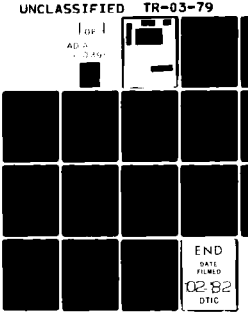
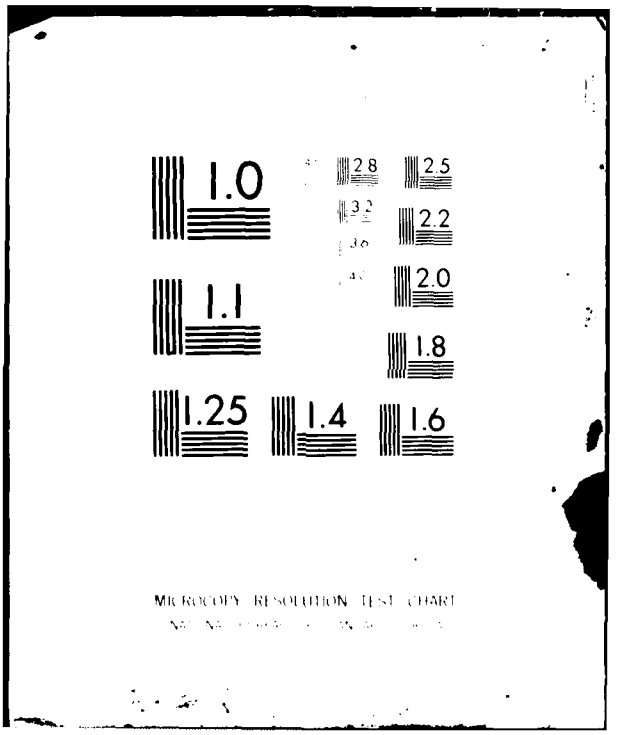
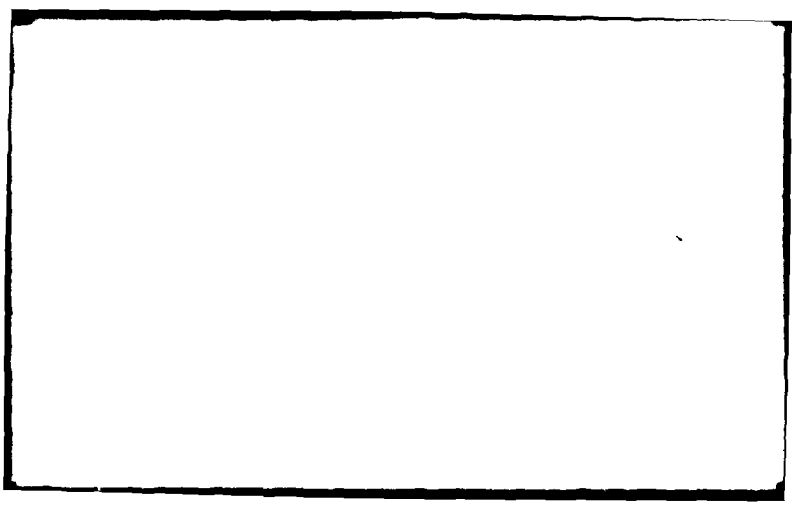
AD-A110 391 HARVARD UNIV CAMBRIDGE MA AIKEN COMPUTATION LAB F/G 12/1
THE SUPER-B-TREE ALGORITHM.(U)
JAN 79 D E WILLARD N00014-76-C-0914
UNCLASSIFIED TR-03-79 NL



END
DATE FILMED
02 82
DTIC


1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6
MICROCOPY RESOLUTION TEST CHART



unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) "The Super-B-Tree Algorithm"		5. TYPE OF REPORT & PERIOD COVERED technical report 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Willard, Dan E.		8. CONTRACT OR GRANT NUMBER(s) N00014-76-C-0914
9. PERFORMING ORGANIZATION NAME AND ADDRESS Harvard University Aiken Computation Laboratory Cambridge, Massacusetts 02138		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research 800 North Quincy Street Arlington, Virginia 22217		12. REPORT DATE January 1979 13. NUMBER OF PAGES 23
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

multidimensional searching
partial match retrieval
k-d tree
quad tree
bounded balance tree
B-tree
super-B-tree
augmented tree
pyramid

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

In the traditional literature concerning AVL 2-3, bounded balance or multiway B-trees, it has been assumed that a pointer-changing operation would require approximately one unit of runtime. This approximation is inapplicable to the augmented trees of BS77, Lu78b, Wi78a and Wi78c, because these trees associate an auxiliary data structure to each interior node, which complicates the cost of pointer-changing operations. Such an operation will consume 1 + Jw units of runtime in an augmented tree application (where

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

J denotes the number of records that gain a new ancestor as a result of the pointer-changing operation, and w is a coefficient depending on the particular application).

The purpose of this paper ~~will be~~ to propose an algorithm which is a generalization of traditional B-trees designed to possess an O(wlog N) worst-case insertion and deletion runtime in an augmented tree environment. This algorithm is significant because it has the optimal runtime order of magnitude.

12

THE SUPER-B-TREE ALGORITHM

by

Dan E. Willard*

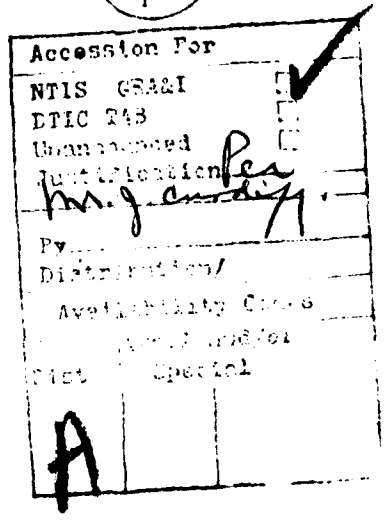
TR-03-79

*Dan E. Willard is now at Bell Laboratories, Holmdel, New Jersey.

Submitted January 12, 1979.

This research was supported in part by the Office of Naval Research under contract N00014-76-C-0914.

403831



ONR/Code 411 IS, Ms. Laura Watson, has been notified that this report is Copyrighted.

INFORMATIVE ABSTRACT

The Super-B-Tree Algorithm

By Dan E. Willard

Computer Science classification: 3.73, 3.74
Keywords: multidimensional searching, partial match retrieval, k-d tree, quad tree, bounded balance tree, B-tree, super-B-tree, augmented tree, pyramid

In the traditional literature concerning AVL, 2-3, bounded balance or multiway B-trees, it has been assumed that a pointer-changing operation would require approximately one unit of runtime. This approximation is inapplicable to the augmented trees of BS77, Lu78, LW78b, Wi78a and Wi78c, because these trees associate an auxiliary data structure to each interior node, which complicates the cost of pointer-changing operations. Such an operation will consume 1 + Jw units of runtime in an augmented tree application (where J denotes the number of records that gain a new ancestor as a result of the pointer-changing operation, and w is a coefficient depending on the particular application).

The purpose of this paper will be to propose an algorithm which is a generalization of traditional B-trees designed to possess an O(wlog N) worst-case insertion and deletion runtime in an augmented tree environment. This algorithm is significant because it has the optimal runtime order of magnitude.

The Super-B-Tree Algorithm

By Dan E. Willard

During the last year three papers (BS77, Lu78, Wi78a) have independently proposed the use of a new data structure whose most general form was called an augmented tree in Wi78c. An augmented tree is defined as a tree representation of a sorted list in which every interior node contains a new field of information which Wi78a calls a "subtree description structure (SDS)." In the context of an interior node v, the term "SDS" refers to any arbitrary user-defined data structure that describes v's descendants. If T is a tree sorted by KEY.1, then one example of an SDS(v) field is a sorted list that enumerates v's descendants by order of increasing KEY.2 value. Another example is a partial match data structure representation (Be75, Ri76, Wi78b, Wi78c) of v's descendants. The definition of SDS fields in Wi78a was carefully worded to encompass all future possibilities by indicating that SDS(v) may be any data structure describing v's descendants.

Research into the retrieval capacities of augmented trees only seriously began during the last year. BS77, Lu78, LW78b

and Wi78a verified that searches for the records satisfying a k-dimensional query of the form

$$a_1 < KEY.1 < b_1 \ \& \ a_2 < KEY.2 < b_2 \ \& \ \ldots \ a_k < KEY.k < b_k$$

can be performed in $O(\log^k N)$ time with a special augmented tree that Wi78c calls a $P_0(k)$ pyramid. More efficient augmented trees were also discussed in Wi78c. The article displays an improved data structure that enables k-dimensional queries to be performed in $O(\log^{k-1} N)$ time without any serious accompanying disadvantage. In the special case where k = 2, this result reduces to a data structure that occupies $O(N\log N)$ space and has the same $O(\log N)$ worst-case retrieval time as one-dimensional sorted lists. There are also many additional retrieval theorems about augmented trees mentioned in Wi78a and Wi78c.

The purpose of this paper will be to explain how records can be efficiently inserted into and deleted from augmented trees. Our algorithm, called the super-B-tree procedure, will be a generalization of traditional B-tree procedures which is specifically designed for the case of augmented trees. The runtime of this procedure can best be explained by letting w denote the amount of runtime needed for a record to be inserted into or deleted from an SDS field (in a particular application). Under the super-B-tree algorithm, any record can be inserted into or deleted from an augmented tree in $O(w\log N)$ worst-case runtime.

The super-B-tree procedure will be quite different from its traditional B-tree counterparts (such as the AVL, bounded-balance,

2-3, and multiway algorithms). To understand the reason for this, let J denote the number of records that gain or lose an ancestor as a result of a pointer-changing operation. Such operations will clearly require $O(1+Jw)$ runtime in the context of an augmented tree. Consideration of the degenerate case where $J \cong N$ indicates that all traditional B-tree algorithms must have an $O(wN)$ worst-case insertion and deletion runtime when manipulating augmented trees. The super-B-tree algorithm will thus require several further optimizations to attain its $O(w \log N)$ worst-case runtime.

In addition to considering worst-case runtime, this paper will also examine a weaker method of measuring runtime called CERT. The definition of CERT, introduced in Wi78a, requires that:

i) the symbol A denote an algorithm which performs insertion and deletion operations in a data structure denoted as D

ii) C denote a sequence of insertion and deletion commands whose length is denoted as $|C|$

iii) data structure D represent the empty set of records before command sequence C is executed

Under these circumstances, algorithm A will be said to have a CERT (the acronym stands for "Conservative Estimate of Runtime") equal to R iff $C|R|$ represents the maximum amount of time that any sequence C can force A to consume.

An article giving a preliminary explanation of the CERT-measured cost of insertion and deletion records in augmented trees was Lu78. The discussion in Lu78 centered around a data structure which was called a $P_0(k)$ pyramid in Wi78c. $P_0(k)$ pyramids are inductively defined as follows:

i) $P_0(1)$ is defined to be a standard tree whose records are sorted by increasing value in their first key

ii) $P_0(k)$ pyramids are augmented trees whose records are sorted by their k-th key and whose SDS fields are $P_0(k-1)$ pyramids

Lu78 displayed an algorithm that inserted and deleted records in $P_0(k)$ pyramids in $O(\log^k N)$ CERT time. The discussion in Lu78 did not consider general augmented trees of arbitrary complexity, and its main algorithm would require significant revision to process several augmented trees in $O(w\log N)$ CERT time. Also Lu78 did not optimize worst-case runtime.

Five months prior to the presentation of Lu78, I submitted a dissertation to the Harvard Mathematics Department (Wi78a) whose subject matter included an independent derivation of this material, as well as a large number of additional and more powerful algorithms. It was understandable that Lueker's presentation did not contain a reference to the more powerful theorems of Wi78a, as the Lu78 presentation was given only shortly thereafter.

The main purpose of Wi78a was to study retrieval in the context of the set of records satisfying arbitrarily complex predicates, and to show how retrieval -- as well as the evaluation of the set's sums, counts, multiplicative product, universal quantifiers, existential quantifiers, mean values, median values, maximum values, and minimum values -- can be performed in $O(\log^2 N)$ or less worst-case runtime for virtually all

commercial user requests. One of the seven chapters of Wi78a discussed an algorithm that was substantially more efficient than the somewhat similar update procedure which later appeared in Lu78. That chapter consisted of:

i) a demonstration that all traditional B-tree algorithms require O(wN) CERT time when applied to augmented trees

ii) a description of a procedure fundamentally different from Lu78 that for <u>arbitrary</u> augmented trees can insert or delete a record in O(wlog N) CERT-measured time

iii) an explanation and formal proof illustrating how the above algorithm can be improved to develop a procedure with a strict O(wlog N) <u>worst-case</u> time

A similar three-part mode of presentation will be used here. The subject matter has been divided so that topics i) and ii) are discussed in section 1, and topic iii) in sections 2 and 3.

All propositions will be given two theorem numbers in this paper. The first number indicates the chronological arrangement of the propositions, and the second indicates where the same proposition can be found in Wi78a.

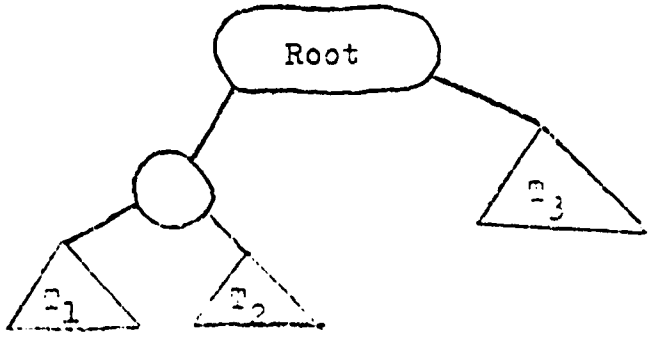
PART 1

The nature of the challenge faced in this paper can be understood if some traditional B-trees (AVL, 2-3, and multiway) are examined in detail. Theorem 1 demonstrates that these algorithms require O(wN) CERT runtime when they manipulate augmented trees. This CERT runtime indicates that these algorithms are substantially more difficult to optimize than was suggested by the simple O(wN) worst-case update time mentioned at the beginning of this paper.

The proof of Theorem 1 assumes that the reader is familiar with AVL, 2-3 and multiway trees, at least to the point of Kn-73 and AHU-74. Readers not familiar with these references may omit this proof and still understand most of the rest of this paper.

Theorem 1: Application of the AVL, 2-3, and multiway procedures to augmented trees will produce an algorithm with an O(wN) CERT runtime (Th 3.2.0).

Proof for AVL trees: A binary tree of height h will be said to have maximal size if it contains precisely 2^h leaves. Let T_1, T_2 and T_3 denote three such maximal-sized trees of height h. Let T denote that AVL tree which is shown in the diagram below:



Let c_1 c_2 c_3 c_4 denote a sequence of four data-modification commands such that

i) Command c_1 causes a leaf to be added to subtree T_1
ii) Command c_2 removes the cited leaf
iii) Command c_3 causes a leaf to be added to subtree T_3
iv) Command c_4 removes the cited leaf

It is easy to verify that the previous four commands will cause the AVL algorithm to move T_2 from the left side of tree T to the right side and then subsequently back to the initial position. Note that this cycle of four commands will require O(wN) time to manipulate the SDS fields. Furthermore, this cycle can be repeated any number of times. Each repetition will thus consume an additional O(wN) time. At least wN^2 runtime must therefore be consumed by a sequence of 5N commands that first builds the initial tree and then executes N repetitions of this cycle. Hence the AVL algorithm will have at least an O(wN) CERT runtime when it manipulates augmented trees. QED

Proof for the case of 2-3 trees: Consider a 2-3 tree where every ancestor of leaf L has 3 sons. Consider a sequence of two commands where

i) Command c_1 causes L to gain a new brother
ii) Command c_2 removes the cited record

Note that these two commands will force the 2-3 algorithm to consume at least O(wN) time. Also, note that the 2-3 tree will be in the same state after the execution of these two commands as it was before. Each repetition of a cycle of these two commands will thus consume an additional O(wN) time. Hence,

an O(wN) CERT runtime will follow by the same argument that was previously used for AVL trees.

QED

Proof for multi-way trees: Note that multi-way trees are the generalization of 2-3 trees for the case when each ancestor has between m and 2m-1 sons. The preceding proof for 2-3 trees can consequently be easily modified to show that multi-way trees also have at least an O(wN) CERT runtime. QED

This paper's super-B-tree algorithm will be a modified version of the Nievergelt-Reingold bounded balance algorithm. In our discussion of this topic, v_L will denote the left son of interior node v, v_R its right son, N_v the number of leaves that descend from v, N_{v_L} the number of descendants of v_L, and p(v) the ratio N_{v_L}/N_v. For fixed constant α, a tree will be said to satisfy the BB(α) "bounded balance" condition, if all its interior nodes v satisfy the inequality $\alpha < p(v) < 1 - \alpha$. Also, throughout this paper, it will be assumed that there exists a one-to-one correspondence between the leaves of our BB(α) tree and the list of elements they represent (as opposed to a correspondence between general nodes of the tree and the list).

The symbol Alg (α, k) will denote the B-tree procedure used in this section for inserting or deleting records in BB(α) trees. The α and k parameters of this procedure will be required to initially satisfy the following inequalities:

(1) $k > 2$

(2) $\alpha \leq 1 - \sqrt{1 - 1/k}$

Alg(α, k) will use a two-step procedure for performing insertion and deletion operations. The first step will modify the tree so that the user's specified record is either inserted into or deleted from the tree in the straightforward manner suggested by this command. The second step will scan the modified tree to determine whether any node's p(v) ratio was caused to exceed the (α, $1 - \alpha$) interval by the first step. For every node violating this range, the second step will utilize one of two "rebalancing" procedures to force p(v) back into the (α, $1 - \alpha$) interval. The more subtle of these procedures, called ROTATE(α, k), is defined as executing

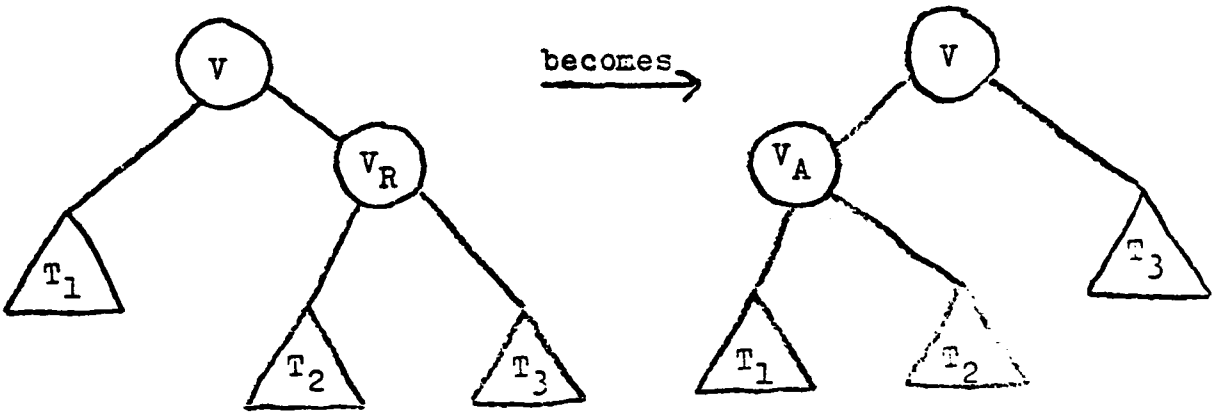
A) the single rotation of Diagram 1 when $p(v) < \alpha$ and $p(v_R) < \frac{1 - k\alpha}{1 - \alpha}$

B) the double rotation of Diagram 2 when $p(v) < \alpha$ and $p(v_R) \geq \frac{1 - k\alpha}{1 - \alpha}$

C) the mirror image of the preceding rotations when $p(v) > 1 - \alpha$

The above ROTATE(α, k) subroutine will be used by Alg(α, k) to rebalance all nodes satisfying $N_v > \alpha^{-2}$ (since under these circumstances ROTATE(α, k) necessarily forces the p-ratios of all affected nodes back into the (α, $1 - \alpha$) range). A different module will be utilized by Alg(α, k) to rebalance nodes whose $N_v < \alpha^{-2}$ because of their less stable nature. In this case, the trivial but inefficient brute force method will be applied

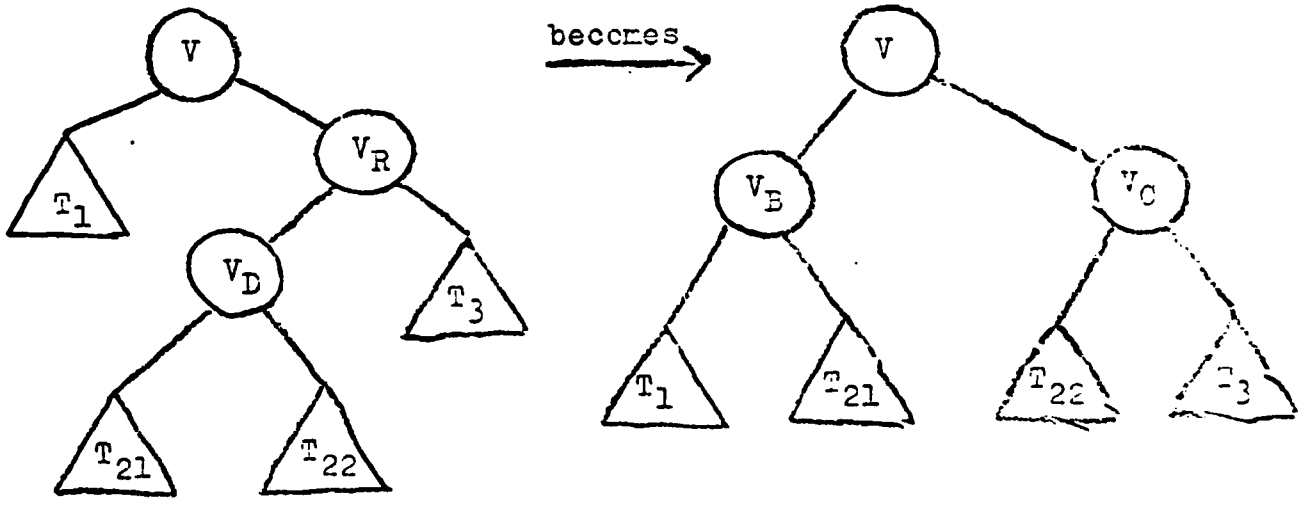
DIAGRAM 1



SINGLE ROTATION (R_1)

$$P(V) < \alpha \; ; \quad P(V_R) \leq \frac{1 - K\alpha}{1 - \alpha}$$

DIAGRAM 2



DOUBLE ROTATION (R_2)

$$P(V) < \alpha \; ; \quad P(V_R) > \frac{1 - K\alpha}{1 - \alpha}$$

to ensure that the entire v-rooted subtree has p-ratios belonging to the (α, 1 -α) range. The inefficiency of the brute force method, or any other details pertaining to it, are unimportant since its restricted application to nodes will small N_v values ensures that it cannot affect the runtime magnitude of Alg(α, k).

The Alg(α, k) procedure described above is a minor generalization of a similar algorithm that was originally proposed by Nievergelt and Reingold in NR73. Their algorithm was for the most part the special version of Alg(α, k) resulting when k = 2. We say "for the most part" because the NR73 algorithm contained a minor error of omission by not including the brute force module.*

The significance of Nievergelt and Reingold's work was that they demonstrated that Alg(α, k) possessed the standard O(log N) insertion, deletion and search times associated with B-trees: and that it ensures that no insertion or deletion can cause any initial BB(α) tree to violate this condition. It is trivial to generalize the NR-73 results for any α and k parameters satisfying equations 1 and 2 (only the runtime coefficient of the Alg(α, k) procedure changes as one varies its parameters within the permitted range).

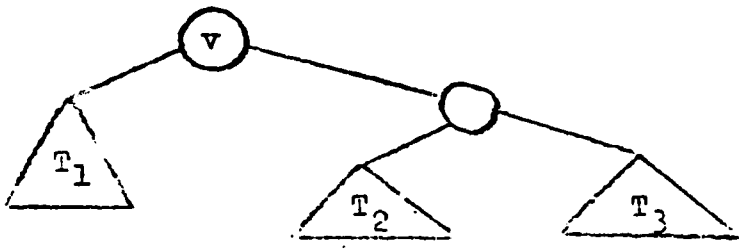
Two theorems will be proven about the Alg(α, k) procedure in this section. The first will indicate that the originally-proposed Alg(α, 2) procedure has an O(w$\sqrt{N}$) CERT runtime when applied to augmented trees, and the second that an O(w log N) CERT can be attained if k is set greater than 2.

---

* To understand why this module is necessary, calculate the value of $p(v_A)$ in Diagram 1 when N_v and α are small.

Theorem 2: Application of the Alg(α, 2) procedure to augmented trees produces an algorithm with an O(wN) CERT (Th 3.2.N).

Proof: Let T denote a BB(α) tree which contains the three subtrees T_1, T_2 and T_3 as shown in the diagram below:



Let us further assume that these three subtrees contain respectively αN, $(1 - 2\alpha)N$, and αN leaves. Let c_1 c_2 c_3 c_4 denote a sequence of four commands such that

1) Command c_1 inserts a leaf into the T_3 subtree
2) Command c_2 deletes the same leaf
3) Command c_3 inserts a leaf into the T_1 subtree
4) Command c_4 deletes the same leaf

It is easy to verify that the previous four commands will cause Alg(α, 2) to move the T_2 subtree from the right side of node v to its left side and then back to the initial position. Note that these movements will force Alg(α, 2) to consume O(wN) time (because the SDS structures must be adjusted whenever T_2 moves). Also note that the cycle of the previous four commands can be repeated any number of times. Each repetition will consume O(wN) additional time. Hence Alg(α, 2) will have an O(wN) CERT runtime for the same reasons that the AVL algorithm had this runtime. QED

The remainder of this section will be devoted to proving that the Alg(α, k) generalization of the bounded balance algorithm will manipulate augmented trees in O(wlog N) CERT runtime if k is chosen greater than 2. In this discussion, we will speak of a tree's collective depth. This will be defined as the sum of the depths of its leaves.

Lemma 3.1: Each application of a single or double rotation to a node v by the Alg(α, k) procedure will cause the collective leaf depth to decrease by at least $(k - 2)\ \alpha\ N_v$.

Proof: Similar methods are used to verify the theorem for the cases of the single and double rotations. It is therefore sufficient to consider the single rotation of Diagram 1.

The rules of the Alg(α, k) procedure imply that this rotation is executed only when

i) there are less than αN_v descendants in the T_1 subtree

ii) there are more than $(k - 1)\ \alpha N_v$ descendants in the T_3 subtree

Also it is apparent that the single rotation of Diagram 1 causes

iii) the depth of all records in the T_1 subtree to increase by 1

iv) the depth of all records in the T_3 subtree to decrease by 1

The collective implication of i) through iv) is that this rotation will produce the predicted loss of leaf depth.

QED

Lemma 3.2: If $k > 2$, then the amount of time that the Alg(α, k) procedure will need to make such adjustments in the SDS fields will always be proportional to the amount of leaf-depth lost during these rebalancing transformations. More specifically, the ratio of time spent adjusting the SDS fields over loss of leaf-depth (during rotations) will always be less than or equal to $\frac{w}{(k-2)\alpha}$ (Th 3.2.R).

Proof: Note that the Alg(α, k) procedure requires no more than wL_v runtime to adjust the SDS fields when it performs a rotation and that the previous lemma indicated a lower bound on the associated loss of leaf-depth. The present lemma is a direct consequence of these observations. QED

Theorem 3: If $k > 2$, then the Alg(α, k) procedure will have an O(wlog N) CERT runtime when it manipulates augmented trees (Th 3.2.S).

Proof: Note that the only part of the Alg(α, k) procedure which is capable of taking more than O(wlog N) time is its ROTATE(α, k) step. Also note that the previous lemma indicated that the runtime of this step is proportional to the loss of leaf-depth. The theorem will thus be proven if sufficient bounds are verified on the size of the collective leaf-depth of EB(α) trees. Such bounds follow from the observation that

if N denotes the maximum size of tree T during command sequence c, then the nonrotating aspects of these commands cannot increase the collective leaf-depth of T by more than $O(|C| \log N)$.

QED

Comment 1: It is interesting to re-examine Nievergelt and Reingold's original Alg(α, 2) procedure in light of the previous two propositions. Note that Theorem 3 holds only when $k > 2$. and it therefore did not preclude Alg(α, 2) from having an $O(w\overline{N})$ CERT runtime. In the context of augmented trees, Alg(α, k) is thus considerably more efficient than Alg(α, 2).

Comment 2: Also, it is interesting to compare Alg(α, k) to the slightly different procedure proposed in Lu78. For $P_0(k)$ pyramids, Lueker's algorithm will have an O(wlog N) CERT. However, it does not generalize easily to other types of augmented trees and must have an $O(w\log^2 N)$ CERT for certain types of trees.

PART 2

The next two sections of this paper will be devoted to explaining how the O(wlog N) CERT of Alg (α, k) can be converted into a strict worst-case runtime with a somewhat more sophisticated procedure called Alg (α, β, K, J). An intuitive description of Alg (α, β, K, J) will be given in this section with a more detailed discussion in the next section.

To understand the intuition behind this procedure, it should be recalled that the O(wlog N) CERT runtime of Alg (α, k) implies that every sequence C of insertion and deletion commands must consume less than $|C| \log N$ runtime (when applied to an initially empty tree). The desired O(wlog N) worst-case runtime will thus be achieved if the sequence is stabilized so that every one of its commands consumes no more than the sequence's approximate average runtime. Alg (α, β, K, J) will attain its O(wlog N) worst-case runtime by performing this optimization.

More specifically, Alg (α, β, K, J) will differ from Alg (α, k) by precluding the possibility that O(wN) worst-case runtime is spent adjusting the SDS fields when the user gives a command that causes a rotation. Instead, it will invoke an evolutionary process for gradually building the new SDS field while the user gives a large number of commands. The purpose of this method will be to insure the achievement of O(wlog N) worst-case runtime.

In our formal discussion, reference will be made to SDS fields which are "partially constructed" as opposed to "fully

constructed." By "partially constructed" we mean an SDS field that has not yet been completely built by the evolutionary process. An SDS field which is not partially constructed will be called "fully constructed." Also, the terms "current" and "anticipated" nodes will be used. The former refers to an interior node presently existing in an augmented tree, and the latter to a new node that a special module of the Alg (α, β, K, J) procedure "anticipates" will be inserted into this tree after a forthcoming single or double rotation. For instance, if Alg (α, β, K, J) anticipates the future execution of the double rotation shown in Diagram 2, then the nodes v_B and v_C would be said to be "anticipated." In general, the symbol ANTICIP(v, R) will denote those nodes that Alg (α, β, K, J) anticipates will be inserted after the application of a rotation R to a node v.

The Alg (α, β, K, J) procedure will be designed to insure that all its current interior nodes have fully constructed SDS fields (since it would otherwise make very little sense to employ augmented trees). Its anticipated SDS fields will typically be partially constructed. The subtle component of Alg (α, β, K, J) is the evolutionary process that <u>efficiently</u> converts the initial "partially constructed" SDS fields of "anticipated nodes" into the final "fully constructed" SDS fields of "current nodes." Much of the essence of this evolutionary process can be explained by examining the J parameter of Alg (α, β, K, J). For constant J whose value

will be defined later, Alg (α, β, K, J) will

i) anticipate the application of a rotation to node v at an early enough time to ensure that at least $3N_v/(J - 3)$ insertion and deletion commands are applied to the v-rooted subtree between the time of anticipation and the time when the rotation is performed

ii) gradually construct the SDS fields required by this rotation during this intervening period in a manner that

a) expends Jw runtime building each SDS field of ANTICIP(v, R) on every occasion when an insertion or deletion command is applied to one of v's descendants

b) ensures that all the SDS fields of ANTICIP(v, R) have become fully constructed before that time when Alg (α, β, K, J) applies rotation R to node v

Once again it should be repeated that the virtue of the above process is that it makes possible an <u>O(w log N) worst-case runtime</u> in the context of a tree whose <u>current SDS fields are always fully constructed</u>. A more formal description of Alg (α, β, K, J) will be given in the next section.

PART 3

In this section, SUP(v) will be defined as the Key-value of the current ancestor closest to v such that the left son of that ancestor is either v or another one of its ancestors. Also, INF(v) will be given the same definition in terms of right sons. These definitions easily imply that a leaf-record will be a descendant of v if and only if it satisfies $INF(v) \leq KEY < SUP(v)$. This inequality should be regarded as holding for both current and anticipated nodes (since it can be interpreted as discussing anticipated descendants in the latter case).

The advantage of this terminology is that it enables us to make much more explicit the last section's intuitive description of "partially constructed" and "fully constructed" SDS fields. Formal definitions can now be given to these concepts. SDS(v) will be defined to be fully constructed if it describes those records satisfying $INF(v) \leq KEY < SUP(v)$; and to be partially constructed if it describes those records satisfying $INF(v) \leq KEY < TEMP(v)$ for some number TEMP(v) which is less than SUP(v).

The evolutionary process used by Alg (α, β, K, J) for gradually enlarging the partially-constructed SDS fields was called the GCAS subroutine in Wi78a (the acronym stands for Gradual Construction of Anticipated SDS fields). This procedure requires an anticipated node v and an integer J as arguments. Upon its invocation, the GCAS will enlarge the partially constructed SDS(v) field by inserting J new records into this

data structure (and also obviously incrementing the value of TEMP(v) to reflect these insertions).

It is straightforward to develop an implementation of the GCAS procedure that requires O(Jw) runtime upon each invocation (because in this runtime the procedure can walk J steps down a linear list of records and expend w runtime inserting each encountered record into SDS(v)). A formal description of GCAS can be found in part 3.4.H of Wi78a. In the interests of brevity, this basically trivial subroutine will not be described here.

Another necessary concept in this section is that of an "untimely disruption." Anticipated node v* will be said to experience an untimely disruption if either SUP(v*) or INF(v*) undergoes a change of value.

Let us recall that N_v is used in this paper to denote the number of v's descendants and C to denote a sequence of insertion and deletion commands whose length is denoted as $|C|$. The symbols I_{cv}, M_{cv} and F_{cv} will respectively denote the initial, maximum and final values of N_v during C.

Lemma 4.1: Let v denote an interior node in tree T, v* an anticipated node belonging to ANTICIP(v, R), and C a sequence of commands that insert and delete records in the subtree of T which is rooted at v. Suppose that $|C|$ invocations of the GCAS procedure are made with the arguments of v* and J during the period of C's execution. SDS(v*) will become

fully constructed before the end of this period if v* experiences NO UNTIMELY DISRUPTIONS during this period and any one of the following three conditions does hold:

(3) $|C| > I_{cv} / (J - 1)$

(4) $|C| > N_{cv} / (J - 1)$

(5) $|C| > F_{cv} / (J - 1)$

Lemma 4.1 requires no proof, since it is an absolutely trivial consequence of the definitions of the GCAS procedure and of partially constructed SDS fields. Most of the rest of this section will show how the Alg (α, β, I, J) procedure attains its efficiency by insuring that the conditions of this lemma are consistently satisfied.

Another definition that will be needed in our discussion is that of a node v's being "affected" by a rotation. This shall be defined to mean that one of the following two conditions is satisfied:

i) the node lies at the root of the subtree which is directly manipulated by this rotation (the prototype example is v in Diagrams 1 and 2)

ii) the node is physically produced by the rotation (v_A, v_B and v_C in Diagrams 1 and 2 are examples)

Lemma 4.2: Let C denote a sequence of commands given to insert and delete records in the subtree that has v as a root. If no rotation affects v during the period in question, then

the p(v) ratio can change by no more than C / M_{cv} during this time (Th 3.6.D).

The proof of Lemma 4.2 has been omitted from this paper because it is trivial (it can be found in Wi78a). Our next topic is the ROTATE(α, k) subroutine. This subroutine was previously used by the Alg(α, k) procedure to determine whether a single as opposed to a double rotation should be applied to rebalance a node whose p(v) ratio exceeds the (α, 1 -α) range. In similar manner, it will be used by Alg(α, β, K, J) . Only the timing of rotations will differ in Alg(α, k) and Alg(α, β, K, J). Instead of automatically executing a rotation as soon as p(v) exceeds (α, 1 -α), Alg(α, β, K, J) will utilize a complicated timing mechanism which invokes ROTATE(α, k) at some time when p(v) has exceeded (α, 1 -α) but still lies within a broader (β, 1 -β) range. The following preliminary lemma will play a major role in our later, more detailed discussion of Alg(α, β, K, J).

Lemma 4.3: There exist coefficients Δ, α, β and k satisfying

$$\Delta > \alpha > \beta$$

such that

i) for every tree all of whose nodes have p-ratios belonging to the (β, 1 - β) range

ii) for every node v within this tree whose p(v) ratio exceeds the (α, 1 - α) interval

the application of the ROTATE(α, k) subroutine to the subtree rooted at v will cause all the affected nodes to obtain p-ratios belonging to the (Δ, 1 -Δ) interval (Th 3.6.A).

One way to verify the preceding lemma is to show that it is a consequence of more or less straightforward algebra. Although each step of this derivation is simple, the total computation is made complicated by the fact that it requires the verification of approximately three dozen equations. Such analysis is clearly too lengthy for one's intuition, and there does exist a simpler proof for those who understand point-set typology.

The latter proof has two parts. It begins with the observation that Lemma 4.3 can be algebraically verified if the lemma is changed to read: $p(v) = \alpha = \beta$. Subsequently, the proof uses typological arguments about continuous functions to show that Lemma 4.3 is a consequence of this observation.

In the interests of brevity, the details of this latter proof will not be given here. Readers interested in this subject should consult pp. 138-144 of Wi78a.

In our discussion of Alg(α, β, K, J), it will be assumed that the α, β, k and Δ parameters have been chosen to satisfy Lemma 4.3 and that J has been chosen to satisfy

(6) $$J \geq 3 + \frac{3}{\beta^2(\Delta - \alpha)}$$

(7) $$J \geq 3 + \frac{3}{\alpha - \beta}$$

Our formal description of Alg(α, β, K, J) begins in the next paragraph. This procedure will be carefully designed to ensure that all nodes have p-ratios consistently belonging to the (β, 1-β) range. It will utilize the GCAS subroutine for constructing anticipated SDS fields in the evolutionary manner previously described. If a node's p(v) ratio exceeds (α, 1 -α) but lies within (β, 1 -β), then Alg(α, β, K, J) will have the ROTATE(α, k) subroutine apply a rebalancing rotation R at the first moment in time when the SDS fields of ANTICIP (v, R) have reached fully constructed states. Should full construction not be reached before the time when p(v) exceeds the (β, 1 -β) interval, then Alg(α, β, K, J) will invoke an inefficient construction procedure that requires $O(wN_v \log N_v)$ runtime to reorganize the v-rooted subtree. The efficiency of Alg(α, β, K, J) will be shown to result from the fact that a proper choice of parameters ensures that the inefficient procedure described in the previous sentence will be executed only when N_v is bounded above by some small constant of K. This paragraph is intended only to provide a brief sketch of the Alg(α, β, K, J) procedure. A much more detailed discussion now follows.

FORMAL DESCRIPTION OF ALG(α, β, K, J): The procedure will take as arguments an augmented tree T and a command to either insert or delete a record y into T. It will be assumed that the four parameters of Alg(α, β, K, J) satisfy Lemma 4.3 and equations 6 and 7. Also, it will be presumed that there is a one-to-one correspondence between the leaves of T and the records

of the list it is representing (as opposed to a pairing between general nodes and records). Upon the user's insertion or deletion command, the following procedure will be executed:

1) First, insert or delete those nodes suggested by the user's command. More specifically, this means that:
   1a) If the user gave a deletion command, then both record y and its father should be deleted (the latter because an interior node has no purpose when it has only one son). Along with these changes, the relevant pointer in y's grandfather is adjusted so that it contains the address of y's brother rather than father.
   1b) If the user gave an insertion command, then perform the approximate inverse of the above operation.
2) Next, update all necessary SDS fields to reflect the preceding insertion or deletion of y. This means that
   2a) All (current) ancestors of y should have their SDS fields updated.
   2b) Also, for every such ancestor v, a check should be made to determine whether Alg(α, β, H, J) has raised a flag indicating the anticipation of a rotation on the subtree rooted at this node. If so, then the anticipated SDS fields of ANTICIP(v, R) should be updated when appropriate.
3) The last portion of Alg(α, β, H, J) will be designed to ensure that the p-ratios of all nodes belong to the (β, 1-β)

interval. This step is the only portion of Alg(α, β, E, J) which requires subtlety to attain efficiency. The following six substeps will be invoked once for each (current) ancestor v of y in <u>bottom-up</u> order:

3a) If $p(v) \leq \alpha$, then raise a flag indicating the anticipation of a leftward rotation through node v (if this flag was not previously raised). Similarly, lower this flag if $p(v) > \alpha$.

3b) In like manner, raise a right-rotation anticipation flag if $p(v) \geq 1 - \alpha$, and lower it otherwise.

3c) Let R_1 and R_2 denote the single and double left rotations illustrated in Diagrams 1 and 2. If the left-rotation flag is raised over v, then make one subroutine-call to GCAS for each anticipated node in ANTICIP(v, R_1) and ANTICIP(v, R_2), with the specification that their SDS fields should each be enlarged by J elements. (A total of three anticipated nodes are affected by these subroutine-calls: in the notation of Diagrams 1 and 2, they are V_A, V_B and V_C.)

3d) If the right-rotation flag has been raised, then make the similar subroutine-calls to GCAS for it.

3e) Let R denote that rotation which the ROTATE(α, R) subroutine indicated would rebalance node v. If p(v) has a value lying outside the (α, $1 - \alpha$) interval but still within the (β, $1 - \beta$) range,

and if all the SDS fields of ANTICIP(v, R) have reached fully constructed states, then execute this rotation.

3f) If p(v) attains a value outside the $(\beta, 1-\beta)$ interval, then invoke the trivial inefficient procedure that requires $wN_v \log N_v$ runtime to ensure that all members of the v-rooted subtree have p-ratios belonging to the (1/3, 2/3) interval.

Comment: One of the themes of the remainder of this section will be that step 3f's inefficiency is unimportant because a later theorem will show that it is executed only when N_v is less than some small constant of N. Before delving into this topic, we introduce some preliminary lemmas.

Lemma 4.4: A tree manipulated by the Alg(α, β, K, J) procedure must have a height of O(log N) magnitude (Th 3.6.B and 3.6.C).

Proof: In view of Lemma 4.3, it is apparent that steps 3e and 3f of Alg(α, β, K, J) ensure that the p-ratios of all nodes in T belong to the $(\beta, 1-\beta)$ interval. HR-73 has indicated that such ratios ensure an O(log N) height.

QED

Lemma 4.5: A single invocation of Alg(α, β, K, J) will never cause steps 1, 2, 3a through 3e to collectively consume more than O(wlog N) worst-case runtime (Th 3.4.I, 3.5.D and 3.5.F).

Proof: Given a tree of height h, it is easy to see that

1) Step 1 will consume no more than O(h) runtime (to locate that record which should be inserted or deleted).
2) Step 2 will likewise require no more than O(hw) runtime.
3) A single invocation of Alg(α, β, K, J) will cause steps 3a through 3e to be iterated no more than h times with each execution consuming no more than cw runtime for some coefficient c that is a function of α, β, K and J.

The preceding observations imply that steps 1 through 3e will collectively consume no more than O(wh) runtime. In view of Lemma 4.4's derivation of h, this establishes our O(wlog N) worst-case runtime. QED

The final goal of this section will be to prove that step 3f also executes efficiently. That result, in conjunction with Lemma 4.5, will show that Alg(α, β, K, J) has an O(wlog N) worst-case runtime. Our discussion of this topic must be prefaced with four preliminary lemmas.

Lemma 4.6: Let C denote a sequence of insertion and deletion commands that are applied to the portion of tree T that is a descendant of root v. Suppose that either $p(v) \leq \alpha$ or $p(v) \geq 1 - \alpha$ during this sequence. If the length of C satisfies

(8) $$|C| \leq \lceil \beta^2(\Delta - \alpha) N_{cv} \rceil$$

then no more than two untimely disruptions will be experienced by the ANCIOP(v, R) nodes (Th 3.6.F).

Proof: It is sufficient to prove the lemma only for the case where $p(v) < \alpha$ interval (since the other case will follow from symmetry arguments). The proof for this case will be based on the following three observations:

1) Let v_R denote v's right son and v_D denote the left son of v_R (as was previously shown in rotation diagrams 1 and 2). It is easy to see that untimely disruptions in this case are possible only if steps 3e or 3f apply a rotation to the subtrees rooted in either v_R or v_D.
2) Let s denote the node of v_R or v_D. An examination of Lemma 4.3, together with steps 3e and 3f of Alg(α,β, [illegible]), reveals that such rotations will be made only when p(s) lies outside the (α, 1 -α) interval, and that they will cause p(s) to move inside the (Δ, 1 -Δ) range.
3) Lemma 4.2 and equation 8 easily imply that if either $p(v_R)$ or $p(v_D)$ lie inside the (Δ, 1 - Δ) interval (at the end of any single command in sequence 8), then this value will remain inside the (α, 1 - α) range for the rest of this command sequence.

It is fairly easy to show that the preceding three observations imply that ANTICIP(v, R) will be caused to have no more than one untimely disruption from node v_R and no more than one untimely disruption from v_D. Thus no more than two untimely disruptions will occur. QED

Lemma 4.7: Let C denote a sequence of insertion or deletion commands that are applied to a subtree within T which is rooted at v. If

(9) $$|C| \geq \left\lceil \frac{3F_{cv}}{J-3} \right\rceil$$

then it is impossible for the value of p(v) to remain in either the (β, α) or $(1-\alpha, 1-\beta)$ intervals during the ENTIRE period in which sequence C is executed (Th 3.6.G).

Proof: The lemma will be verified by means of contradiction. Our goal will thus be to show that a contradiction will arise if it is assumed that p(v) will remain in either the (β, α) or $(1-\alpha, 1-\beta)$ interval during the entire execution of sequence C.

Let C* denote the sequence of the last $\left\lceil \frac{3F_{cv}}{J-3} \right\rceil$ commands that appear in sequence C. The desired contradiction will be obtained if we examine this sequence.

The significant characteristic of sequence C* is that equation 6 implies that its length must satisfy the inequality of

(10) $$|C^*| \leq \left\lceil \beta^2(\Delta - \alpha) F_{cv} \right\rceil$$

The above equation, together with the hypothesis of Lemma 4.7, imply that Lemma 4.6 is applicable to sequence C*. That lemma indicates this sequence can contain no more than two untimely disruptions. It thus follows that sequence C* will contain a subsequence of $\left\lceil \frac{F_{cv}}{J-3} \right\rceil$ consecutive commands that contain no

untimely disruptions. In view of Lemma 4.1, it is apparent that ANTICIP(v, R) will become fully constructed as a result of the repeated invocations of GCAS that are made by steps 3c and 3d during this period.* This latter fact is important because step 3e of Alg(α, β, K, J) is designed to apply rotation R to node v as soon as the ANTICIP(v, R) SDS fields reach states of full construction. In view of Lemma 4.3, this rotation will move p(v) into the (Δ, 1 - Δ) interval, which is a subset of (α, 1 - α). Hence, the desired contradiction has been reached, since it was impossible to keep p(v) outside (α, 1 - α).

QED

Lemma 4.8: If step 3f of Alg (α, β, K, J) applies a rotation to node v, then at the time of this rotation N_v must satisfy the following equation (Th 3.6.H):

$$(11) \qquad \left\lceil \frac{3N_v + 3}{J - 3} \right\rceil > \lceil (\alpha - \beta) N_v \rceil - 1$$

Proof: Note that Alg(α, β, K, J) will apply step 3f to node v only if this node's p(v) value lies outside the (β, 1 - β) range. Also, note that Lemma 4.2 implies that the period between the last time when p(v) retained a value inside the (α, 1 - α) interval and the moment when it moves outside the (β, 1 - β)

---

* Some especially attentive readers may wish to understand why slightly different denominators of J-1 and J-3 appeared in Lemma 4.1 and the present lemma. The answer is that the lemmas discuss different sequences of command with the 4.1 sequence contained within the present sequence. Consequently N_v has slightly different meanings in these two lemmas, and the denominator must be adjusted to make Lemma 4.1 applicable to Lemma 4.7.

interval must include a minimum of $\lceil(\alpha-\beta)\,N_v\rceil$ insertion and deletion commands which are applied to v's set of leaf descendants. This fact, in other words, means that there must be a minimum of $\lceil(\alpha-\beta)\,N_v\rceil - 1$ commands that are applied to v's subtree during the interim period when p(v) lies in either the (β, α) or $(1-\alpha, 1-\beta)$ intervals. The proof of the present theorem can be completed if we observe that Lemma 4.7 implies that this same set of insertion and deletion commands is forbidden to contain more than $\left\lceil \frac{3N_v + 3}{J - 3} \right\rceil$ instructions. The observations in the preceding two sentences imply that equation 11 must hold.

QED

Lemma 4.9: Each invocation of step 3f of Alg(α, β, K, J) will consume no more than O(w) runtime.

Proof: Equation 7 implies that equation 11 can be satisfied only for integers N_v that are less than some fixed constant of K. In view of Lemma 4.8, this bound implies that step 3f is applied only to nodes that have less than K descendants. Hence step 3f must consume less than K log Kw runtime for some fixed constant K. QED

Theorem 4: The Alg (α, β, K, J) super-B-tree procedure will possess an O(w log N) worst-case insertion and deletion and deletion runtime.

Proof: An immediate consequence of Lemmas 4.4, 4.5 and 4.9.

QED

Remark 4.1: An augmented tree will be said to satisfy the separability condition if all pointers within each SDS field necessarily contain the addresses of other data items in the same SDS field. All the augmented trees in BS77, Lu78, LW78b, Wi78a, and part 2 of Wi78c were separable, and the super-B-tree theorem thus guarantees their O(wlog N) worst-case update time. Technically, the theorem does not apply to inseparable augmented trees, such as the $P_d(2)$ pyramid of Wi78a or the $P_h(2)$ pyramid of Wi78d (which lock together the SDS fields of father and son nodes by having records in one field point to their counterparts in the other). Nevertheless, it is absolutely trivial to modify the super-B-tree algorithm so that it also applies to these latter types of pyramids. A detailed discussion of this revision has been omitted from this paper because it raises no new interesting theoretical issues.

Remark 4.2: A detailed examination of Alg(α, β, K, J) indicates that it has an efficient CERT runtime coefficient but an inefficient worst-case coefficient. Both coefficients can be markedly improved with the use of several additional modules. One especially attractive module is a procedure that treats step 3 of Alg (α, β, K, J) as a background process whose node rebalancing operations are typically deferred to

times when the computer would otherwise be idle. There are also many other important coefficient optimization techniques. These were not discussed here because they were deemed unsuitable for an introductory article.

Remark 4.3: There is an interesting modified version of the super-B-tree algorithm that may be useful in certain applications. This variation will be called the multipath super-B-tree. It will assign each interior node a multiple number of children, rather than two. The N_v value of a node v at a depth d in such a tree with N descendents will be required to satisfy

(12) $$\alpha^{-1} n^{-d} N < N_v < \alpha n^{-d} N$$

for some appropriately chosen constants of n and α. Properly defined node splitting and merging rules will enable the multipath algorithm to have O(wlog N) worst-case or OERT insertion and deletion runtime (the same is not true when the multiway algorithm of Mn73 is applied to augmented trees). Comparisons between multipath and binary super-B-trees reveal that the former has an improved memory space, insertion and deletion coefficients while the latter shows improvement in the retrieval time coefficient.

CONCLUSION

The fundamental concept which motivated much of this research was that the bounded balance method (introduced in NR73) is preferable to other B-tree methods because the value of $p(v)$ changes very slowly during a sequence of insertion and deletion commands. Alg(α, β, E, J) utilized this stable behavior to successfully anticipate its future rotations at a sufficiently advanced time to ensure an O(wlog N) worst-case runtime for augmented trees.

The reason for our interest in augmented trees is of course that BS77, Lw78b, Wi78a and Wi78c have shown that they improve retrieval time. The super-B-tree algorithm thus appears to optimize insertion and deletion runtime for a data structure that is attracting increasing interest.

REFERENCES

AVL-62 G. M. Adel'son-Vel'skii and E. M. Landis. "An Algorithm for the Organization of Information," Sov. Math. Dokl., 3 (1962), pp. 1259-1262.

AHU-74 Alfred Aho, John Hopcroft, and Jeffrey Ullman, The Design and Analysis of Computer Algorithms. Addison-Wesley, Reading, Mass., 1974.

Be-75 Jon L. Bentley, "Multidimensional Binary Search Trees Used for Associative Searching, CACM, 18:9(1975), pp. 509-517.

BF-78 Jon Bentley and Jerome Friedman, "Algorithms and Data Structures for Range Searching," Proceedings of the Computer Science and Statistics 11th Annual Symposium on the Interface (March 1978), pp. 297-307.

BS-75 Jon L. Bentley and Donald F. Stanat, "Analysis of Range Searches in Quad Trees," Inf. Proc. Letters. 3:6(1975), pp. 170-173.

BS-76 Jon L. Bentley and Michael I. Shamos, "Divide-and-Conquer in Multidimensional Space," Proc. 8th Annual ACM Symposium on Theory of Computing. 1976, pp. 220-230.

BS-77 Jon Bentley and Michael Shamos. "A Problem in Multi-Variate Statistics: Algorithm, Data Structure, and Applications," Proceedings of the 15th Allerton Conference on Communications, Control, and Computing (Sept. 1977), pp. 193-201.

DL-76 David Dobkin and Richard Lipton, "Multidimensional Searching Problems," SIAM J. Comput., 5:2(1976), pp. 181-186.

FB-74 R. A. Finkel and J. L. Bentley, "Quad Trees: A Data Structure for Retrieval on Composite Keys," Acta Inf., 4(1974), pp. 1-9.

Fr-78 W. R. Franklin, "Locating a Point in Overlapping Regions of Hyperspace," Technical Report CRL-64 (1978) of R.P.I., Troy, New York.

K-73 Donald Knuth. The Art of Computer Programming, Vol. 3: Sorting and Searching. Addison-Wesley, Reading, Mass., 1973.

Lu-78 George Lueker, "A Data Structure for Orthogonal Queries." 19th Symposium on Foundation of Computer Science (1978). pp. 28-34.

Lum-70 V. Y. Lum, "Multi-Attribute Retrieval with Combined Indexes," CACM (Nov. 1970), pp. 660-665.

LW-78a D. T. Lee and C. K. Wong, "Worst-Case Analysis for Region and Partial Region Searches in Multidimensional Binary Search Trees and Balanced Quad Trees," Acta. Inf., 9(1977). pp. 23-29.

LW-78b D. T. Lee and C. K. Wong, "Quintary Tree: A File Structure for Multi-Dimensional Database System," IBM Watson Research Report RC 7285 (August 28, 1978), #31364.

NR-73 J. Nievergelt and E. M. Reingold, "Binary Search Trees of Bounded Balance," SIAM J. Comput., 2:1(1973). pp. 33-43.

Ri-76 Ronald Rivest, "Partial Match Retrieval Algorithms," SIAM J. Comput., 5:1(1976), pp. 19-50.

Wi-78a Dan E. Willard, Predicate-Oriented Database Search Algorithms. Ph.D. thesis for Harvard Mathematics Department; formally submitted May 3, 1978; accepted Sept. 29, 1978; published as an official Harvard Aiken Computer Lab Report TR-20-78; also disseminated as one of twenty volumes in Garland Publishing Company's series of 20 "Outstanding Dissertations in Computer Science."

Wi-78b Dan E. Willard, "Balanced Forests of K-d* Trees as a Dynamic Data Structure"; submitted to CACM; also issued as a Harvard Aiken Computer Lab Report TR-23-78.

Wi-78c Dan E. Willard, "New Data Structures for Orthogonal Queries." submitted to CACM; also issued as a Harvard Aiken Computer Lab Report TR-22-78.

Wi-78d Dan E. Willard, "Data Pyramids," a forthcoming article.