOST PATRICIA ALGORITHMS	PAGE 146
5.4.3 ALGORITHM:	TRANSFORM RIGHT THREADED TO PREORDER SEQUENTIAL COMBINING RLINK-LTAG	PAGE 102
C.2 TIMING FOR A	TRAVERSAL ITERATION	PAGE 142
FIGURE C-2 TIMING FOR A	TRAVERSAL LOOP USING ALGORITHM 2.3	PAGE 145
5.4.2 ALGORITHM: POSTORDER	TRAVERSAL OF STRUCTURE WITH COMBINED RLINK-LTAG	PAGE 101
#5.3.1 ALGORITHM:	TRAVERSE A PREORDER SEQUENTIAL STRUCTURE IN POSTORDER	PAGE 88
1.4 THE DIGITAL	TREE	PAGE 10
FIGURE 5-4 A BINARY	TREE	PAGE 87
FIGURE 1-3 A DIGITAL	TREE	PAGE 14
FIGURE 4-1 A SPARSE	TREE	PAGE 64
A.3 ALGORITHM: PRINT A	TREE	PAGE 116
FIGURE 1-2 A SORTED BINARY	TREE	PAGE 11
FIGURE 2-1 A SIMPLE PATRICIA	TREE	PAGE 16
8.1.4 DELETE A NODE FROM THE	TREE	PAGE 121
8.1.1 READ TEXT AND CREATE A	TREE	PAGE 119
5.1 A RIGHT THREADED PATRICIA	TREE	PAGE 77
M: LIST ALL MATCHES IN A PATRICIA	TREE	PAGE 29
: DELETE A PATRICIA NODE FROM THE	TREE	PAGE 38
: SEARCH FOR A NODE IN A PATRICIA	TREE	PAGE 25
MPLE OF A RIGHT THREADED PATRICIA	TREE	PAGE 78
INSERT A NEW NODE INTO A PATRICIA	TREE	PAGE 27
CREATE A RIGHT THREADED PATRICIA	TREE	PAGE 84
DDS FOR REPRESENTING THE PATRICIA	TREE	PAGE 77
- SEARCH FOR A NODE IN A DIGITAL	TREE	PAGE 12
3.0 ALTERING THE PATRICIA	TREE - NODE DELETION	PAGE 32

8.1.2 DISPLAY THE TREE AND/OR THE TEXT PAGE 120
FIGURE 2-4 A PATRICIA TREE BUILT BY ALGORITHMS 2.2 AND 2.1 PAGE 23
AL REPRESENTATION OF THE PATRICIA TREE OF FIGURE 2-1 PAGE 21
IGURE 2-2 IF WE ADD A KEY TO THE TREE OF FIGURE 2-1 THE TREE IS CHANGED PAGE 18
FIGURE 5-2 THE TREE OF FIGURE 5-1 AS BUILT BY ALGORITHM 2.2 PAGE 80
GORITHM: CONVERT A RIGHT THREADED TREE TO PREORDER SEQUENTIAL PAGE 93
HOW WELL BALANCED IS A PATRICIA TREE? PAGE 30
6.3 SOME USEFUL TRICKS INVOLVING COMPARISON STRINGS PAGE 109
FIGURE 1-1 A SAMPLE TRIE MEMORY PAGE 8
ALGORITHM: SEARCH FOR A KEY IN A TRIE MEMORY PAGE 9
3.2 DETERMINING TD, TI, TT PAGE 52
.2.1 ALGORITHM: FIND PD, TD, TI, TT PAGE 54
FIGURE 3-2 TYPE 1 NODE STRUCTURE, BACKWARD POINTERS ARE INDICATED PAGE 34
SUBTREE DELETION CORRESPONDS TO A TYPE 2 DELETION PAGE 58
FIGURE 3-3 TYPE 2 NODE STRUCTURE PAGE 35
FIGURE 3-4 TYPE 2A NODE STRUCTURE PAGE 36
FIGURE 3-5 TYPE 2B PAGE 39
FIGURE 3-6 TYPE 2C PAGE 40
FIGURE 3-7 TYPE 2D PAGE 41
FIGURE 3-8 TYPE 2E PAGE 42
FIGURE 3-9 TYPE 2F PAGE 43
6.3 SOME USEFUL TRICKS INVOLVING COMPARISON STRINGS PAGE 109
5.3.2 CAN THE STRUCTURE BE UTILIZED? PAGE 91
FIGURE 4-4 PREORDER AND ENDORDER VISITS PAGE 71