THE EFFECT OF A SPLIT INTERVAL ON SIMPLE PREFIX B^+-TREES

By

TIMOTHY L. TOWNS

Bachelor of Science

Harding University

Searcy, Arkansas

1979

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fullfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
July, 1983

THE EFFECT OF A SPLIT INTERVAL ON

SIMPLE PREFIX B^+-TREES

Thesis Approved:

James R. Van Doren
Thesis Adviser

Michael J. Folk

Sharilyn A. Thoreson

Norman N. Durham
Dean of the Graduate College

1161125

PREFACE

This study examines the effect a split interval has on a simple prefix B^+-tree. A simple prefix B^+-tree is a cousin of the well-known B - tree indexing organization and a split interval is a proposed method to improve the performance of this organization. The purpose of this paper is to determine the usefulness of a split interval by empirically testing its effect on an experimental implementation of a simple prefix B^+-tree.

I would like to express my gratitude to my major adviser, Dr. James R. Van Doren for his guidance and instruction in this study. I would also like to thank Dr. Sharilyn Thoreson and Dr. Mike Folk for serving on my graduate committee.

I would also like to thank John Kerns, Robert Schneider, and Mike Bates for their assistance and motivation in completing this paper.

Finally, I would like to express my appreciation for the support, money, and patience my parents so gladly offerred during my studies.

TABLE OF CONTENTS

Chapter		Page
I.	INTRODUCTION	1
II.	AN OVERVIEW OF B^+-TREES	3
	B^+-Index	3
	B^+-File	5
	Properties of a B^+-Tree	6
	Searching and Updating a B^+-Tree	7
	Summary	9
III.	SIMPLE PREFIX B^+-TREES	10
	Insertion Into a Simple Prefix B^+-Tree .	12
	Effect of Separators on a B^+-Index . . .	16
	Split Interval	17
	A Difference of Opinion	20
	Insertion Utilizing a Split Interval . .	22
	Deletion Utilizing a Split Interval . . .	23
	Prefix B^+-Trees	24
	Summary	28
IV.	EMPIRICAL MEASUREMENT CONCERNING THE EFFECT A SPLIT INTERVAL HAS ON SIMPLE PREFIX B^+-TREES .	30
	Test Cases	31
	Implementation of a Simple Prefix B^+-Tree	32
	Node Organization	33
	The Structure of a Node	35
	Page Replacement Method	36
	Implementation of a Split Interval .	38
	"Words" File and Random Sampling . .	39
	Statistics Module	41
	Results and Analysis of Empirical Testing	42
	Effect on Separator Length	43
	Effect on Storage Utilization . . .	45
	Effect on Tree Height	47
	Effect on Storage Requirements . . .	48
	"Student's t-Test	49
V.	SUMMARY, CONCLUSIONS, AND SUGGESTED FUTURE RESEARCH	53
	Summary	53

Chapter Page

Conclusions 54
Suggested Future Research 55

SELECTED BIBLIOGRAPHY 57

APPENDIX 59

LIST OF TABLES

Table Page

I. Average Length of Separators at Level Two (SIB Constant at One) 43

II. Average Length of Separators at Level One (SIL Constant at One) 45

III. Storage Utilization at the Leaf Level (SIB Held Constant at One) 46

IV. Increased Branching Degree at Level One Due to Increased SIB and SIL 48

V. Total Storage Requirement for B^+-File, B^+-Index, and Entire Tree 50

VI. 95% Confidence Interval for the Difference in Storage Requirement Means 52

LIST OF FIGURES

Figure		Page
1.	Separation of B^+-Tree Components	4
2.	A "Full" B^+-File Leaf Node	11
3.	Split of a B^+-File Leaf Node After Insertion of the Key "Cards"	11
4.	A Simple Prefix B^+-Tree	13
5.	A "Full" B^+-Index Node	15
6.	Index Nodes Resulting From a B-Tree Type Split of the B^+-Index Node in Figure 5 . .	15
7.	Incorrect Split of the B^+-Index Node in Figure 5	16
8.	A B^+-File Node and the Separators Derived From Each Gap	19
9.	A Partial Subtree of a Simple Prefix B^+-Tree	25
10.	A Prefix B^+-Tree Derived From the Simple B^+-Tree in Figure 4	27
11.	Node Organization	34
12.	The Page Buffer	35

CHAPTER I

INTRODUCTION

A B-Tree is a well-known method for indexing a large collection of data. This indexing method is important because B-trees are dynamic and provide logarithmic search and update times. One popular variant of a B-tree is a B^+-tree. In a B^+-tree, all records indexed by the tree reside in the leaf level nodes of the tree. A simple prefix B^+-tree is an extension on the idea of a B^+-tree where the nonleaf nodes or index part of a B^+-tree is replaced by a smaller but equivalent index made up of separators. Separators are strings derived from actual keys occurring in the leaf level of a B^+-tree. The motivation for a simple prefix B^+-tree is the reduction in size and height of the index part of the tree gained by the introduction of separators to replace actual keys. A reduction in the height of the index part of the tree is important because the height of the index has a direct influence on the performance of the index. A reduction in the size of the index means that fewer nodes are required to make up the index and thus the entire tree.

This paper examines a simple prefix B^+-tree with the added feature of a split interval. A split interval allows

a node to split in places other than the middle of the node. This is done to promote shorter separator strings into the index part of the tree and thereby further reduce the height and size of the index.

The primary intent of this paper is to study empirically what effect a split interval has on the performance of a simple prefix B^+-tree and what improvements or consequences arise from its use. The reader should note that this paper was written with the assumption that the reader is familiar with B-trees and the nomenclature associated with B-trees.

Chapter II of this paper contains a brief outline of B^+-trees with an emphasis on the differences between B^+-trees and B-trees. This chapter may be skipped by the reader familiar with B^+-trees.

Chapter III introduces a simple prefix B^+-tree and the notion of a split interval. Descriptions of the algorithms for insertion and deletion into a simple prefix B^+-tree with a split interval are given. Also the predicted effects of a split interval on a simple prefix B^+-tree are given.

Chapter IV contains the results of a study to empirically determine the effect a split interval has on a simple prefix B^+-tree. Descriptions of test cases, the experimental implementation of simple prefix B^+-tree, and the results derived from the test cases are given.

The final chapter contains a summary of the work done and conclusions concerning the use of a split interval.

CHAPTER II

AN OVERVIEW OF B^+-TREES

An important variant of a standard B-tree is a B^+-tree suggested by Knuth (9, section 6.2.4) and described by Comer (4). In a B^+-tree, the tree structure is separated into two distinct parts, a B^+-index and a B^+-file. This separation is possible because all records in the tree have been moved to the leaf level nodes. A B^+-index consists of the upper level or nonleaf nodes and is used to direct searches to the leaf level nodes where records or pointers to records reside. A B^+-file is an ordered set of leaf nodes which contain the records indexed by the B^+-index. Figure 1 illustrates the separation of a B^+-index and a B^+-file.

B^+-Index

An index or upper level node contains many elements known as entries. In a conventional B-tree, an entry is an ordered pair (k,r) where k is a key and r is a record or associated information. Entries in the nodes of a B^+-index contain no records because all records have been moved to the B^+-file. Keys occurring in a B^+-index entry are copies of actual keys having been inserted at the leaf level of the tree. These key copies that comprise the B^+-index are

propagated up into the B^+-index when nodes at the leaf level split.

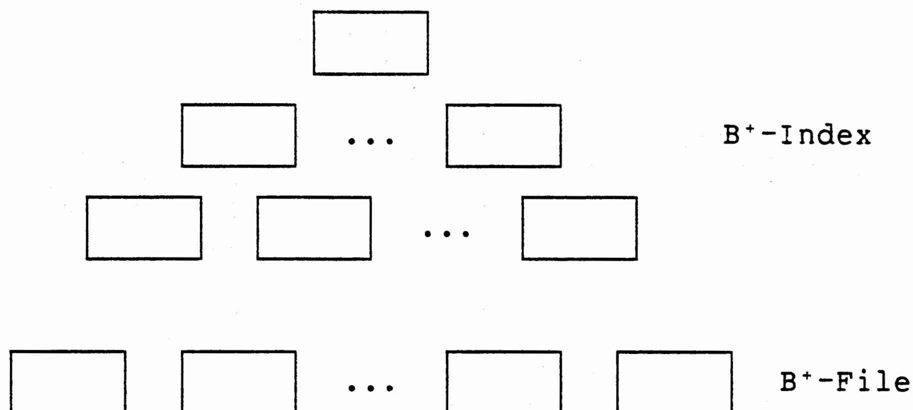

Figure 1. Separation of B^+-Tree Components

Because entries in the nodes of a B^+-index contain no records, an entry in a B^+-index is shorter than an entry of a conventional B-tree index node. A shorter entry means more entries may be packed into a node, thereby increasing the branching degree or order of the node. This increased branching degree of index nodes means the height of the index may be reduced. A reduction of tree height is important because a search of a tree must proceed in a path from the root node to a leaf node. Each node in the traversal path must be referenced or visited to determine the next node in the traversal path. When the nodes of the tree reside on external storage, a node visit means an

expensive disk access will take place. It has been shown that when indexing files where the file and the index reside on external storage, the primary performance bottleneck is the number of accesses to external storage (6). This is due to the seek time and rotational delay typically associated with external storage devices like a disk drive. Therefore, the performance of the indexing method will improve as the tree height is reduced. However, if a tree contains very large nodes, the time required to transfer the node may become a performance bottleneck.

B^+-File

A B^+-file is a logically ordered set of leaf nodes. The order of the leaf nodes is maintained by the index part of the tree. An entry in a leaf node of a B^+-file is an ordered pair, (k,r) where k is a key and r is a record or a pointer to the location of the record associated with k. The entries of a leaf node are in ascending order by key.

Some implementations of a B^+-tree have the leaf nodes linked together from left-to-right. In this case, each leaf node has a rightmost pointer that serves as the link from that node to the next leaf node in collating sequence order. These horizontal links may be traversed beginning at the leftmost leaf node and continuing to the rightmost leaf node to facilitate sequential processing of all records at the leaf level. The rightmost pointer of the rightmost leaf node contains a "null" pointer that signals the end of the

linked list of leaf nodes. In a standard B-tree, all nodes of the tree must be visited, possibly using a preorder traversal, to process the file sequentially. A preorder traversal of a B-tree would require additional main memory requirements for a stack. A stack is needed to store the nodes in the traversal path so they only have to be read once.

An additional property of this linked list of leaf nodes is that finding the next or successor record of a record already found is easy. At most, one additional access to external storage is required to fulfill this query. In a conventional B-tree, finding the successor of a record may mean a traversal of one or more nodes.

Properties of a B^+-Tree

A B^+-tree is a balanced multiway search tree that retains the advantages of B-trees (1). These advantages may be enumerated as follows:

1. Storage utilization is at least 50 percent at any time and should be considerably better on the average.
2. The tree is a dynamic structure so storage is requested and released as the file grows and contracts.
3. The tree structure provides for both random and sequential processing.
4. Logarithmic search and update times are guaranteed.

5. A dynamic B^+-tree requires no periodic reorganization.

Searching and Updating a B^+-Tree

A search of a B^+-tree begins at the root and proceeds down through the levels of the tree until a leaf node is reached. At each level of the search path an index node is referenced to find the pointer to the next node in the search path. If a key in an index node matches the search key, then the nearest pointer to the right is followed to continue the search. In a standard B-tree, a match of a key in the index part with the search key would cause the search algorithm to halt.

All insertions into a B^+-tree are done at the leaf level. Therefore, to insert into a B^+-tree, the index must first be searched to find the proper leaf node for the insertion. If the leaf node has room in it for the insertion, the key and record pair are inserted into the leaf node and the insertion operation is complete. If the leaf node is full, then the leaf node must be split becoming two and a copy of the middle key of this leaf node is passed up to the parent node for insertion. If this parent node is also full, then the index node must split passing up its middle key to its parent node. This process may propagate all the way to the root node where a split of the root node causes the tree to increase in height by one. Thus, as leaf

nodes split, copies of keys existing at the leaf level are propagated up into the index to form a B^+-index.

An alternative to node splitting during an insertion operation is an overflow (1). An overflow is performed by moving entries from the node that is full to a sibling node to avoid the split or to balance storage utilization. When an overflow is performed, the key used to separate the two nodes participating in the overflow must be replaced by a new key that serves to separate the new configuration of the two nodes.

Deletions from a B^+-tree are always done at the leaf level. Therefore, a deletion operation involves searching the tree to locate the proper entry in a leaf node and removing it from the leaf node. If after the deletion the leaf node is at least half full, then the deletion operation is complete. Otherwise, a merge with a subling node is required. A merge is performed by moving entries from the node where the deletion occurred to a sibling node that has sufficient space for the entries. Also, the key in the index part of the tree that served to separate the two merged or concatenated nodes must be removed. This deletion of an index key will always be a deletion from a leaf node of a B^+-index which is structured like a B-tree. Other keys in a B^+-index are unaffected by such a deletion. Deletion of entries in a standard B-tree require the location of a predecessor or successor entry if the deletion occurs in a nonleaf node. If the deletion of an entry from an index

node causes that index node to be less than half full, a merge of that index node with a sibling is required. This means an index entry in the parent node of the index node that is less than half full must be deleted. This merging process may propagate to the root, possibly causing the tree to decrease in height by one level.

A possible alternative to merging nodes during a deletion operation is to perform an underflow (1). An underflow is performed by moving entries from a sibling node into the node where the deletion occurred. This will return the storage utilization of the node where the deletion occurred to 50 percent or more.

Summary

In this chapter, B^+-trees have been shown to be a superior variant of a conventional B-tree. B^+-trees retain the significant advantages and properties of a B-tree, and because all insertions and deletions occur at the leaf level, the algorithms to perform these operations are simpler than their B-tree counterparts. A more thorough treatment of B^+-trees is presented by Webster (16).

CHAPTER III

SIMPLE PREFIX B^+-TREES

In 1977, Bayer and Unterauer (2) introduced two modifications to a B^+-tree known as simple prefix B-trees and prefix B-trees. Their modifications are possible because they recognized the separation of a B^+-index and a B^+-file. In their paper, a B^+-index is referred to as a B*-index and a B^+-file is referred to as a B*-file. However, in this paper B^+-index and B^+-file will be used. This is in accordance with the nomenclature used by Comer (4).

Bayer and Unterauer made the important observation that the keys in a B^+-index are used only to direct the search algorithm to the proper leaf node in a B^+-file. Therefore, they proposed replacing a B^+-index made up of copies of actual keys from a B^+-file with an equivalent B^+-index made up of shorter strings derived from actual keys in a B^+-file. As an example of the derivation of shorter strings to comprise a B^+-index, suppose the leaf node in Figure 2 is full and we wish to insert the key "Cards". This insertion will require the node to split into two nodes as shown in Figure 3.

In a B^+-tree this split would cause the key "Mets" to be propagated up to its parent node for insertion. However, Bayer and Unterauer realized that a shorter string derived from the key "Mets" could be used to separate the two leaf nodes. Therefore, the letter "M" could be used to replace the full key "Mets" in the index.

.Cubs.Expos.Mets.Phillies.Pirates.

Figure 2. A "Full" B^+-File Leaf Node

.Cards.Cubs.Expos.	.Mets.Phillies.Pirates.

Figure 3. Split of a B^+-File Leaf Node After Insertion of the Key "Cards"

They call such a string a "separator". In general, for the example above, any string s, such that

Expos < s <= Mets

could be used in the index to separate the two nodes. Note that the actual key "Mets" qualifies as a separator. However, it is more appropriate to select the shortest

separator possible because shorter separators reduce the height and size of the B^+-index.

A simple prefix B^+-tree is therefore defined as a B^+-tree in which the B^+-index is replaced by a B-tree of variable length separators. Figure 4 is an example of a simple prefix B^+-tree where it is assumed that a node of the tree may contain a maximum of two keys or separators.

The determination of separators to replace actual keys in the B^+-index relies heavily on the following property that will be assumed for the remainder of this paper.

> Assuming that the keys are words over some alphabet and the ordering of the keys is the alphabetic order, then the "prefix property" given below holds (2, p.12):
>
> Let x and y be any two keys such that $x < y$. Then there is a unique prefix y' of y such that (a) y' is a separator between x and y, and (b) no other separator between x and y is shorter than y'.

The technique of determining separators is a form of rear compression of keys. This technique is similar to the techniques described in Chang (3) and Wagner (15), but does not require the computing overhead inherent in their schemes.

Insertion Into a Simple Prefix B^+-Tree

To insert into a simple prefix B^+-Tree, a path is traversed from the root node to the appropriate leaf node. If the leaf node has sufficient room for the insertion, it

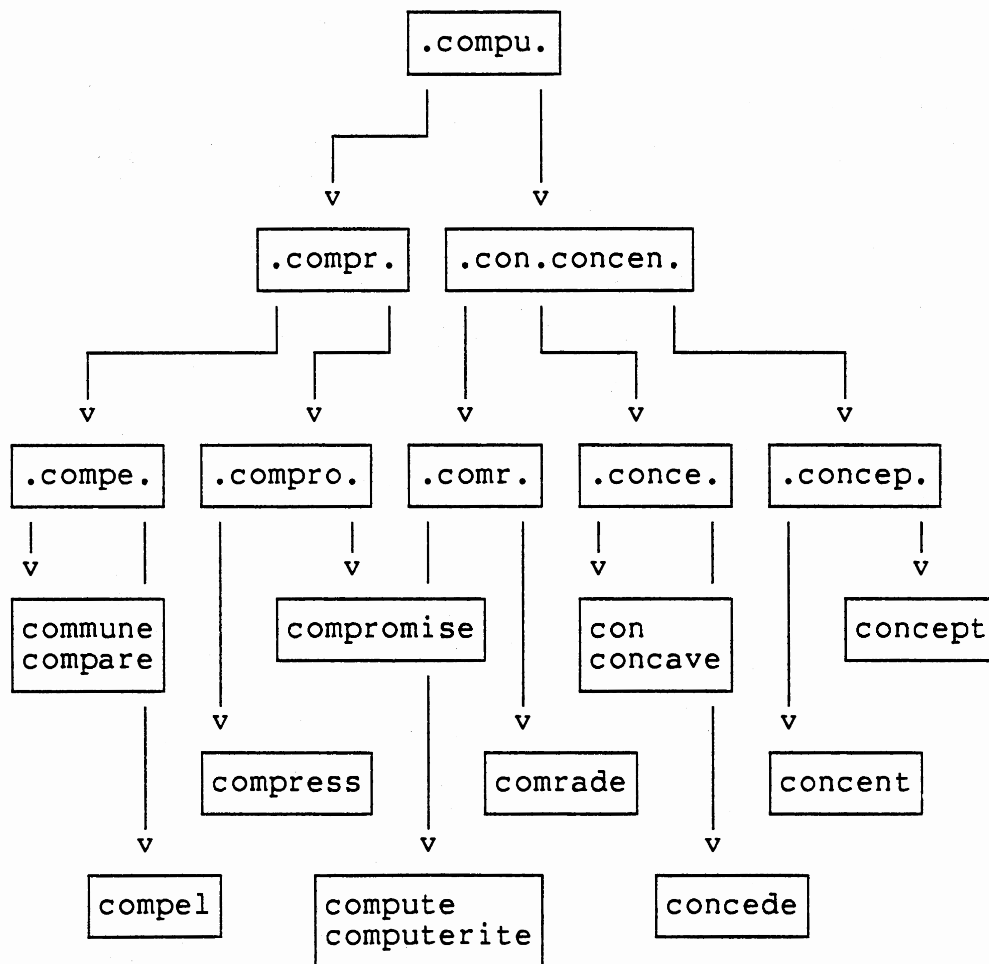

Figure 4. A Simple Prefix B^+-Tree

is done and the insertion is complete. However, if the leaf node is full, then the leaf node must split and a separator must be constructed and passed up to its parent node for insertion. If an index or branch node splits, the procedure of calculating a separator to be passed up is no longer useful. One of the separators in the index node must be passed up to its parent. The following example should illustrate why this is true.

Assume the index node in Figure 5 has had the separator "abdc" inserted into it and the index node must split but has not yet done so. A search of this node with the key "aca" would indicate that the next pointer to be followed in the search path is "4". When the node splits using the conventional B-tree method, the nodes shown in Figure 6 result. A search of this subtree of the index for the key "aca" results in pointer "4" being selected again as the next pointer to be followed in the search path. However, if a separator is derived from the separator "acda" during the index node split, the nodes shown in Figure 7 result. When this subtree is searched for the key "aca" pointer "5" is erroneously selected instead of pointer "4". Therefore, when an index node splits the derivation of separators to be passed up to the parent node is not useful because the search capability of the index is destroyed. When an index node is split, one of the separators in that node must be propagated up.

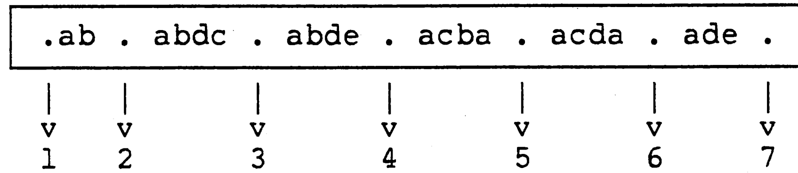

Figure 5. A "Full" B^+-Index Node

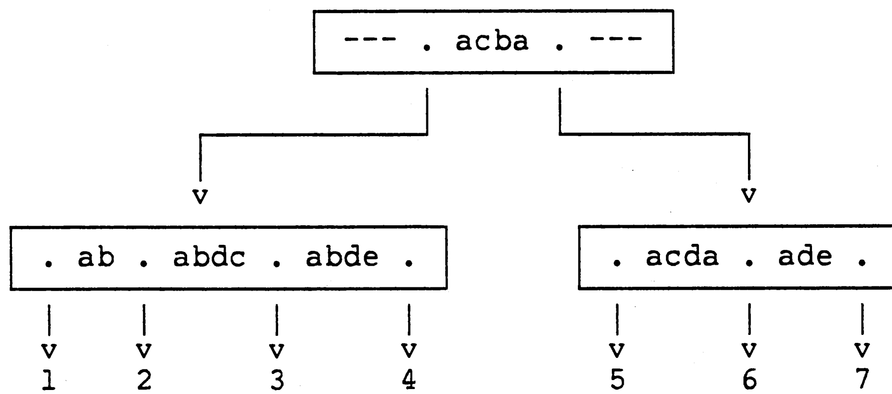

Figure 6. Index Nodes Resulting From a B-Tree Type Split of the B^+-Index Node in Figure 5

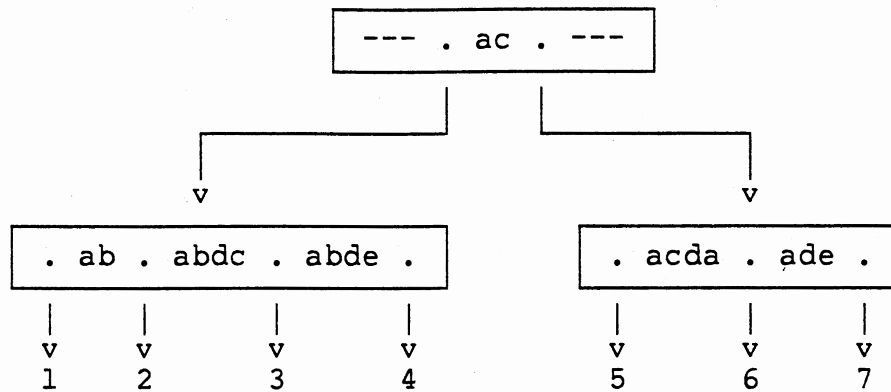

Figure 7. Incorrect Split of the B⁺-Index Node in Figure 5

Effect of Separators on a B⁺-Index

The movement of separators into a B⁺-index instead of full keys has a couple of pleasing effects on a B⁺-index. First of all, the height of a B⁺-index may be reduced because the branching degree of index nodes has been increased due to the existence of shorter strings in the index. Second, because separators are usually shorter than full keys more separators may be packed into a node thereby reducing the total number of index nodes required to make up a B⁺-index.

Split Interval

In an attempt to move shorter separators into the B^+-index of a simple prefix B^+-tree, Bayer and Unterauer introduced the concept of a "split interval". Normally, when a node in the B^+-file of a simple prefix B^+-tree splits, the middle of the leaf node is found, a separator is constructed, and that separator is passed up to the parent node. However, when a split interval is used, the middle of the leaf node is found, separators are constructed for all keys within the interval, and the shortest separator within the interval is then passed up to the parent node. Thus, a split interval is merely the number of keys or separators around the middle of the node which are considered for choosing a suitable split point.

The idea of a split interval can also be applied to an index node. In this case, the shortest separator within the split interval is propagated up to the parent.

The question of finding the middle of a node is more complex in simple prefix B^+-trees than in conventional B-trees. This is due to the variable length strings that occur in simple prefix B^+-trees. In this paper, there are two ways to view the middle of a node, the logical and physical middle. The physical middle of a node is the point where a node is bisected into two equal halfs. The logical middle of a node is the split point nearest the physical middle of a node.

A split interval may be defined more precisely as a count of the number of "gaps" around, and including the logical middle of a node, that are candidate split points. A gap is the interval (x(i),x(i+1)], where x(i) are keys or separators in collating sequence order. Thus, a node of cardinality (n-1) defines n gaps. Note that a "]" indicates that the key or separator x(i+1) is included in the interval, while a "(" indicates that the key x(i) is not included in the interval. A separator s is said to "fill" a gap (x(i),x(i+1)] if and only if x(i) < s <= x(i+1). A split interval of three may now be defined as the three gaps, (x(i-2),x(i-1)], (x(i-1),x(i)], and (x(i),x(i+1)] where the gap (x(i-1),x(i)], is the logical middle of the node. For the remainder of this paper, the logical middle of a node will be known simply as the "middle" gap. In general, a split interval of "k" means that (k-1)/2 gaps on either side of the middle gap will be examined to determine the best split point for the node. All split intervals in this paper are symmetrical around the middle gap and therefore all split intervals are required to take on odd values. Even numbered split intervals define non-symmetrical split intervals. Note that a split interval of one is equivalent to the conventional split associated with simple prefix B^+-trees.

For an example of a split interval defined using gaps, see Figure 8 below where it is assumed that the node is a

leaf node about to split and that all records or pointers to records have been removed. Also, assume that the middle gap of the node is gap number three.

S	De	Der	Derivativ	Do	Doubt	
K	Data	Deque	Derivation	Derivative	Double	Doubt
G	1	2	3	4	5	

Figure 8. A B^+-File Node and the Separators Derived From Each Gap

In Figure 8, the first line is the separators derived at each gap, the second line is the keys residing in the node, and the third line is the gap numbers. If a split interval of one is used to split the node in Figure 8, the split point would be gap number three and the corresponding separator passed up would be "Derivativ". Recall that a split interval of one corresponds to the standard split associated with simple prefix B^+-trees. If a split interval of three is used, then separators to fill gaps 2, 3, and 4 would be calculated and gap 4 would be chosen as the split point due to the length of the separator "Do". If a split interval of five is used, then separators to fill gaps 1, 2, 3, 4, and 5 will be constructed. In this case, gap 4 will be chosen over gap 1 because gap 4 is closer to the middle

of the node. Note that gaps occurring off the end of the node are ignored since they do not define a suitable split point for the node.

Bayer and Unterauer conjecture the following effects of SIL, split interval for leaf nodes, and SIB, split interval for branch or index nodes, on a simple prefix B^+-tree (2, p.14):

1. An increase of SIL should decrease the average length of separators in the B^+-index, thereby reducing the number of nodes required for the index part of the tree.
2. An increase of SIB should favor the shorter separators in the index to be located near the root, thereby increasing the the branching degree of nodes near the root, where a high branching degree is most beneficial.
3. Increasing both SIB and SIL causes the height of the B^+-index to decrease but also decreases the storage utilization because nodes may now be less than half full.

A Difference of Opinion

Part of point number two above has been seriously questioned by Rosenberg and Snyder (13). They object to the notion that a high branching degree is most beneficial near the root. They concede that a high branching degree of nodes near the root causes the tree height to decrease and

they call trees that are as short as possible "visit-optimal". This is because the shortest trees require the minimum number of accesses or "visits" to traverse the tree from root to leaf. Unfortunately, they point out that trees that are "visit-optimal" are also nearly "space-maximal". Space maximality means that more nodes than are actually required are used to construct the tree and many of these nodes have a low branching degree. These extra nodes are located in the lower levels of the tree and are caused by the high branching degree of nodes located above them.

Rosenberg and Snyder (13) conclude that a high branching degree is actually most beneficial in the lower levels of the tree. Their study indicates that trees with a high branching degree in the lower level nodes are "space-optimal". Space optimality means that the tree was constructed from a minimum number of nodes. Furthermore, trees that are "space-optimal" are nearly "visit-optimal". Therefore, "space-optimal" trees are not only compact, meaning that a minimum of nodes were used to construct them, but also short enough to provide good performance when searching them.

In order to make B-tree type structures "space-optimal", Rosenberg and Snyder (13) provide a linear time, in place compaction algorithm for B-tree type structures. This scheme may be executed when B-tree type files are "backed up" for error recovery or archival purposes. After a tree "backup" is done, the compaction algorithm may be

executed to restore the tree to a compacted form.

They also mention that very large trees in compacted "space-optimal" form will not degenerate considerably from this compacted form due to insertions and deletions if the tree is relatively static between backup reorganizations. Relatively static here means that there is no more than a 2.5 percent change in the tree between "backup" reorganizations. The authors feel that this rate is not unreasonable for very large data bases. This matter will be addressed again in chapter four when the effect of split intervals on simple prefix B^+-trees is analyzed.

Insertion Utilizing a Split Interval

Insertion into a simple prefix B^+-tree where a split interval is utilized begins by searching the B^+-index to find the proper leaf node. If the leaf node must be split, choose the gap within the split interval yielding the shortest separator and split the node at that gap propagating the separator to the parent. If insertion into an index or branch node causes a split, then choose the shortest separator within the split interval and pass it up to the parent node.

One way to postpone a split and increase storage utilization is the use of an overflow. However, because separators are variable in length, a separator in the parent node may now be replaced by either a longer or shorter separator. Therefore, overflows may now propagate and cause

further splits or overflows if a separator is replaced by a longer separator. Such propagation is expected to be infrequent. Note also that split intervals may also be applied to overflows.

Deletion Utilizing a Split Interval

Deletion from a simple prefix B^+-tree always occurs in a leaf node. If a deletion causes two leaves to be merged then the corresponding separator in the parent node must be deleted. This deletion will always be a deletion from a leaf node of the B^+-index, which is organized as a B-tree. Thus, these deletions are simpler than general deletions from B-trees and other separators in the B^+-index are not affected by such a deletion. If an index node must be merged with a sibling due to a deletion, then the separator separating the two nodes to be merged must be deleted from the parent node.

Underflows are another way to handle deletions. In this case, keys and records or separators are moved from a sibling to the node where the deletion occurred. This movement of keys or separators may then cause a separator in the parent node to be replaced by either a longer or shorter separator. The problem of replacing separators in the parent node during underflows is analogous to the situation associated with overflows. Split intervals may also be applied to underflows.

Prefix B^+-Trees

In many practical applications, like textual databases, sets of keys that arise are often clustered together. Clustering means that the collating sequence "distance" between successive separators is small. Therefore, all separators in a given subtree of a simple prefix B^+-tree may share a common prefix. Bayer and Unterauer (2) propose removing this common prefix from separators in a simple prefix B^+-tree in order to further reduce the height and size of the B^+-index. They suggest that the common prefix be kept in the predecessor nodes rather than repeatedly stored in the subtree itself.

Figure 9 is a partial subtree of a simple prefix B^+-tree. Consider node T. The parent of node T contains LL(T) and SU(T) which are the largest lower bound and the smallest upper bound of node T. For all keys, k, and separators, s, which are or might be stored in node T or the subtree with node T as the root, the following holds:

$$LL(T) \le k < SU(T)$$

$$LL(T) \le s < SU(T)$$

In node T, p(0), p(1), ... , p(j) are pointers to the successors of node T and s(1), s(2), ... , s(j) are separators.

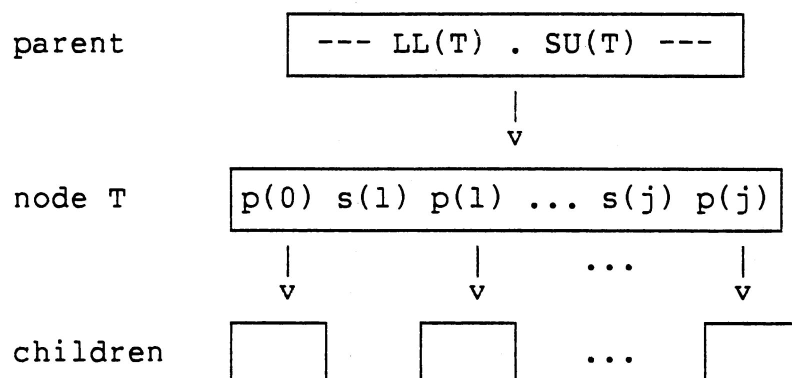

Figure 9. A Partial Subtree of a Simple Prefix B^+-Tree

LL(p(i)) and SU(p(i)) are the largest lower bound and the smallest upper bound of the subtree pointed to by p(i). Therefore, to find the largest lower bound and smallest upper bound of a given subtree of node T, the following should be used.

$$LL(p(i)) = s(i) \quad \text{for } i = 1,2,\ldots,j.$$
$$= LL(T) \quad \text{for } i = 0.$$

$$SU(p(i)) = s(i) \quad \text{for } i = 0,1,\ldots,j-1.$$
$$= SU(T) \quad \text{for } i = j.$$

Therefore, if a subtree of node T contains a nonempty common prefix k(i), it must be defined as follows: Let k(i) be the longest common prefix (possibly the "null" or "empty"

string) of LL(p(i)) and SU(p(i)), then the common prefix k(i) of the subtree rooted at p(i) is defined as follows:

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

The reader should note that in the definition of the common prefix k(i) above, k(i)l(j)z and k(i)l(j+1) are concatenations of characters derived from separators.

Now reconsider the simple prefix B^+-tree of Figure 4. When common prefixes are factored out of the tree, the prefix B^+-tree that results is shown in Figure 10. Separators appearing in the B^+-index of a prefix B^+-tree are known as partial separators because the common prefix has been removed.

Algorithms for search, insertion and deletion in prefix B^+-trees are given in the article by Bayer and Unterauer but not repeated here because prefix B^+-trees are not the main concern of this paper. However, it should be noted that the algorithms for these operations are more complex than their simple prefix B^+-tree counterparts. For instance, suppose that an insertion into the parent node of T in Figure 9 changes the common prefix for node T. This change may then cause the partial separators appearing in node T to shrink or expand. A reduction in the length of partial separators

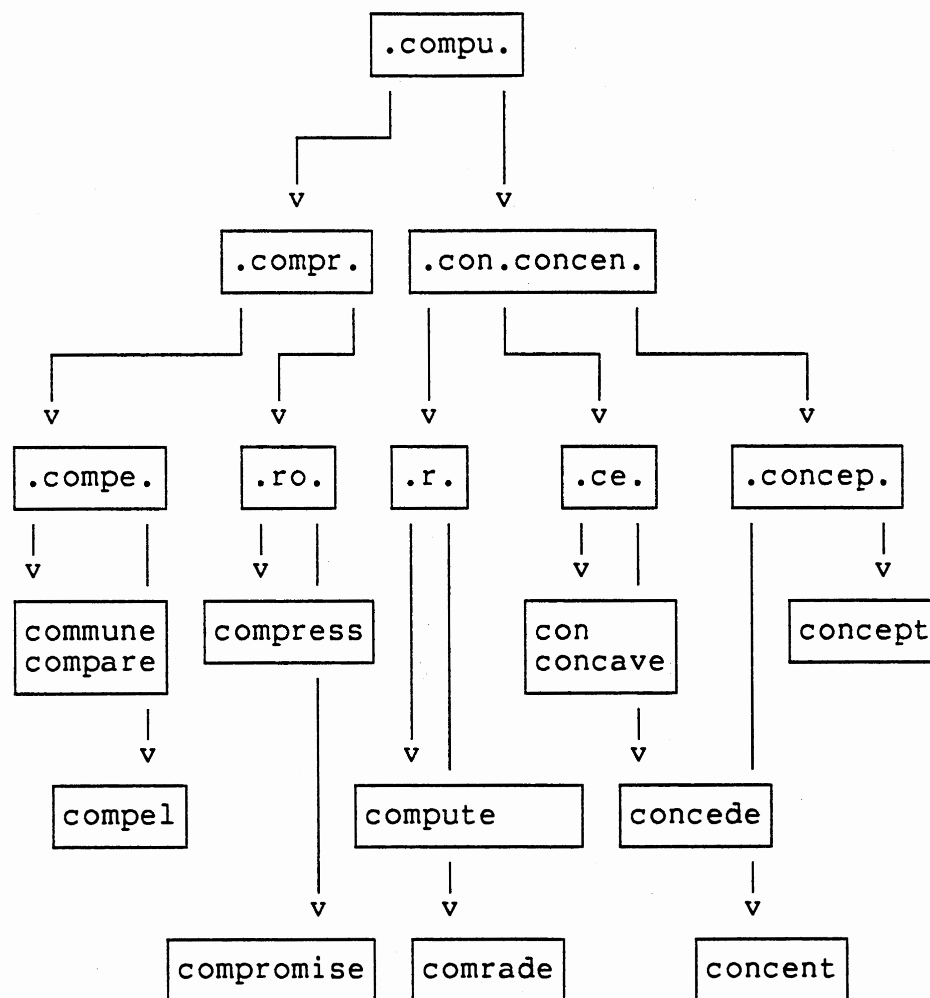

Figure 10. A Prefix B^+-Tree Derived From the Simple Prefix B^+-Tree in Figure 4

may now cause the node T to be less than 50 percent full and require a merge or underflow operation. Expansion of partial separators may now cause the node T to be full and therefore require a node split or overflow. Deletion from the parent of node T may cause an analogous situation to arise.

Summary

The introduction of shorter strings into the B^+-index of simple prefix B^+-trees serves to increase the branching factor of index nodes and thereby decrease the height of the B^+-index. The reduction of height in the B^+-index is important because it reduces the number of external storage accesses required to traverse the tree.

However, additional complexity is introduced in some areas when this type of indexing is used. They are:

1. Algorithms for search, insertion, and deletion must be capable of handling variable length strings.
2. Additional time is required to search a node due to the variable length of separators occurring in the node. A node containing fixed length keys or separators could be searched using a binary search (9, section 6.2.4). This is not possible with variable length strings.
3. If prefix B^+-trees are utilized, additional processing may be required for some insertions or deletions where the common prefix of a given subtree

is altered.

Bayer and Unterauer (2 ,p.16) provide some experimental results concerning computing time and disk accesses when a B^+-tree, a simple prefix B^+-tree, and a prefix B^+-tree are compared.

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

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

Feng (5) suggests that the above results indicate that simple prefix B^+-trees are more cost effective than prefix B^+-trees in a dynamic environment. However, in a static environment a prefix B^+-tree may be superior to a simple prefix B^+-tree because minimizing the search time of a static index is more important than minimizing its initial construction time. A more thorough analysis of prefix B^+-trees may be found in the paper by Feng.

CHAPTER IV

EMPIRICAL MEASUREMENT CONCERNING THE EFFECT A SPLIT INTERVAL HAS ON SIMPLE PREFIX B^+-TREES

Bayer and Unterauer (2, p.12) introduce the concept of a split interval for both the index or branch and leaf nodes. Their motivation for using a split interval is that shorter separators will be moved up into the index part of the tree thereby increasing the branching degree of upper level nodes and decreasing the height and size of the B^+-index. In their paper, they have implemented and tested both a simple prefix B^+-tree and a prefix B^+-tree but do not include the concept of a split interval in their analysis. Therefore, the primary focus of this paper is an empirical examination of the effect of split intervals on a simple prefix B^+-tree. In particular, the following four areas will be studied to see how a split interval effects them. The four areas are:

1. Average length of separators occurring in the B^+-index.
2. Height of the B^+-index.
3. Storage utilization of nodes at each level of the

tree.

4. Total storage requirement for the tree.

After studying the effect of a split interval on these four areas, an attempt to determine the value of a split interval for both branch and leaf nodes will be made.

Test Cases

To facilitate empirical measurement, a simple prefix B^+-tree was implemented and a set of test cases was designed to determine the effect a split interval has on a simple prefix B^+-tree. The experimental implementation of a simple prefix B^+-tree is described later in this chapter.

Each test case consisted of inserting 20,000 randomly selected keys into an initially empty tree. The source for all keys inserted in all test cases was the "inwrds" file. This file consisted of a wide variety of words from the English language. Words from the "inwrds" file were randomly selected for insertion by using an algorithm from Knuth (8). The "inwrds" file and the method of random selection will be discussed in more detail later in this chapter.

All nodes of the experimental simple prefix B^+-tree had a node size of 128 locations, where a location may store an integer or character. This node size is probably too small for practical applications but was set to 128 to force the index part of the tree to be larger than if a greater node size had been chosen.

Split intervals for both branch and leaf nodes were allowed to take on the values one, three, and five. The set of test cases used consisted of every possible combination of split intervals for branch and leaf nodes. In other words, the first test case was (SIB=1, SIL=1) and the last test case was (SIB=5, SIL=5) and there were nine total test cases. Split intervals greater than five are generally too large for the node size of 128 because they define split points that may be off the end of a node.

Each test case was replicated ten times to provide a body of data substantial enough to provide valid empirical data when averaged. Replications across test cases consisted of the same sequence of keys being selected from the "inwrds" file for insertion. In other words, replication "n" of test case (SIB=1, SIL=1) consisted of the same sequence of keys to be inserted as did replication "n" of test case (SIB=3, SIL=1). This was necessary to allow a valid comparison of test cases.

Implementation of a Simple Prefix B^+-Tree

All test cases were run on an experimental implementation of a simple prefix B^+-tree. This implementation was written in the programming language "C" and was executed on a Perkin Elmer 3230 running the UNIX (14) operating system. High-level Program Design Language (PDL) descriptions of the algorithms used in the

experimental implementation are provided at the end of this paper in the Appendix. The purpose of this section is to provide the reader with a description of the pertinent parts of the implementation.

Node Organization

Separators appearing in a simple prefix B^+-tree are variable in length. Thus, nodes used to construct a simple prefix B^+-tree must have a different organization than nodes used to construct B-trees or B^+-trees where fixed length keys are utilized.

The node organization used for the simple prefix B^+-trees constructed for this paper is shown in Figure 11. L/B is an integer value which is set to one if the node is a leaf or is set to zero if the node is a branch node. L/B was used primarily by the tree search algorithm to determine when the leaf level of the tree had been reached. NL is a count of the number of locations used in the node. NL could be a count of the number of bytes used in the node but, in this implementation the node is conceptualized as an array of storage locations, where each location is capable of storing a two byte integer or a one byte character. This matter will be discussed further in the next section. NS is an integer value which is a count of the number of separators or keys in the node. Separator length l(i), is the length in characters of separator s(i). All separator lengths l(i), are integer values which are packed together

into the left end of the node. Separators and pointers are shown as s(i) and p(i), respectively.

```
L/B|NL|NS|l(1)|...|l(j)|p(0)|s(1)|...|s(j)|p(j)|---
```

Figure 11. Node Organization

All nodes used in the experimental implementation were identical in structure. An entry in a node consisted of a key or separator and pointer pair. Pointers in index or branch nodes are integers containing the node number of a child node, while pointers at the leaf level contained the "null" value, integer -1.

Searching a node organized like the node shown in Figure 11 requires a linear search because the location of separators within the node is not previously known. In contrast to nodes containing fixed length strings, where a binary search of the node is possible, a linear search means additional computing time to locate the string in question. This may develop into a performance bottleneck if the nodes are very large.

The node organization described here is not the only feasible organization. In a paper by Lomet (10), he describes a node organization for such a node in which

separators of equal length are organized into tables within the node. The motivation for this type of organization is that a majority of the storage allocated in the node for separator lengths can be eliminated and thereby free more space for storage of separators. Each table within the node implicitly supplies the length of all separators in that table.

The Structure of a Node

The node organization presented in the previous section was implemented using the "C" programming language structure shown in Figure 12. This structure served as a page buffer in the experimental implementation and was dimensioned to allow four pages or nodes of the tree to be present simultaneously in main memory. Note that a page is equivalent to a node of a tree in this discussion.

```
struct node {
  short leaf;
  int   num_locs_used;
  int   num_of_seps;
  union {
    short ival;
    char  cval;
  } store[ ];
} page[4];
```

Figure 12. The Page Buffer

Recall from the previous section that a node was conceptualized as an array of locations where each location was capable of storing a two byte integer or a one byte character. This is due to the "union" data type used in the page buffer declaration shown in Figure 12. The "union" data type was used to simplify the implementation of a node and the algorithms used to modify it. The "union" data type causes storage to be allocated for the largest data type present in the union. In this case, each element in the array "store" of Figure 12 has two bytes allocated to it. This allocation corresponds to the allocation for a short integer. The array "store" is used to store separator lengths, separators, and pointers.

In this study, if a node is said to have size "x" it means that the array "store" has been dimensioned to "x". Note that a node of size "x" does not correspond to "x" contiguous bytes of memory. This is because a character stored in the array "store" only utilizes one byte of the two bytes allocated to each element of the array. However, the "union" data type does allow "x" contiguous bytes to be simulated.

Page Replacement Method

In an attempt to reduce the number of I/O transactions required to perform operations such as insertion and preorder traversal, the "least recently used" or "LRU" page replacement algorithm was utilized (9). This algorithm

requires the page in the page buffer that was referenced the longest time ago to be replaced when a page is requested from external storage. This page replacement policy was used because simple prefix B^+-tree restructuring frequently requires that three nodes (two siblings and a parent) be in memory simultaneously. The LRU algorithm guarantees that these nodes will stay in memory during the restructuring.

The LRU method was implemented using two data structures, a page buffer and a page queue. The page buffer is shown in Figure 12 and is capable of storing four pages in main memory simultaneously. The page queue was used to store the page numbers of the pages currently present in the page buffer. An entry in the page queue consisted of not only a page number but also a pointer to the location of that page in the page buffer. The page queue was also used to retain the relative order in which the pages in the page buffer were referenced.

The LRU page replacement policy worked well for the experimental implementation. Most trees studied had a height of four or less and because the page buffer was dimensioned to four, the root page was nearly always resident in the page buffer. This is important because any search of the tree begins with the root and if the root page is present in the page buffer, at least one fewer I/O transaction is required to search the tree.

Implementation of a Split Interval

To split a node where a split interval is utilized, one must first find the "middle" gap of a node. After finding the "middle" gap, other gaps on either side of the "middle" gap that are part of the split interval are considered as possible split points.

The problem of finding the middle gap of a node is more complex in simple prefix B^+-trees than in conventional B-trees because of the variable length strings that occur in simple prefix B^+-trees. In this implementation, the following procedure was used to locate the middle gap of a node.

1. Subtract the number of separators from the total number of locations used in the node. This is the physical middle of the node.
2. Find the first gap immediately to the right of the physical middle found in step 1. This gap then qualifies as the "middle" gap.

Although the two step procedure given above does not always find the gap nearest the physical middle of a node, the procedure was easy to implement and no doubt found the "proper" middle gap at least half the time.

After locating the middle gap, the split interval, SIB or SIL, is referenced to determine how many gaps on either side of the middle gap should be split point candidates.

Recall that the criterion for determining the best split point within the split interval is the gap that provides the shortest separator to propagate up. However, in this implementation, an additional constraint has been placed on this selection process. If two or more gaps within the split interval determine separators of equal length, then the gap closest to the middle gap is chosen. This was done in hope of reducing the number of nodes whose storage utilization is far below 50 percent due to the split interval.

"Words" File and Random Sampling

The source of all keys inserted into the trees studied in this paper was the "words" file used by the UNIX (14) operating system to facilitate the checking of spelling in documents. The file is made up of a great variety of words from the English language, including technical words, abbreviations, numbers, and proper nouns. It is sorted by ASCII collating sequence and consists of 24001 variable length strings.

The "words" file described above is not a good source for random insertions into a tree structure. Because it is sorted, any random selection algorithm that proceeds from one end of the file to the other will insert an ordered partition of the file. The insertion of this ordered partition will force the tree to take a worst-case form where most nodes have a storage utilization of only 50

percent. In order to avoid this situation, an algorithm from Knuth (8) was used to "shuffle" the file randomly.

The algorithm used to shuffle the file required the file to be in a relative record format to facilitate addressing. The transformation to a relative record format was done by transforming each variable length string in the "words" file to a fixed length record stored in the "inwrds" file. The shuffling algorithm makes one complete pass over the "inwrds" file swapping the current record with a randomly chosen second record located somewhere below the current record in the file. Initially, the current record is the first record in the file.

Another algorithm from Knuth (8) was used to do random selecting of words from the "inwrds" file. This algorithm made a single pass over the file conditionally selecting words using a random number generator. Each record may be selected with probability, p/n, where p is the number of words to be selected and n is the cardinality of the file.

Before the nine test cases of this study were executed on the experimental implementation, the "inwrds" file was shuffled. This gave the "inwrds" file an order that was retained for each replication of all nine test cases. After shuffling, the random selection algorithm was executed for each replication of each test case. To insure that the same sequence of keys was selected for each replication across the test cases, the same random number generator seed was used to initialize the random selection algorithm for each

replication across test cases. In other words, replication "n" of test case "x" had the same random number seed as replication "n" of test case "y".

Using a file like the "inwrds" file to build simple prefix B^+-trees, represents a real application of simple prefix B^+-trees. Words in the "inwrds" file are variable in length and many intervals within the file contain clusters of words whose collating sequence distance is small. Thus, the simple prefix B^+-trees built for this study correspond to a word index like the ones used for a document database or dictionary.

Statistics Module

As a part of the experimental implementation, a module was written to collect statistics concerning the structure of the tree. The statistics were collected by traversing the tree in preorder. The following statistics concerning separator length were compiled for each iteration of each test case: 1) mean, 2) standard deviation, 3) minimum length, 4) maximum length, and 5) count of the total number of keys or separators. These statistics were compiled for each level of the tree.

Additionally, the following statistics were compiled for each level of the tree concerning storage utilization: 1) mean, 2) standard deviation, 3) minimum storage utilization, 4) maximum storage utilization, and 5) count of the total number of nodes at each level. Also, a count of

the total number of nodes used in the construction of the tree was calculated.

The calculation of storage utilization requires a special note. Only the elements of the array "store" in Figure 12 were considered in the calculation of storage utilization. Thus, storage utilization was calculated by dividing the number of locations of array "store" used by the dimension of the array. Other overhead elements of the node such as the number of separators stored were ignored in the calculation because these overhead elements were ignored in past studies involving average storage utilization in B-trees.

Results and Analysis of Empirical Testing

The nine test cases outlined earlier in this chapter were run and the information derived from these test cases was compiled for analysis. The data used to represent each test case was compiled by averaging the ten replications of each test case. Tables appearing in this section were derived from this compiled data and are presented to provide empirical evidence of the effect a split interval has on a simple prefix B^+-tree.

All trees produced by the nine test cases had a height of four. In the discussion that follows, the levels of a tree are numbered from zero to three where level zero is the root level and level three is the leaf level. Also, the reader should recall that SIB is the split interval for

branch or index nodes and that SIL is the split interval for leaf nodes.

Effect on Separator Length

Table I shows the effect of SIL on the length of the separators occurring in level two. The data in Table I is derived from the test cases where SIL took on the values one, three, and five while SIB was held constant at one. The reduction in separator length shown in Table I is quite significant as the average length of a key inserted into the tree was 7.19 characters.

TABLE I

AVERAGE LENGTH OF SEPARATORS AT LEVEL TWO
(SIB CONSTANT AT ONE)

SIL	AVE LEN	NODES L2	NODES LEAF	BD
1	4.47	151	2131	14.1
3	3.65	134	2139	16.0
5	3.46	132	2173	16.5

Table I also illustrates the reduction in the number of nodes required to make up level two and the increased branching degree achieved from the propagation of shorter

separators. However, note that the number of nodes required for the leaf level increases as SIL increases. This is due to the increased probability that a leaf node in the tree will split due to the uneven manner in which nodes split. As SIL increases, so does the difference between the storage utilization of the two nodes participating in the split. No analytical evidence that a highly uneven split of a node increases the probability that a node will split is provided here. However, it should be intuitively clear because the number of splits occurring at a level of the tree is always one less than the number of nodes making up that level of the tree.

Another consequence of highly uneven splits and the corresponding increased probability that a node will split is that additional separators are propagated up into the next level of the tree. Thus, as a split interval increases in size it selects shorter separators to propagate up, but it also increases the number of separators being propagated up due to the highly uneven splits. Therefore, the reduction in the size of the index part of the tree is not as significant as desired because the index now contains a greater number of separators.

Table II shows the effect SIB has on the length of separators occurring at level one of the tree. The data in this table comes from the test cases where SIB was varied but SIL was always one. Since SIL was constant, the average length of all separators entering the B^+-index was 4.47

characters as shown in Table I for test case (SIB=1, SIL=1).

TABLE II

AVERAGE LENGTH OF SEPARATORS AT LEVEL ONE
(SIL CONSTANT AT ONE)

SIB	AVE LEN	NODES L1	NODES L2	BD
1	4.00	9.7	151	15.5
3	3.44	9.6	152	15.8
5	2.75	8.0	153	19.1

Table II clearly indicates that shorter separators were propagated into the upper levels of the index as SIB increased. Also shown are the average number of nodes at level one and two and the increased branching degree achieved from propagating shorter separators into the upper levels of the index via an increase in SIB.

Effect on Storage Utilization

Table III presents storage characteristics for the leaf level where SIL was allowed to vary. Note that as SIL increases the average storage utilization does not decrease with any significance. However, an increase in SIL does

cause a decrease in the minimum storage utilization and an increase in the number of nodes comprising the leaf level. Evidence of highly uneven splits is shown by the reduction in minimum storage utilization. As mentioned earlier, these highly uneven splits cause an increase in the probability that a node at that level will split and therefore increase the storage requirement for that level.

TABLE III

STORAGE UTILIZATION AT THE LEAF LEVEL
(SIB HELD CONSTANT AT ONE)

SIL	AVE	MIN	NODES LEAF
1	68.14	39.14	2131
3	67.88	31.17	2139
5	66.86	21.03	2173

All trees constructed for this study were created by random insertions into an initially empty tree. B-trees constructed by random insertions into an initially empty tree achieve an average storage utilization of 69 percent as shown by Yao (18). Interestingly, this average storage utilization was also achieved by the trees in this study.

This result is interesting because B-trees utilize fixed length keys and nodes with preset maximimum and minimum branching degree, while simple prefix B^+-trees utilize variable length keys and nodes with a variable branching degree. Thus, one would not expect the same result concerning storage utilization because of the variable branching degree of nodes in a simple prefix B^+-tree. Also, overhead elements in the node were ignored in the calculation of average storage utilization just as Yao (18) did in his study on B-trees.

Effect on Tree Height

The data from the test cases used in this study did not provide any direct evidence that the use of a split interval caused a reduction in tree height. This is because the height of all trees studied was four. However, Table IV was constructed to show that conditions favorable to height reduction were present. Table IV shows the increased branching degree of nodes occurring at level one of the trees in test case (SIB=5, SIL=5) in comparison to the branching degree achieved by the trees in test case (SIB=1, SIL=1). It has been shown by Rosenberg and Snyder (13) that the shortest trees occur when the branching degree of nodes near the root is as high as possible.

TABLE IV

INCREASED BRANCHING DEGREE AT LEVEL ONE
DUE TO INCREASED SIB AND SIL

SIB	SIL	NODES L1	NODES L2	BD
1	1	9.7	151	15.5
5	5	6.5	131	20.2

Effect on Storage Requirements

In general, an increase of a split interval at level x tends to decrease the storage requirement for level (x-1) but increase the storage requirement for level x. Empirical evidence of this can be seen in Table I where the number of nodes required for level two and the leaf level is shown. This general effect is due to the propagation of shorter separators into level (x-1) thereby decreasing the number of nodes required to build level (x-1) and due to the increase of highly uneven splits at level x causing the number of nodes required for level x to increase. An increase in the number of node splits also means that more separators will

be propagated into the index part of the tree thereby reducing the significance of the reduction in index size.

Table V shows the storage requirement for a B^+-file or leaf level and B^+-index. Also the total storage requirement for the entire tree is shown. Note that the B^+-index decreases in size as SIL increases and the leaf level or B^+-file increases in size as SIL increases. Also note that SIB has virtually no effect on the size of the index. Unfortunately, as SIL increases the total storage requirement of the tree seems to increase. The reduction in size of the B^+-index is offset by the increase in the size of the B^+-file and thereby increases the total storage requirement for the tree. However, the data in Table V does include a possible exception to the increase in total storage requirement. The data for the test cases where SIL = 3 shows an actual reduction in total storage requirement for the tree. This exception may indicate an argument for a small split interval at the leaf level.

"Student's" t-Test

The analysis associated with this study has been primarily a comparison of means provided by averaging the ten replications of each test case. However, to provide some assurance that these means are statistically significant and not the result of random fluctuations, the "Student's" t-test (17) was applied to the data. The "Student's" t-test makes the assumption that the parent

population is the normal distribution and is used when the sample size is very small (less than 50). In this study, the sample size was ten for each test case.

TABLE V

TOTAL STORAGE REQUIREMENT FOR B^+-FILE, B^+-INDEX, AND ENTIRE TREE

SIB	SIL	B^+-FILE	B^+-INDEX	TOTAL
1	1	2131.2	161.4	2292.6
1	3	2139.4	143.1	2282.5
1	5	2172.6	140.7	2313.3
3	1	2131.2	162.2	2293.4
3	3	2139.4	143.0	2282.4
3	5	2172.6	137.4	2310.0
5	1	2131.2	160.9	2292.1
5	3	2139.4	144.5	2283.9
5	5	2172.6	138.0	2310.6

In particular, a 95 percent confidence interval for the difference in storage requirement means was calculated. In doing this, it was assumed that the test cases were "paired" samples. This was assumed because each sample or replication across each test case consisted of an identical sequence of keys to be inserted. In calculating a confidence interval for a difference in storage requirement means, the test case (SIB=1, SIL=1) was used as a reference

point or control because it corresponds to the conventional split at the "middle" of a node. The reader should note that the assumptions concerning the normal distribution as the parent population and paired samples may be questionable.

Table VI shows the result of calculating a 95 percent confidence interval for the difference in storage requirement means utilizing the "Student's" t-test. The results presented in Table VI involve the total storage requirement for the index part of the tree, leaf level, and the entire tree. A difference in means is considered to be statistically significant if the data point 0.0 does not appear within the interval. Thus, for the test case (SIB=1, SIL=3) the average reduction in the storage requirement for the index part of the tree was statistically significant while the average increase in the storage requirement for the leaf level was not. Table VI serves to verify the results shown in Table V and the discussion in the previous section concerning storage requirement.

One consequence of using SIL is that more nodes are required to make up the leaf level. This is shown in Table VI. The introduction of more nodes at the leaf level means that the average branching degree of nodes at the leaf level is reduced. This is unfortunate because the paper by Rosenberg and Snyder (13) points out that a high branching degree is most beneficial at the lowest levels of the tree.

TABLE VI

95% CONFIDENCE INTERVAL FOR THE DIFFERENCE IN STORAGE REQUIREMENT MEANS

SIB	SIL	LEAF LEVEL	INDEX	TOTAL
1	3	+8.2 ± 12.4	-18.2 ± 3.5	-10.1 ± 12.3
1	5	+41.4 ± 10.3	-20.6 ± 3.2	+20.7 ± 10.7
3	1	+0.0 ± 0.0	+0.9 ± 2.6	+0.8 ± 2.6
3	3	+8.2 ± 12.4	-18.3 ± 5.5	-10.2 ± 12.9
3	5	+41.4 ± 10.3	-24.2 ± 4.3	+17.2 ± 10.8
5	1	+0.0 ± 0.0	-0.4 ± 2.9	+0.8 ± 2.8
5	3	+8.2 ± 12.4	-18.7 ± 3.1	-10.6 ± 12.8
5	5	+41.4 ± 10.3	-21.0 ± 2.7	+20.3 ± 11.0

The specification of which level of the tree is the leaf level may be a source of some confusion here. This is because the leaf level of the simple prefix B^+-trees used in this study contains pointers to hypothetical records instead of full records. In the analysis of Rosenberg and Snyder (13), they consider the records themselves to be the leaf level and the B^+-file level of a simple prefix B^+-tree to be part of the tree index. Therefore, a high branching degree in simple prefix B^+-trees is most beneficial at the B^+-file or leaf level.

CHAPTER V

SUMMARY, CONCLUSIONS, AND SUGGESTED FURTHER RESEARCH

Summary

A simple prefix B^+-tree is a B^+-tree where the index or nonleaf part of a B^+-tree has been replaced by an equivalent index made up of shorter strings known as separators. Separators are derived from the actual keys residing in the leaf level of the tree and are used in the index part of the tree instead of actual keys to decrease the height and size of the index part of the tree.

A split interval permits a node in a simple prefix B^+-tree to split in other places besides the middle of a node. This is done to promote even shorter separators into the index part of a simple prefix B^+-tree and thereby further reduce the height and size of the index part of the tree. Split intervals may be applied to leaf and branch nodes of a simple prefix B^+-tree.

This paper is an empirical study concerning the effect a split interval has on the performance of a simple prefix B^+-tree. To facilitate this study, a simple prefix B^+-tree

utilizing the concept of a split interval was implemented and a set of test cases were executed on this implementation to derive information concerning the effect a split interval has on the performance of the tree. Data from these test cases were compiled and organized into tables to illustrate the effect of a split interval on a simple prefix B^{+}-tree.

The general effect of a split interval at level x of a simple prefix B^{+}-tree is to decrease the storage requirement for level (x-1) and increase the storage requirement for level x. The storage requirement for level (x-1) is reduced because shorter separators are selected by the split interval. However, the storage requirement for level x is increased because as a split interval increases in size so does the difference in storage utilization between the two nodes that result from the split. Highly uneven splits at level x causes the probability that a node at that level will split to increase and thereby increases the number of nodes required to make up level x. Highly uneven splits also increase the number of separators being propagated up into the index part of the tree and thereby reduce the significance of the reduction in total storage requirement for the tree.

Conclusions

The use of a split interval at the leaf level promotes shorter separators into the index part of the tree and thereby reduces the size and possibly the height of the

index part of the tree. However, the data derived from the test cases used in this study show that the increase in the total storage requirement for the leaf level due to the split interval tends to offset the reduction in size of the index part of the tree. In most cases, the total storage requirement for the tree increased as the split interval at the leaf level increased. However, the test cases where the split interval at the leaf level was three showed an actual reduction in the total storage requirement for the tree. Therefore, the use of a small split interval at the leaf level may be worthwhile. This matter needs to be studied further before any concrete conclusions can be drawn, however.

The use of a split interval in the index part of the tree promotes shorter separators into the upper levels of the tree and increases the branching degree of the upper level nodes. However, the use of a split interval in the index part of the tree is not recommended because it has been shown by Rosenberg and Snyder (13) that a high branching degree in upper level nodes is not a desirable property.

Suggested Further Research

The data derived from the test cases used in his study show that a small split interval at the leaf level of the tree may be useful. This matter needs further research to make any solid conclusions, however. Also the paper by

Rosenberg and Snyder (13) concerning where in a B-tree type index a high branching degree of nodes is most beneficial needs further study including the compaction algorithm given there.

SELECTED BIBLIOGRAPHY

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

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

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

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

(5) Feng, A. L. "A Study of Two Competing Index Mechanisms: Prefix B^+-Tree and Trie Structures." (Unpub. M.S. Thesis, Oklahoma State University, 1982.)

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

(7) Held, G. and Stonebraker, M. "B-Tree Re-examined." Communicatins of the ACM, Vol. 21, No. 2 (Feb., 1978), 139-143.

(8) Knuth, D. E. The Art of Computer Programming Vol. 2: Seminumerical Algorithms, Addison Wesley Publ. Co., Reading Mass., 1973.

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

(10) Lomet, D.B. "Multi - Table Search for B - Tree Files." IBM Technical Disclosure Bulletin Vol. 22, No. 6 (Nov. 1979), 2565-2570.

(11) Bays, C. and Durham, S.D. ACM Transactions of Math. Software, Vol. 2 (1976), 59-64.

(12) McCreight, E.M. "Pagination of B* - Trees with Variable-Length Records." Communications of the ACM, Vol. 20, No. 9 (Sept. 1977), 670-674.

(13) Rosenberg, A.L. and Snyder, L. "Time and Space Optimality in B - Trees." ACM Transactions on Database Systems, Vol. 6, No. 1 (March 1981), 174-183.

(14) UNIX is a Trademark of Bell Laboratories.

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

(16) Webster, R. E. "B^+-Trees." (Unpub. M.S. Report, Oklahoma State University, 1980.)

(17) Wonnacott, T.H. and Wonnacott, R.J. Introductory Statistics, John Wiley and Sons, New York, 1972, 166-173.

(18) Yao, A.C. "On Random 2,3 Trees." Acta Informatica, Vol. 9, No. 3 (1978), 159-170.

APPENDIXES

SYMBOL LEGEND:

ancest_stk - ancestor stack used to store ancestory during a search.
buf_ptr - pointer to a page(node) in the page buffer.
file_name - character array containing the name of the tree storage file.
in_str - input string or input key.
in_strlen - length of input string, in_str.
key_file - file containing keys to insert into a tree.
match - flag indicating a key to be inserted is already present.
new_page_ptr - pointer to a new page in the page buffer that is required for a split.
nn_ptr - pointer to a new page in the page buffer that is required for a split.
node - a structure with the following parts:
1. leaf - flag that is set if node is a leaf.
2. number of separators.
3. number of locations used in the array store.
4. store - array containing separator lengths. pointers, and separator characters.

node_ptr - pointer to a location in the array "store" of a node.
num_of_pages - initial number of pages on the available page list.
on_ptr - pointer to a node in the page buffer currently a part of a tree.
page_buffer - buffer that is used to store nodes of the tree in main memory.
page_num - page number.
page_queue - queue containing page numbers and pointers to the location of a page in the page buffer. used to retain order of page references.
ptr - child pointer to be inserted.
root - contains the page number for the root.
sample_size - number of keys to insert.
sep - character array containing a separator.
sep_len - length of sep.
sib - split interval for branch or index nodes.
sil - split interval for leaf nodes.
sl_ptr - pointer to a location in the array "store" of a node containing a separator length. separator length.
split_ptr - pointer to the location in the array store of a node where a split will occur. will occur.
ssl_ptr - pointer to the location in the array store of a node containing the separator length of the separator to the right of split_ptr.
success - flag indicating whether or not an insertion was successful.

up_ptr - child pointer propagated up due to a split.
up_str - separator propagated up due to a split.
up_strlen - separator length associated with up_str.

```
main()

{
  open key_file;

  request file_name, sample_size, sil, sib, and
    num_of_pages from the user;

  init_tree(file_name, num_of_pages);

  select_sample(sample_size, key_file, sil, sib);

  close key_file;

  exit;
}
```

```
select_sample(sample_size, key_file, sil, sib)

{
  t = m = 0;
  while (m < sample_size) {

    read a word from key_file;

    if((size of key_file - t)*ranf(0) < sample_size) {
      in_strlen = length of word just read;
      tree_insert(in_str, in_strlen, sil, sib);
      m = m + 1;
    }
    t = t + 1;
  }
  return;
}
```

```
tree_insert(in_str, in_strlen, sil, sib)

{
  if (root = -1) {    /* empty tree */
    root = the next available page number;
    buf_ptr = lru_buffer(root);
    success = node_insert(in_str, in_strlen, buf_ptr, -1);
  }
  else {
    fetch the pages in the traversal path using lru_buffer
      and stack their page numbers on ancest_stk until a
      leaf node is reached;
  }

  success = node_insert(in_str, in_strlen, buf_ptr, -1);

  while (!success) { /* while insertion is unsuccessful */

    if (ancest_stk is empty) { /* split the root */
      page_num = the next avaiable page number;
      new_page_ptr = lru_buffer(page_num);
      split_ptr = get_split_point(buf_ptr,up_str,
                      up_strlen,ssl_ptr,sil,sib);
      split(split_ptr, new_page_ptr, buf_ptr, ssl_ptr);
      page_num = the next available page number;
      buf_ptr = lru_buffer(page_num);
      success = node_insert(up_str,up_strlen,buf_ptr,
                      up_ptr);
      root = page_num;
    }
    else {  /* split a node other than the root */
      up_ptr = the next available page number;
      new_page_ptr = lru_buffer(up_ptr);
      split_ptr = get_split_point(buf_ptr,up_str,
                     up_strlen,ssl_ptr,sil,sib);
      split(split_ptr,new_page_ptr,buf_ptr,ssl_ptr);
      page_num = pop the ancest_stk;
      buf_ptr = lru_buffer(page_num);
      success = node_insert(up_str,up_strlen,buf_ptr,
                      up_ptr);
    }
  }
  return;
}
```

```
get_split_point(buf_ptr,sep,sep_len,ssl_ptr,sil,sib)

{
  find the middle gap of the node pointed to by buf_ptr;

  derive a separator for the middle gap and save it and
    its length in sep and sep_len respectively;

  examine the gaps to the left of the middle gap that
    are within sil or sib for a shorter separator. If a
    shorter separator is found, save it and its length
    in sep and sep_len;

  examine the gaps to the right of the middle gap that
    are within sil or sib for a shorter separator. Save
    in sep and sep_len if found;

  /* split_ptr is pointer to the gap where split will */
  /* occur. Also ssl_ptr is pointer to location of    */
  /* separator length, sep_len.                       */

  return(split_ptr);
}
```

```
init_tree(file_name, num_of_pages)

{
  initialize the page queue;

  create and open the tree storage file, file_name;

  initialize the tree storage file to all leaf nodes.
    The number of pages initialized is num_of_pages;

  initialize the available page list;

  set the root to -1 to indicate an empty tree;

  return;
}
```

```
lru_buffer(page_num)

{
  search the page queue to see if the requested page,
    page_num is present;

  if (page_num is not present in the page queue ) {

    /* page to be paged out is at front of page */
    /* queue.                                    */

    if (page to be paged out has been altered)
      write it to the page storage file;

    read the requested page, page_num from the
      storage file;

    place the requested page in the available space
      in the page buffer vacated by the page that
      was paged out;

    insert page_num at the rear of the page queue;
  }
  else
    move the page queue entry associated with
      page_num from its current position to the rear
      of the page queue making it the most recently
      referenced page;

  /* buf_ptr points to the location in the page */
  /* buffer where the page associated with      */
  /* page_num is located.                       */

  return(buf_ptr);
}
```

```
node_search(in_str,in_strlen,match,sl_ptr,buf_ptr)

{
  if (node pointed to by buf_ptr is empty)
    return(-1);   /* signals empty tree */
  else {
    sequentially search the node pointed to by buf_ptr
      for the proper position for insertion or the
      proper pointer to follow in the traversal path.
  }

  /* node_ptr is position for insertion or pointer to */
  /* next pointer in traversal path.                   */

  return(node_ptr);
}
```

```
put_in_node(node_ptr,sl_ptr,in_str,in_strlen,ptr,buf_ptr)

{
  calculate the number of locations needed for the
    insertion;

  set success to false if node will be overfull due
    to the insertion. set to true otherwise;

  shift the separators and pointers to the right of
    node_ptr to the right to allow room for the
    insertion;

  insert in_str into the node;

  shift the contents of the node between node_ptr and
    sl_ptr to the right to allow room for the inser-
    tion of in_strlen;

  insert in_strlen;

  update the number of locations used and the number
    of separators for the node pointed to by buf_ptr;

  return(success);
}
```

```
node_insert(in_str, in_strlen, buf_ptr, ptr)

{
  node_ptr = node_search(in_str,in_strlen,match,
                              ,sl_ptr,buf_ptr);

  if (node_ptr = -1) {  /* empty tree */

    set the following variables in the node:
      1. number of locations used.
      2. number of separators.

    insert the following into the node:
      1. in_strlen.
      2. "null" pointer, -1.
      3. in_str.
      4. "null" link pointer, -1.

    success = true;
  }
  else {
    if (match) {  /* key already present */
      success = true;
    }
    else
      success = put_in_node(node_ptr,sl_ptr,in_str,
                            in_strlen,ptr,buf_ptr);
  }
  return(success);
}
```

```
split(split_ptr, nn_ptr, on_ptr, ssl_ptr)

{
  move separator lengths to the right of ssl_ptr
    (ssl_ptr + 1 for branch nodes) from the node
    pointed to by on_ptr to the node pointed to by
    nn_ptr;

  if (node pointed to by on_ptr = leaf) {
    move separators and pointers to the right of
      split_ptr from the node pointed to by on_ptr
      to the node pointed to by nn_ptr;

    link the two nodes together;
  }
  else {
    do the same as above except move all separators
      and pointers one gap to the right of split_ptr
      to the node pointed to by nn_ptr;
  }
  shift the separators and pointers remaining in the
    node pointed to by on_ptr to the left to compact
    the node;

  calculate the number of separators and locations used
    for the nodes pointed to by on_ptr and nn_ptr;

  return;
}
```

2

VITA

Timothy L. Towns

Candidate for the Degree of

Master of Science

Thesis: THE EFFECT OF A SPLIT INTERVAL ON SIMPLE SIMPLE PREFIX B^{+}-TREES

Major Field: Computing and Information Sciences

Biographical:

Personal Data: Born in Lyons, Kansas, July 1, 1957, the son of Marion L. Towns and Patrica R. Towns.

Education: Graduated from Putnam City West High School, Bethany, Oklahoma, in May, 1975; recieved Bachelor of Science degree in Mathematics from Harding University, Searcy, Arkansas, in December, 1979; completed requirements for the Master of Science degree at Oklahoma State University, Stillwater, Oklahoma, in July, 1983.

Professional Experience: Graduate Assistant, Oklahoma State University, Computing and Information Sciences Department, 1980 - 1983.