UPDATING BINARY TREES

By

Joseph Carl Culberson

Research Report CS-84-08

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

March 1984

Updating Binary Trees

by

Joseph Carl Culberson

A thesis

presented to the University of Waterloo

in partial fulfillment of the

requirements for the degree of

Master of Mathematics

in

Computer Science

Waterloo, Ontario, 1983

Abstract

Updating Binary Trees

It has been widely believed that deletions followed by insertions caused the internal path length of binary trees to be reduced on average. Recent empirical results obtained by Eppinger [E] have cast doubt upon this assumption. This thesis adds to the empirical evidence through additional simulations, and investigates some plausible explanations. A few hypotheses are formed and simulations performed to test them. In addition, a brief exploration is made of a special case in which the number of available elements is the same as the number of nodes in the tree.

Acknowledgements

I would like to thank my supervisor Professor Ian Munro for suggesting this topic and for his advice during the developement of this thesis. I would also thank Professor P. A. Larson and Professor G. V. Cormack for taking time to read this thesis.

Table of Contents

UPDATING BINARY TREES

Joseph Culberson

University of Waterloo

1. Introduction

Binary search trees are well known data structures which have been studied for many years. Knuth [K1] gives the following definition of a binary tree.

> A *binary tree* is a finite set of nodes which is either empty, or consists of a root and two disjoint binary trees called the left and right subtrees of the root.

A *binary search tree* on a set D, the *domain* of the tree, is a binary tree which has associated with each node an element $key(v) \epsilon D$, the *label* or *key* of the node, such that each key in the left subtree is less than the key of the node, and each key in the right subtree is greater than the key of the node. The roots of the left and right subtrees are generally referred to as the left and right sons of the node, and the node is the father of its sons. The primary root, usually referred to as the root, is the node which has no father. Any node in which both the left and right subtrees are empty is called a leaf.

The *successor* of any node is the node in its right subtree which has the smallest key. The left subtree of the successor must necessarily be empty. If the right subtree of the node is empty, then the successor is null. The *predecessor* of any node, defined analogous to the successor, is the node in the left subtree with the largest key.

The updating problem consists of two types of transformations on a binary search tree; insertion and deletion. Given an element x from D, inserting x into a binary tree consists of creating a node, assigning x as the key of the node, and adding the node to the tree so that

the binary search invariant is maintained. By far the most natural insertion algorithm is the one recursively stated as

```
INSERT(x, v)
   if v is empty, add node with key(node)=x.
   else if key(v)<x INSERT(x, r(v));
   else INSERT(x, l(v))
```

where $l(v)$ is the left subtree of v, and $r(v)$ is the right subtree. The algorithm is called initially on the root of the search tree.

The problem of deleting a key from the tree is more complex. We cannot just delete the node associated with the key, since this would also remove the left and right subtrees. The tree has to be re-connected with the subtrees of the deleted node in a way which preserves the search invariant. Unlike the insertion problem, there are many algorithms which can be used for deletions. Several algorithms are discussed in more detail in the next chapter.

Hibbard [H] was one of the first to publish a deletion algorithm. In the Hibbard algorithm, the successor is used to replace the deleted element; that is, it is logically moved so that the father, left son and right son of the deleted element become the father, left son and right son of the successor, respectively. The previous right son of the successor becomes the corresponding son of the successor's old father. (In the degenerate case, the successor retains its right son.) If the right subtree of the deleted element should happen to be empty, that is, there is no successor, then the father of the deleted element becomes the father of the left son.

Knuth [K3] gives an improved algorithm, in which, if the deleted element has no left son, then the right son becomes the son of its grandfather; otherwise, it is the same as the Hibbard algorithm. Why is this an improved algorithm? To answer this, we first need to define some measure of the goodness of a tree. The cost of a binary search tree is the average number of comparisons required in any successful search for a key in the tree. The search algorithm is similar to the insertion algorithm, in that we start at the primary root of the tree and proceed left and right down the tree until either the key is found, or we reach an

empty tree, which indicates the key is not present and the search has failed. Knuth shows ([K3]) that the cost of a tree resulting from a Knuth deletion is always less than or equal to the cost of the tree resulting from a Hibbard deletion of the same element.

The general updating problem is to find a deletion algorithm which will give the tree of least cost for every sequence of insertions and deletions beginning on some initial tree. As Knot points out [K], although the Knuth algorithm is better for any given deletion, it does not follow that it is always better for every sequence of insertions and deletions. As stated here, the general problem is probably ill-posed.

A better statement of the problem is to consider the average cost of trees resulting from various sequences of insertions and deletions for each algorithm. To make the average meaningful, the resulting trees should all be of the same size. The specific update problem considered here is the behavior of binary trees of size N when each deletion is followed by an insertion, and each such pair constitutes one update. Each deletion selects a node for deletion with uniform probability of $\frac{1}{N}$. Each insertion selects a key from D with uniform probability. The average behavior of the cost of the tree over a long sequence of such updates is considered.

The trees are created by issuing a sequence of N insertions. This results in a probability distribution of shapes of trees, $\pi_0(N)$, called a random distribution of binary trees. This is the distribution of shapes of trees which occurs if each of the $N!$ permutations of the keys is equally likely. Knuth [K3] gives the average cost of such trees as

$$C_N = 2\left(1+\frac{1}{N}\right)H_N - 3$$

where H_N is the harmonic number.

Hibbard proved in 1962 that building random trees of size $N+1$, and then randomly deleting a node using the Hibbard algorithm, resulted in trees with distribution $\pi_0(N)$. Knot extended this result in 1972 [K], to show that if $N+K$ items were initially inserted, then followed by K deletions of the first K items inserted in the same order as they were inserted,

then the resulting trees would still be random.

However, Knot also showed that following the deletions by further insertions destroyed the randomness property. Empirical results seemed to show that updates using either the Hibbard or Knuth deletion algorithms resulted in trees with lower average cost than random trees. Knot's conjecture was that deletions would not make the cost greater than pure insertions would. In 1978, Jonassen and Knuth [JK] proved, using rather heavy mathematics, that for trees of size three the average cost would be reduced under the updating process for either algorithm. Thus, Knot's conjecture was proved for trees of size three.

In 1983 Eppinger [E] published results of extensive simulations showing that for trees larger than 128 nodes, the cost of the trees increased if sufficiently many updates were performed using the Hibbard algorithm. Thus, Knot's conjecture seems to be disproved for trees larger than about 128 nodes. However, modifying the algorithm so that the replacement node was randomly chosen to be either the successor or the predecessor resulted in trees with reduced cost.

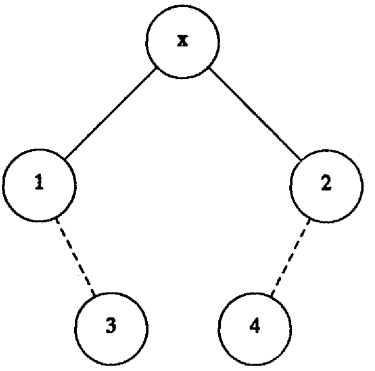
Determining the limiting distribution of trees and thus the limiting average cost, for large trees seems to be very difficult. No theoretical proof or disproof of Knot's conjecture has as yet been obtained. In this examination of the problem, it is my intention to present some rudimentary explanations of the behavior of trees, making some intuitive assumptions. To test the assumptions, various deletion algorithms were devised and simulations performed using these algorithms. In addition, a special case in which the domain is finite and of cardinality N is cursorily examined in hopes that it might shed light on the more general problem. This special case is easily solved for specific small trees under various updating algorithms.

In chapter 2, the various algorithms that are considered are outlined. In chapter 3, a brief examination of the special case problem is given. Intuitive notions of why trees behave as they do under various algorithms are presented in chapter 4. In addition, various measures for determining the average shape of trees are developed. A summary of the results of the simulations is presented in chapter 5, together with some interpretations.

2. Replacement Policies

Although there is only one basic insertion algorithm, there are many possible deletion algorithms for binary trees. In large part, the deletion algorithms can be distinguished by the replacement policy they use; that is, the policy used to select the node which is to replace the deleted node when it is not a leaf.

To describe the various algorithms in this thesis, the replacement policies used will be described in terms of the following diagram.



Replacement Labeling Diagram

Here, "x" is the node to be deleted. Nodes 1 and 2 are the left and right sons of x, and nodes 3 and 4 are the predecessor and successor of x. The dashed lines indicate that there may be several nodes in the intervening interval. Of course, not all these nodes need exist in a given situation. In particular, if the left son of a node is its predecessor, then 3 and 1 will refer to the same node. Similarly, if the right son is the successor, then nodes 2 and 4 are the same.

To keep the tree connected when x is deleted and x is not a leaf, the nodes adjacent to x must be re-connected in some manner which does not change the search property of the tree. This is usually accomplished by choosing some node to replace the deleted node. The replacement node becomes the son of the deleted element's father, and its children must be such that the connectivity and the search property of the tree remain intact.

The replacement node can only be chosen from the nodes 1, 2, 3, and 4, unless drastic restructuring of the tree is to be allowed. There are further constraints upon which of these are allowable choices. For example, if both 1 and 2 are non-null, then we can only choose 2 as the replacement if it does not have a left son. Otherwise, the order of the keys represented by the tree would not be preserved. Also, if 4 is chosen, then the right son of 4 becomes the left son of the father of 4, unless we wish to recursively find a replacement node for 4. Symmetric arguments apply to the choices 1 and 3.

Deletion algorithms are defined by the way in which the tree is re-connected, which for the algorithms being considered here, can be defined in terms of a replacement policy. Thus, the Hibbard algorithm can be expressed as "the first choice for a replacement node is 4 and the second choice is 1." Remember that 4 and 2 may be the same node. Similarly, the Knuth algorithm can be expressed as "choose 1 if 2 is null, else choose 2 if 1 is null, else choose 4." Note that this algorithm does not make 1 its first choice in all cases, since if the right son of 1 is null, 1 could be chosen even if 2 did exist. In this example, it is easy to see that the treatment of 1 and 2 are symmetric, and thus the Knuth algorithm can be expressed equivalently as "choose 2 if 1 is null, else choose 1 if 2 is null, else choose 4."

In both the Hibbard and Knuth algorithms, there is an asymmetry in the choice of the replacement nodes, in that the replacement node is more likely to be chosen from the right subtree rooted at x, than from the left. I will refer to this asymmetry as right-asymmetry. It turns out that asymmetry in deletion is very important to the behavior of trees under the updating process. If in any description of a deletion algorithm, 1 and 2 are swapped throughout, and 3 and 4 are also swapped, then the new algorithm will be symmetric to the original. I will refer to these as mirror images of one another. Thus, a symmetric algorithm can be obtained from any asymmetric one by alternating between the mirror image versions of it.

For purposes of this study, I chose the following five algorithms. The description is in terms of the replacement policy.

[1] First choice is 4, and second choice is 3; that is, choose the successor or else choose the predecessor.

[2] Randomly choose between algorithm 1 and its mirror image, choosing either with equal probability. This is then a symmetric algorithm.

[3] The Knuth algorithm. Choose 1 if 2 is null, else choose 2 if 1 is null, else choose 4.

[4] The Knuth algorithm applied symmetrically by choosing between mirror image versions of algorithm 3 at random.

[5] This algorithm uses the same replacement policy as algorithm 4. However, upon each insertion a small amount of rebalancing may be done, and similarly following each deletion there may be some rebalancing. The rebalancing during insertion ensures that if a node has only one son, then the son is a leaf. Following a deletion, if the father of the replacement node (or the father of the deleted node if it is a leaf) has now only one son, then it is deleted and re-inserted into the subtree rooted at its son.

The reasons for selecting these particular algorithms for study will be expanded upon in later chapters, but it seemed wise to list the algorithms here for future reference.

3. A Special Case - Exact Fit Domain

As mentioned in the introduction, the general problem of solving for the limiting distribution of trees under the updating process using any given deletion algorithm seems to be very difficult. However, if we assume that the domain from which the elements are chosen for insertion into the tree is finite and of small order, then the problem is somewhat simplified. In particular, solutions over domains which have the same number of elements as nodes in the tree can be found easily for small trees.

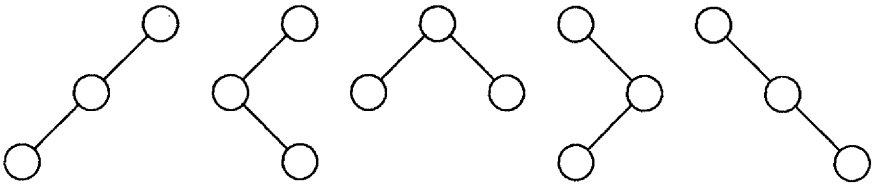
In an exact fit domain, each update will consist of a deletion followed by the reinsertion of the deleted element. Since there is only one possible assignment of the keys to the nodes of any tree, this implies that each update which deletes a given node from a particular tree, transforms that tree to a specific (possibly the same) new tree. If T_N is the set of binary trees having N nodes, then there will be exactly N such transformations on each tree in T_N; but these do not necessarily result in N distinct trees. If the probability of updating on each key is the same, then each of these N transformations has the same probability. The updating process can then be modeled by a Markov chain, where the steady state distribution can be found by solving a set of linear equations.

The internal path length of a tree is the sum over all nodes of a tree of the depth of the node. The expected internal path length for a distribution d is given by

$$E(I_d) = \sum_{t \in T_N} \pi_{dt} I_t \tag{3.1}$$

where π_{dt} is the probability of t in the distribution d, and I_t is the internal path length of t. The expected path length for the steady state distribution will be denoted by $E(I_\infty)$.

As an example, consider the set T_3, of trees of size three. The keys will be chosen from the domain D = {1,2,3}. The five possible trees are shown in the following diagram.



A B C D E

Binary Trees of Size Three

The probability distribution of the trees after d updates will be represented by $\pi_d = \left[\pi_{dA}, \pi_{dB}, \pi_{dC}, \pi_{dD}, \pi_{dE}\right]$. Assuming that the trees are initially generated by inserting the elements of D in random order, then following Knuth [K3], the initial distribution of the trees [A,B,C,D,E] is easily seen to be $\pi_0 = \left[\frac{1}{6}, \frac{1}{6}, \frac{2}{6}, \frac{1}{6}, \frac{1}{6}\right]$. The expected internal path length for this initial distribution is $E(I_0) = \frac{16}{6} = 2.667$.

To find the set of transformations, we must first select some deletion algorithm. Algorithm 1 described in chapter 2 will be used for this example. The transformation probabilities are most easily enumerated in a matrix.

	A	B	C	D	E
A	$\frac{1}{3}$	$\frac{1}{3}$	0	0	0
B	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$	0	0
C	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{2}{3}$	$\frac{1}{3}$	$\frac{1}{3}$
D	0	0	0	$\frac{1}{3}$	$\frac{1}{3}$
E	0	0	0	$\frac{1}{3}$	$\frac{1}{3}$

The matrix is generated column by column. The interpretation of an entry M_{ij} is that the tree of column j has probability M_{ij} of being transformed to the tree of row i. For example, consider the column headed B. Since there are three nodes in the tree, there are three possible transformations, each of probability $\frac{1}{3}$.

If the bottom node of tree B is deleted and re-inserted, the tree B is again obtained. If

the middle node of B is deleted, algorithm 1 specifies that the bottom node is brought up as a replacement. Upon re-insertion of the deleted element, tree A is obtained. Finally, if the root of B is deleted, then algorithm 1 specifies that the predecessor must be used as a replacement, and this means the bottom node becomes the new root. Upon re-insertion, tree C is obtained. These are the only transformations possible on B, and so the transformation probabilities are those given in column B of the matrix.

The matrix can now be interpreted as the transition matrix of a Markov process. If π_d is the current distribution, then the probability distribution after one more update is $\pi_{d+1} = M\pi_d$, where M is the transformation matrix. The steady state assumption says that $\lim_{d\to\infty}\pi_d = \pi_{d+1} = \pi_\infty$. On substituting into the previous equation, we get a set of linear equations in π_∞. However, these equations do not provide a unique solution for π_∞. We need to replace one of the above equations with another specifying that the sum of the probabilities is one. Upon replacing the first equation, the steady state distribution is solved as $\pi_\infty = \left[\frac{1}{6}, \frac{2}{6}, \frac{3}{6}, 0, 0\right]$. The steady state internal path length is $E_1(I_\infty) = \frac{15}{6} = 2.5$.

If instead of algorithm 1, the Hibbard algorithm is used, the only change in the transformation matrix M occurs in column B. When the root of B is deleted using the Hibbard algorithm, the middle node is used as the replacement node. Upon re-insertion of the deleted element, the tree obtained is E, where in the previous case the tree C was obtained. For trees of three nodes, this is the only transformation affected. The resulting steady state distribution is $\pi_\infty = \left[\frac{3}{24}, \frac{6}{24}, \frac{9}{24}, \frac{2}{24}, \frac{4}{24}\right]$. The internal path length is $E_H(I_\infty) = \frac{21}{8} = 2.625$.

The values for algorithm 3, the Knuth algorithm, can be similarly obtained. For algorithm 3, $\pi_\infty = \left[\frac{2}{12}, \frac{3}{12}, \frac{4}{12}, \frac{1}{12}, \frac{2}{12}\right]$ and $E_3(I_\infty) = \frac{16}{6} = 2.667$.

Symmetric algorithms can be analyzed similarly. The transformation matrix is just the

element by element average of the two matrices generated for the mirror image algorithms involved. For the symmetric Hibbard algorithm, $\pi_\infty = \left[\frac{3}{24}, \frac{4}{24}, \frac{10}{24}, \frac{4}{24}, \frac{3}{24}\right]$ and the internal path length is $E_{HS}(I_\infty) = \frac{31}{12} = 2.5833$. For algorithm 2, the results are $\pi_\infty = \left[\frac{1}{12}, \frac{2}{12}, \frac{6}{12}, \frac{2}{12}, \frac{1}{12}\right]$ with internal path length $E_2(I_\infty) = \frac{5}{2} = 2.5$. Algorithm 4, the symmetric Knuth algorithm, results in $\pi_\infty = \left[\frac{1}{6}, \frac{1}{6}, \frac{2}{6}, \frac{1}{6}, \frac{1}{6}\right]$ and internal path length of $E_4(I_\infty) = \frac{16}{6} = 2.667$.

On the basis of these results, it might be thought that algorithm 1 or 2 is the best deletion algorithm, but as in the case of the general tree of three nodes analyzed in [JK], these results are entirely misleading when extrapolated to larger trees. In the 3-trees, any redistribution which increases the probability of C decreases the average internal path length. The noteworthy point is that each of the right asymmetric algorithms decreases the probability of the right most trees and increases the probability of those to the left. For larger trees, some of the best balanced trees are also reduced in probability, but this means that the internal path length is then increased on average.

For example, consider algorithm 1. For this algorithm, it is easily seen that no tree which has fewer than 2 nodes in its right subtree can be transformed through any sequence of updates to a tree with 2 or more nodes in the right subtree. This implies that all such trees must have probability zero of occurring in the steady state. For trees of size six or larger, all the trees with minimum internal path length have 2 or more nodes in the right subtree. These are also the trees with the highest probabilities in the initial distribution. Thus, upon removing these in the steady state, it is unlikely that the internal path length will be improved.

For algorithm 1 on trees of size 4, $E_1(I_\infty) = \frac{799}{168} = 4.756$ while the initial internal path length is $E(I_0) = \frac{29}{6} = 4.833$. Thus, even here algorithm 1 tends to reduce the internal

path length very little. For trees of size five, the initial internal path length is $E(I_0) = 7.4$, while the steady state internal path length is $E_1(I_\infty) = 7.488$. Thus for trees of size five, algorithm 1 yields trees which are worse than the initial trees on average.

Not all the news is bad, however. For each of the algorithms except algorithm 5, deleting a leaf and re-inserting it is the identity transformation. Thus, linear trees, which have only one leaf, tend to have a high probability of transforming to another tree. Balanced trees with many leaves tend to be more miserly. So, linear trees may have reduced probabilities in the final distribution. A general theory relating these effects would definitely be an asset here.

The number of different trees increases rapidly with the number of nodes, the number being $\frac{1}{N+1}\binom{2N}{N}$ (see [K1]). This implies that some automated process would be appropriate. Solutions for trees of size six or more, could then easily be found.

In the preceding discussion, it was assumed that the steady state distribution did not depend on the initial distribution. However, it has also been noted that, at least for algorithm 1, there are some trees which can never be generated from certain other trees through any sequence of updates. If the steady state for some algorithm resulted in two or more subsets of trees with non-zero probabilities which did not have any connecting transformations, then the assumption would not be justified. To justify the assumption, at least for the algorithms under consideration, (with the possible exception of algorithm 5) the trees can be ordered using a left-right skew measure (see chapter 4.2). This measure is such that for any tree, if n is a node of the tree with a non-empty right subtree, then deleting a node from that subtree using any deletion algorithm which does not alter the tree outside of the subtree, will decrease the measure. Further, inserting a node into the left subtree of any node will also decrease the measure.

For all trees except a left linear tree, there is some node with a right subtree. Deleting that node using any but algorithm 5, may result in a right descendant being used as a replace-

ment, thus effectively removing a node from the right subtree. When the key is re-inserted, the new node must fall to the left of the replacement, thereby increasing the left subtree. Thus, the measure of the new tree is less than the old. Therefore, for these algorithms, there is some sequence of transformations which will result in a left-linear tree. This implies that there cannot be two non-communicating subsets of trees in the steady state.

4. How Trees Grow

The measure usually given for a binary tree is the internal path length. This can be directly related to the number of comparisons necessary on average to complete a successful search for an element in the tree, by the formula $Ipl = N(C_N - 1)$, where N is the number of nodes in the tree, and C_N is the number of comparisons. (See [K1]). Thus, it is an effective measure of the relative cost of using that tree. The internal path length of a binary tree is the sum over all nodes of the tree of the number of edges in the path from the root to the node. An equivalent formulation is

$$Ipl = \sum_{v \in T} |l_v| + |r_v| \tag{4.1}$$

where $|l_v|$ is the number of nodes in the left subtree rooted at v, and $|r_v|$ is the number of nodes in the right subtree.

In trying to determine how a tree behaves under the updating process, Eppinger and others measured the change in internal path length. A height balanced tree will have a smaller internal path length than a tree that is badly unbalanced. The worst case tree is one with only one leaf. If the number of nodes is N, then the internal path length of this tree will be

$$Ipl = \frac{N^2 - N}{2} \tag{4.2}$$

Alternatively, if the tree is as height balanced as possible, then

$$Ipl = (N+1)\left\lfloor \log_2(N+1) \right\rfloor - 2^{\left\lfloor \log_2(N+1) \right\rfloor + 1} + 2 \tag{4.3}$$

Since Eppinger's results indicate that the internal path length increases under the updating process using the Hibbard algorithm, it would appear that the trees are becoming more stringy. However, a tree which is a left linear list, a tree which has one path that zig-zags left and right, and a tree which is a right linear list all have the same worst case internal path length. Thus, the internal path length cannot tell us much about the shape of the tree.

In the chapter on the exact fit domain problem, it was noted that right-asymmetric algo-

rithms tended to produce trees that had significantly more nodes in the left subtree than in the right. In the simulations presented so far, the domain is huge, though still of finite cardinality and of finite range. In the following I will argue intuitively that the updating process, with keys chosen uniformly from such a domain for insertion and nodes chosen uniformly from the tree for deletion, using a right-asymmetric deletion algorithm, will result in trees which are deficient in nodes in the right subtree.

4.1. Why Trees Go Bad

To aid intuition, I will first define an *average tree*. The idea is to describe the average shape of a set T_N of binary trees, each tree having N nodes. The average tree will be a full binary tree of height N, with 2^N-1 vertices. Each vertex will have an occurrence value that indicates the average number of times that node would occur if each tree in the set were mapped directly onto the average tree. The occurrence number must be in the range $[0,1]$, and the occurrence of the root must be exactly one, as every tree must have a root. The sum of all the occurrence values is clearly just N, and thus we can refer to this as an average tree of N nodes.

The internal path length of the average tree is the sum over all vertices of the depth of the vertex times the occurrence value of the vertex. This is clearly equivalent to the average path length of the set T_N. An equivalent formulation is

$$E(I) = \sum_{v \in A} |l_v| + |r_v| \tag{4.5}$$

Here, $|l_v|$ is the number of nodes in the left subtree of v, that is the sum of the occurrence values in the left subtree, and $|r_v|$ is the number of nodes in the right subtree. Note the similarity to equation (4.1). The average initial tree will have a shape that is the average produced by all possible insertion sequences. The internal path length of the average initial tree will then be (see [K3])

$$E(I_0) = 2N((1+\frac{1}{N})H_N-2) \tag{4.6}$$

Since the insertion sequences occur in symmetric pairs, the number of nodes in each of

the two subtrees of any node must be the same in the average initial tree. This implies that the occurrence values of the vertices on each level are the same. Such an average tree will be referred to as having level distribution, and will be called a uniform tree. The idea is that the distribution of the nodes is uniform from left to right over the vertices of the average tree. The number of nodes to the left of the root in the uniform tree (and therefore the number to the right) is $\frac{N-1}{2}$.

Now assume that the updating process can be carried out upon the average tree, and that we are using the right-asymmetric Hibbard algorithm. Since the keys are selected uniformly from the domain, the root key will initially be in the center of the domain (on average). Suppose that after some number of updates the root is deleted, and that the right subtree is not empty, so that the successor will be brought up to replace the root. The root key will have effectively migrated $\frac{1}{N+1}$ of the domain interval to the right, since N uniformly spaced keys implies $N+1$ divisions of the interval. The probability of deleting the root is $\frac{1}{N}$ on any given update, so on average N deletions must be performed before the root is again chosen for deletion. All of these deletions are followed by insertions, and each insertion will be distributed uniformly over the interval. Thus, since the root is fixed during these updates, the size of the left subtree will increase by about $\frac{N}{N+1}$, while the right subtree decreases by about the same amount. Since initially the right subtree had $\frac{N-1}{2}$ nodes, we can expect that after $\frac{N}{2}$ such deletions the right subtree will be empty. Thus, about $\frac{N^2}{2}$ updates will be required to empty the right subtree.

It might be supposed that the right subtrees of every node would be emptying at the same rate, and thus after $\frac{N^2}{2}$ updates the tree would have reached its leftmost imbalance. However, Eppinger's results suggest that N^2 updates are required for the tree to reach its greatest imbalance. With a bit more vigorous hand waving, this gap can also be explained.

Consider the left son of the root. Initially the occurrence of this vertex will be slightly less than one. The initial occurrence of the left son of the root is easily computed. The left son can only be null if the input sequence has the largest value appearing first. Thus the occurrence of the left son is $\frac{N!-(N-1)!}{N!} = \frac{N-1}{N}$. With repeated updates on the root, the occurrence must eventually increase to one. Each time this vertex is updated, and its right subtree is not empty, it too will migrate to the right in the domain interval by $\frac{1}{N+1}$. Since its occurrence is close to one, its update rate will be nearly the same as the root, and therefore its migration rate will be nearly the same. Thus, the interval between the root and its left son will remain constant at $\frac{1}{4}$ of the domain interval, which implies that the right subtree of the left son will remain nearly constant at $\frac{N-3}{4}$ nodes, until the right subtree of the root becomes empty.

When the root is deleted and its right subtree is empty, then by the Hibbard algorithm the left son becomes the root. The process now repeats, but this time there only about $\frac{1}{2}$ as many nodes in the right subtree. Generalizing for further nodes, we can see that the time to "empty" all right subtrees is $N^2\left(\frac{1}{2}+\frac{1}{4}+\cdots\frac{1}{2^N}\right) \approx N^2$. Of course, the process won't really empty all the right subtrees, because the further down the left side of the tree we go, the lower are the occurrence values of the vertices. This implies that the update rates, and consequently the migration rates, will also be slower, and so the right subtrees will grow somewhat during the migration. However, this does suggest why the process requires N^2 updates to reach a steady state.

As the average tree becomes increasingly left skewed, the occurrence values deep in the left subtree are increased, while those in higher levels in the right subtree are decreased. Referring to equation (4.5), we see that this implies that the internal path length must increase, reaching some limit after about N^2 updates.

In Eppinger's experiments, and in Knot's, the internal path length originally decreases before increasing. This can most easily be explained in terms of what occurs in low levels of the average tree. It was noted in chapter three, on exact fit domains, that trees with one leaf tended to decrease in probability in the steady state. If the same phenomenon occurs here, the occurrence values of vertices close to the leaves should decrease. Note that in the average initial tree the leaves each have occurrence of $\frac{1}{N!}$.

Looking at a typical random tree, we see that if a node is deleted which has no right son, then if there is a left subtree, substantial rebalancing can occur. This is because replacing the deleted node with the left son causes the path length of every node in the left subtree to be decreased by one. The new node inserted in the second part of the update will probably be at about the same depth as the deleted node on average, and so the internal path length will be reduced. However, nodes will generally have no right subtree only if they are near the fringe of the tree. The rebalancing effect will therefore be of limited extent, and will eventually be overwhelmed by the severe left skewing of the average tree. When the algorithm is applied symmetrically, however, there should be no left skewing, and in that case the rebalancing should be the major effect, as Eppinger's experiments show.

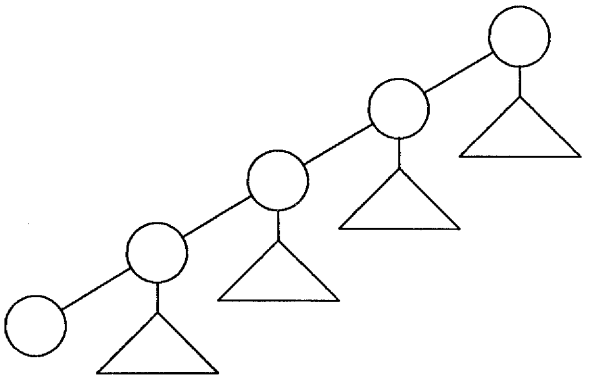
4.2. Measuring A Tree

The ideal way to measure the average behavior of binary trees under the updating process would be to measure changes to the average tree for a large number of trees. However, this is not practical, since trees of even a few hundred nodes would require more vertices in the average tree than there are particles in the universe. The problem of what information to gather depends on what changes to the properties of the tree one expects to verify. In addition, for extensive simulations there is the practical matter of acquiring the information and providing storage for it. Since I could reasonably expect the process to reach some average steady state, it was also possible to divide the information gathering problem into two parts; first, measuring how the tree changed as it approached the steady state, and second, measur-

ing properties of the final average tree.

As will be explained in chapter 5, it seemed reasonable to store the final tree for each run of the update process, and so the problem of gathering information on the final average tree could be postponed until after the simulations were complete. The problem remaining was to decide on reasonably inexpensive measures which would adequately describe the process of change to the tree.

From the analysis of the previous section, it would seem that the final average tree produced by a right-asymmetric algorithm might appear somewhat as follows.



Steady State Tree

In this diagram, each of the subtrees represented by the triangles would presumably become small as the updates progressed. In addition, the number of nodes along the left side of the tree should on average increase. I chose the number of nodes as an appropriate measure, and because of the shape of the diagram, called the set of nodes the *backbone* of the tree, and called the measure the *length* of the backbone.

The sum of the occurrences on the path from the root to any leaf of a uniform tree is the same. This sum is also the expected number of nodes which would lie on that path. So the number of nodes in the backbone is the same as the number in any other path specified by some right-left sequence in the average initial tree. The average length of the backbone of a tree of size N is just the average length of the backbone in the left subtree plus one for

the root, from whence

$$\bar{L}(N) = 1+\frac{1}{N}\sum_{k=0}^{N-1}\bar{L}(k),\ \bar{L}(0)=0 \qquad (4.6)$$

$$= \sum_{k=1}^{N}\frac{1}{k}$$

$$= H_N$$

Another way of measuring the left-right imbalance of a tree would be to count the number of nodes in the left and right subtrees of the root. However, this seems unsatisfactory, since it does not measure the imbalance of any subtree. Once the right subtree becomes empty, no distinction is made between trees in which the right subtree of the left son contains all the remaining nodes, and those in which it is empty. A natural extension of this idea is to count the nodes in the left and right subtrees of every node, and take as a measure the total left and right counts. Thus,

$$Lm = \sum_{v \in T} |l_v|$$

and

$$Rm = \sum_{v \in T} |r_v|$$

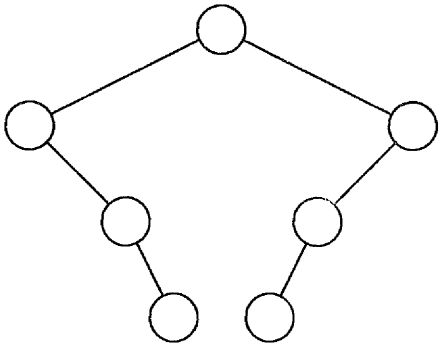
define the left and right measures.

If we examine equation (4.1), we see that $Ipl = Lm+Rm$. Getting the left and right measures of a tree gives the internal path length for free. In general, as the tree becomes increasingly left skewed, the left measure will increase and the right measure will decrease, while the sum of the two will also increase. The relative balance of a tree will be defined as

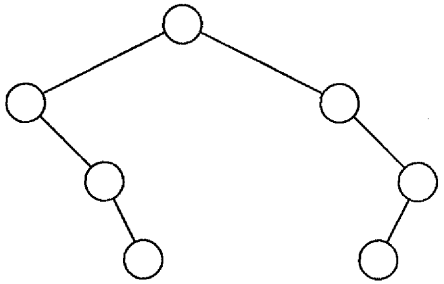
$$Bal = \frac{Rm-Lm}{Rm+Lm}$$

A maximally imbalanced tree will have a relative balance of plus or minus one, and will also have a maximal internal path length.

Consider the following tree.



This tree appears to be left-right balanced, and indeed it is easy to obtain $Lm = Rm = \frac{Ipl}{2} = 6$. If the entire right subtree were now deleted, one would expect that the remaining tree would exhibit some left imbalance. However, $Lm = Rm = \frac{Ipl}{2} = 3$ for this remaining tree. Now consider the following tree.



Here, $Lm = 4$, $Rm = 8$ and $Bal = \frac{1}{3}$. If the bottom left node of the tree is removed and added as the left son of the bottom right node, then $Lm = 5$, $Rm = 8$ and $Bal = \frac{3}{13}$. The internal path length has increased by one, but what is more notable is that moving a node from the left subtree to the right subtree has decreased the balance both relatively and absolutely, falsely indicating that the tree is less right skewed.

If a binary tree is drawn such that the horizontal distance between the children of every node is the same, then the tree would map onto a triangular set of vertices. A full tree of $2^N - 1$ nodes would map onto a triangle of $\frac{N^2+N}{2}$ vertices, with the number of occurrences at

each vertex being the corresponding number of Pascal's triangle. The relative balance is a measure of the imbalance of this structure.

Trees could more reasonably be drawn with the horizontal distance between the children some fraction of the distance between the node and its sibling. The fraction would be chosen so that a projection of the tree onto a horizontal line would preserve the left to right ordering of the nodes. If the fraction is $\frac{1}{X}$, then $X \geq 2$ will preserve this ordering. This leads naturally to the idea of normalized left and right measures. For $X>0$, the left normal measure and the right normal measure are defined as:

$$Lnm(X) = \sum_{v \in T} \frac{|l_v|}{X^{depth(v)}} \tag{4.7}$$

$$Rnm(X) = \sum_{v \in T} \frac{|r_v|}{X^{depth(v)}}$$

When $X = 1$, the equations are those for Lm and Rm respectively. The skew of a tree is defined as

$$Skew(X) = Rnm(X) - Lnm(X) \tag{4.8}$$

and the relative skew as

$$Rel(X) = \frac{Skew(X)}{Rnm(X) + Lnm(X)} \tag{4.9}$$

It is not difficult to show that for $X \geq 2$, moving any leaf in any tree to any external node to the right of the leaf will increase the skew. Also, for $X \geq 2$, for any node, if a node is deleted from a left subtree of the node, the skew will be increased. Inserting a node into the right subtree will increase this value even further. To see this, just notice that the total contribution to both left and right measures of any node totaled over all vertices on the path from the son of a root to the node is less than the contribution of the node at the root. It is still possible for a tree to be skew balanced and have an excess of nodes in its left subtree. However, the subtrees must themselves be right skewed for this to happen. Intuitively, for all X, if the tree is framed using the fraction $\frac{1}{X}$, then a skew-balanced tree would "hang level" if suspended from the root.

Recall that $Ipl = Lm+Rm$ for any tree. Similarly, the normalized internal path length can be defined as $Nipl(X) = Lnm(X)+Rnm(X)$. It is straight forward to calculate the *Nipl* of certain trees. For a linear tree,

$$Nipl(X) = \sum_{i=0}^{N-1} \frac{N-i-1}{X^i}$$

$$= \begin{cases} \dfrac{\left(\dfrac{1}{X}\right)^N - \dfrac{N}{X} + N - 1}{\left(\dfrac{1}{X}-1\right)^2} & \text{if } X \neq 1 \\ \dfrac{N^2-N}{2} & \text{if } X = 1 \end{cases}$$

For a balanced tree, the formulation is a bit more complex. The sum of the left and right subtrees of all nodes on level i is just the total number of nodes minus the number down to and including the nodes of level i. Since the last full level is $\lfloor \log_2(N+1) \rfloor - 1$, and level i has 2^i nodes, the formulation for the balanced tree is:

$$Nipl(X) = \sum_{i=0}^{q-1} \frac{N - \sum_{j=0}^{i} 2^j}{X^i}$$

$$= \begin{cases} (N+1)\dfrac{\left(\dfrac{1}{X}\right)^q - 1}{\dfrac{1}{X}-1} - 2\dfrac{\left(\dfrac{2}{X}\right)^q - 1}{\dfrac{2}{X}-1} & \text{if } X \neq 1,\ X \neq 2 \\ (N+1)q - 2^{q+1} + 2 & \text{if } X = 1 \\ (N+1)\left(2 - \dfrac{1}{2^{q-1}}\right) - 2q & \text{if } X = 2 \end{cases}$$

where

$$q = \lfloor \log_2(N+1) \rfloor$$

As X becomes large, the *Nipl* of a tree becomes less dependent on the shape of the tree, with $\lim_{X \to \infty} Nipl(X) = N-1$ for all trees. For this project, normalized values were gathered for $X = 1$ and $X = 2$ at various points during the simulation.

The nodes of a full tree would be distributed uniformly if drawn with a fractional

displacement of sons of $\frac{1}{2}$ and projected onto a horizontal line. If the line is then subdivided into equal intervals, each interval would hold an equal number of nodes. A profile of a tree is just a bar chart of the number of nodes falling into each slot. Profiles for specific trees and for average trees are included in the next chapter.

It should be noted that average profiles may not give an accurate picture of a set of trees. Two full trees would yield on average a flat profile, but two trees, one with the nodes distributed over the left subtree and the other with the nodes distributed over the right subtree would also give a flat profile. Since there can only be one flat profile, we might erroneously conclude that the two profiles represented similar average trees. However, the internal path lengths of the two average trees would be quite different. A similar cautionary note should be heeded when trees are skew balanced. Skew balancing does not guarantee height balancing, especially when average trees are concerned.

4.3. Predictions for Algorithms

The algorithms outlined in chapter 2 were chosen to try to demonstrate some of the effects of changing the deletion algorithm. The only difference between algorithm 1 and the Hibbard algorithm occurs when the right subtree of the deleted node is empty and the left is not. In this case, two differences in the effects of the different algorithms should occur. The first of these is that for nodes near the fringe of the tree, the subtrees will not be significantly rebalanced in algorithm 1 as they might be in the Hibbard algorithm. This should mean that there will be less initial improvement for algorithm 1, and that the final tree will have an even longer path length.

The second difference occurs when the root of the tree has migrated to the extreme right of the domain and has an empty right subtree. When the root is then deleted, it does not move back to the root of the left subtree, but instead moves only $\frac{1}{N+1}$ to the left to its predecessor. The net effect then is that the root stabilizes over some small region near the extreme right of the domain. The migration of its left son is not affected, so the next right

subtree will now begin to decrease at the rate of $\frac{1}{N^2}$ per deletion. Thus, the stable tree will still require $O(N^2)$ updates to be achieved. However, the final distribution of right subtrees will be smaller than for Hibbard, since subtrees are not pulled up beyond the root, and coupled with the fact that less rebalancing occurs near the fringe, this implies that the internal path length of these trees should be considerably longer than for the Hibbard algorithm. The trees will be considerably more left-skewed as well.

Algorithm 2 is just the symmetric version of algorithm 1. Being symmetric, there should be no skewing. Since less rebalancing occurs, there should be less improvement than for the symmetric Hibbard algorithm.

Algorithm 3 differs from the Hibbard algorithm in that when the left subtree of the deletion node is empty, substantial rebalancing of the subtree can occur. Thus, more rebalancing of the fringe areas of the tree can occur, leading to greater initial improvement of the internal path length. There should be little long term gain, however, since the left subtree of the root, and those of its left descendants, will almost never be empty. Thus, the stable tree should be as left skewed as the one for the Hibbard algorithm, with only a small improvement in internal path length due to better rebalancing in the fringe areas.

Being symmetric, algorithm 4 should not exhibit any skew imbalance. Since more rebalancing occurs near the fringe, we might expect the asymptotic value of the internal path length to be smaller than for the symmetric Hibbard algorithm.

Algorithm 5 was selected on the advice of Ian Munro. The idea was to try to minimize the internal path length with a minimal rebalancing scheme.

5. Simulation Results

The simulation of the various updating algorithms is straightforward. A simulation consists of several trials, or runs, performing a long series of updates on a single tree in each run. In each run, an initial tree of N nodes is created by generating N random keys and inserting them into an initially empty tree. Data is then gathered, then blocks of updates are performed, each followed by more data gathering. When enough updates have been performed, the run is completed, and the process is repeated for the next run. The problem of how many updates are enough is a bit tricky. If too few are performed, then the tree might not have reached a stable configuration. Alternatively, if a great many are performed, it may be wasteful of computer resources with little additional information gained. So it seemed reasonable to allow for the restarting of the simulation to extend the number of iterations for each run.

To allow such extensions to runs, after each run completes, the entire tree is written to a file. This design also allows the process to be restarted should the computer system crash during a simulation. Finally, keeping each of the final trees allows for additional data gathering on the final state of the trees after the simulations are complete.

During each run, the length of the backbone, and the left and right normal measures for $X=1$ and $X=2$ were regularly computed and added to a total computed in previous runs. When the simulation was complete, this information could be used to compute the average of these measures at each of the appropriate iteration points. An initial file specified to the process the number of runs, the number of update iterations to be performed on each run, the points during the simulation when data was to be gathered, and other details such as the size of the tree to be used in the simulation. This design allowed not only the number of iterations of each run to be extended, but also the number of runs to be increased if further verification of results were desired.

Simulations for each of the five algorithms described in chapter two were carried out for trees of 32, 64, 128, 256, 512 and 1024 nodes. For each size of tree, fifty runs were made.

Approximately $2N^2$ iterations were performed on each of the larger trees, while proportionally greater numbers of iterations were performed on the smaller trees as the relative variation in these tended to be greater.

The language used for the simulations was C, running in the Unix environment on a Vax 780. About 120 hours of central processor time were used.

Pseudo-random sequences of integers were generated using the algorithm given by Knuth in volume 2, chapter 3.2.2 as Algorithm M. (See [K2]) The integers were used as keys during insertion, and the middle bits were used to randomly select nodes for deletion.

In table 5.1, the final internal path lengths for trees of the various sizes for each of the algorithms is listed. In addition, for purposes of comparison, the relevant data from Eppinger's simulation is listed, H being his results for the Hibbard algorithm, and HS being his results for the Hibbard algorithm applied symmetrically. Finally, BAL is the ratio of the balanced tree path length to the expected.

TABLE 5.1

Ratio of Final Path Length to Expected Initial Length

	For trees of size					
Alg.	32	64	128	256	512	1024
1	1.237	1.353	1.533	1.850	2.299	2.957
2	0.954	0.956	0.932	0.934	0.917	0.906
3	0.947	0.957	0.989	1.051	1.141	1.277
4	0.903	0.891	0.878	0.878	0.876	0.871
5	0.827	0.829	0.828	0.835	0.832	0.830
H		0.97	1.00	1.06	1.16	1.30
HS		0.905	0.890	0.888	0.890	0.881
BAL	0.736	0.732	0.729	0.728	0.727	0.726

The final internal path lengths were obtained by averaging the sum of all the left and right measures when the number of iterations was greater than N^2. As expected from the discussion in chapter 4.3, the internal path length produced by algorithm 1 is considerably greater than that produced by either algorithm 3 or by the Hibbard algorithm. For trees of 1024 nodes, algorithm 1 produces a final average path length of more than twice that produced by the Hibbard algorithm. Also, algorithm 3, the Knuth algorithm, is only marginally better than the Hibbard algorithm.

A comparison of the results of the various symmetric algorithms also seems to agree with predictions. Algorithm 2, which does little rebalancing, does not do as well as the symmetric Hibbard algorithm, which in turn is just marginally worse than algorithm 4, the symmetric Knuth algorithm. Algorithm 5 gives the over all best results, but then it cheats by doing some explicit rebalancing on each update.

Eppinger gave a least squares multiple regression approximation for the ratios of the Hibbard algorithm as

$$E_H(I_\infty) \approx 0.020 \log_2^2 N$$

where I have only given the highest order term. A similar regression for algorithm 3 gives

$$E_3(I_\infty) \approx 0.016 \log_2^2 N$$

For algorithm 1, the best fitting polynomial in $\log_2 N$ seems to be cubic. The leading term is $0.0068 \log_2^3 N$. However, an even better fit for algorithm 1 is given by

$$E_1(I_\infty) \approx 0.0627\, N^{\frac{1}{2}} + 0.8695$$

There are so few data points (only 6) available for these computations, that any such results must be treated with skepticism, unless some theoretical reason can be found to support them. Additional data points could be obtained by further simulations; however, to be definitive, large trees would be required. Considering that the number of updates required to obtain the steady state set of trees appears to be $O(N^2)$, the cost of getting these data values may prove to be prohibitive.

Graphs showing the ratios of the average internal path length to the initial expected path length plotted against the number of updates expressed as a fraction of N^2 for algorithms 1 and 3 can be found in the appendices. Algorithm 1 is interesting in that almost no initial improvement in the internal path length is observed. The Knuth algorithm (number 3) shows similar initial improvement to that exhibited by the Hibbard algorithm.

In table 5.2, the average lengths of the backbones, expressed as a ratio to the expected initial length, are given. Again, these were obtained by averaging data for points where the

number of iterations exceeded N^2. As expected, the values for algorithms 1 and 3 increase dramatically as updates are performed.

TABLE 5.2

Ratio of Final Length of Backbone to Initial Expected Length

	For trees of size					
Alg.	32	64	128	256	512	1024
1	2.104	2.608	3.437	4.740	6.640	9.399
2	0.922	0.884	0.885	0.796	0.887	0.875
3	1.424	1.606	1.890	2.286	2.813	3.563
4	1.034	1.067	1.065	1.055	1.096	1.096
5	1.127	1.124	1.130	1.115	1.171	1.101

Algorithm 1 produces by far the longest backbone, being more than two and half times as long as that of algorithm 3, for trees of size 1024. For 1, regression analysis shows that

$$\bar{L}_1(N) \approx 0.27\, N^{\frac{1}{2}} + 0.50$$

while for algorithm 3

$$\bar{L}_3(N) \approx 0.08\, N^{\frac{1}{2}} + 0.975$$

An interesting and unexpected result is that algorithm 2 appears to reduce the length of the backbone. On the other hand, comparing 4 and 5, it seems that rebalancing tends to lengthen it somewhat.

Graphs of the backbones, given as ratios to the expected initial backbone, plotted against the number of iterations divided by N^2 for algorithms 1 and 3 are included in the appendices.

As predicted, the balance and skew measures for algorithms 1 and 3, show the final trees to be extremely left skewed. In table 5.3, the results for these two processes are tabulated. For all the other algorithms, the relative balances were essentially zero for both $X=1$ and $X=2$, also as expected. Although the relative skew calculated for $X=2$ is generally more negative than that for $X=1$ for processes 1 and 3, I have been unable to derive any real insight from that observation. As for previous tables, these values are averages taken when the number of iterations exceeds N^2.

Profiles, as described in chapter 4.2, are also included in the appendices. One tree is

TABLE 5.3

Distortion of Trees by Algorithms 1 and 3

	Relative Balance		Relative Skew	
Size	1	3	1	3
32	-0.523	-0.186	-0.798	-0.372
64	-0.581	-0.227	-0.868	-0.475
128	-0.658	-0.288	-0.911	-0.584
256	-0.733	-0.362	-0.943	-0.683
512	-0.797	-0.434	-0.963	-0.761
1024	-0.849	-0.514	-0.975	-0.826

profiled as it appears initially, and as it appears after it has been acted on by each of the various algorithms. The initial tree in each profile was generated by the same initial sequence of keys. Following these, average final profiles for each of the various algorithms are given. The profiles designated by algorithm 'X' are those of an average over fifty runs of trees with no updates performed. As might be supposed, 1 and 3 tend to have profiles indicating extreme left skewing, while the others tend to produce more balanced profiles. One interesting note is that 2 appears to reduce the ends of the profile, giving a slight hump shape. Together with the reduced length of the backbone mentioned earlier, this seems to indicate that the outer edges of the tree are being reduced and the nodes are being concentrated toward the center. No explanation has been offered to me for this, and I offer none to the reader.

In tables 5.4 through 5.8 the average number of nodes in the right subtree of each of the ten topmost members of the backbone are given for each of the five algorithms. For the balanced algorithms 2, 4 and 5, these values are much as would be expected from balanced trees. For algorithm 1, the topmost subtree seems to be almost constant at 1.4 over the different sizes of trees. Note that the only way nodes can be added to this subtree is if they are inserted to the right of the root. However, if the root is deleted, and the first two subtrees are empty, then the third subtree becomes the second. This explains why there seems to be a sudden jump in the average size of the second right subtree, over the size of the first. The size of the second subtree seems to be about $N^{\frac{1}{2}}/2$. The sizes taper off gradually with depth. For trees of size 1024, the first subtree with average size of zero occurs at depth 77.

For algorithm 3 (table 5.6), the subtrees are somewhat larger than for algorithm 1. The sizes taper off more quickly as well. For trees of size 1024, the first zero subtree occurs at a depth of 31.

TABLE 5.4

Average Right Subtrees for Algorithm 1
For trees of Size

Depth	32	64	128	256	512	1024
1	1.02	1.58	1.78	1.22	1.46	1.30
2	2.32	4.98	4.54	7.88	10.36	16.22
3	3.72	5.18	5.98	10.26	12.40	19.38
4	4.24	6.04	8.74	10.48	17.50	22.46
5	4.06	5.60	8.98	10.74	14.74	17.52
6	3.26	4.72	8.50	12.84	14.60	19.42
7	2.56	5.38	9.72	10.78	15.66	22.60
8	0.96	4.48	7.18	12.70	16.44	18.82
9	0.96	4.10	7.08	9.52	15.10	18.64
10	0.16	3.90	6.66	10.70	16.58	21.54

TABLE 5.5

Average Right Subtrees for Algorithm 2
For trees of Size

Depth	32	64	128	256	512	1024
1	15.86	31.32	58.48	157.16	238.88	460.28
2	7.90	19.98	40.46	62.02	142.36	301.72
3	3.26	6.54	16.78	22.88	87.48	168.84
4	0.84	1.68	5.48	6.62	22.72	56.74
5	0.28	0.30	1.02	1.56	11.80	22.18
6	0.04	0.04	0.42	0.66	2.16	5.04
7	0.00	0.00	0.30	0.06	0.54	1.38
8	0.00	0.00	0.06	0.04	0.20	0.96
9	0.00	0.00	0.02	0.00	0.02	0.12
10	0.00	0.00	0.02	0.00	0.00	0.06

TABLE 5.6

Average Right Subtrees for Algorithm 3
For trees of Size

Depth	32	64	128	256	512	1024
1	6.14	10.02	14.54	21.76	34.64	49.80
2	8.82	14.38	22.80	32.86	50.64	79.24
3	4.84	11.20	22.00	32.80	52.98	70.72
4	3.98	8.48	15.76	29.40	52.36	64.98
5	1.60	6.42	16.26	26.28	38.78	67.08
6	0.44	3.34	10.06	23.98	42.88	57.82
7	0.24	1.38	7.88	21.62	39.70	62.72
8	0.10	0.74	4.44	13.88	33.80	49.00
9	0.00	0.14	2.42	13.36	27.98	57.52
10	0.00	0.02	0.86	11.80	25.84	52.84

TABLE 5.7

Average Right Subtrees for Algorithm 4
For trees of Size

Depth	32	64	128	256	512	1024
1	15.92	31.22	62.12	113.84	245.58	520.30
2	7.30	16.24	34.96	85.32	149.88	231.06
3	2.92	7.12	13.26	33.20	59.72	144.76
4	1.26	2.52	7.02	11.50	27.92	72.36
5	0.32	1.36	3.02	3.70	15.74	27.22
6	0.00	0.18	1.04	1.54	4.10	13.62
7	0.00	0.12	0.48	0.56	1.62	4.10
8	0.00	0.02	0.12	0.22	0.52	1.68
9	0.00	0.02	0.00	0.06	0.00	0.38
10	0.00	0.00	0.00	0.00	0.00	0.22

TABLE 5.8

Average Right Subtrees for Algorithm 5
For trees of Size

Depth	32	64	128	256	512	1024
1	16.52	31.22	63.92	116.72	238.22	551.96
2	6.82	16.16	34.46	83.72	141.26	245.74
3	2.82	6.24	12.74	30.18	66.22	110.02
4	1.14	2.88	6.10	12.44	34.10	58.78
5	0.10	1.32	2.94	3.78	13.26	28.58
6	0.02	0.36	1.04	1.68	6.04	12.58
7	0.00	0.20	0.34	0.62	2.94	5.46
8	0.00	0.02	0.06	0.14	1.28	1.34
9	0.00	0.02	0.02	0.12	0.44	0.84
10	0.00	0.00	0.00	0.00	0.08	0.18

6. For the Future

I hope that this thesis has shed some light on the problem of what happens to binary trees under extensive updating, and will provide some possible directions for future exploration. Towards that last goal, I will include here some questions that I consider interesting.

The following algorithm, which I call RR for recursive replacement, is one that a naive programmer hooked on recursion might use.

```
Procedure RR(ν).
 begin
 Y := successor(ν);
 if (Y not null) begin
      RR(Y);
      replace ν with Y;
      end;
 else begin
      Y := predecessor(ν);
      if (Y not null) begin
           RR(Y);
           replace ν with Y;
           end;
      else /* ν is a leaf */
          detach ν from tree;
 end.
```

As it stands, RR is right asymmetric. Structurally, each deletion would only remove a leaf from a tree. Thus, it would seem to be symmetric to insertions, which only add leaves to the tree. However, the value distribution would be affected when nodes other than leaves were deleted. This algorithm should prove worse than any of those tested. The reason is that as the right subtrees of the backbone become empty, lower subtrees are never brought up to replace them. In the previous chapter, it was pointed out that algorithm 1 never brought up a replacement for the topmost right subtree. There, that subtree could only receive nodes from insertions, and it appeared that that subtree remained nearly constant at an average of 1.4 nodes. RR should then cause the same sort of thing to happen for every right subtree. In addition, no rebalancing during deletions ever occurs, not even, as in algorithm 1, for subtrees of the predecessor or the successor. This leads to the conclusion that the internal path length could be as bad as $O(N^2)$.

Another question is whether there is a deletion algorithm which would leave the trees with the distribution of initial random trees. This question is particularly interesting when one remembers that Hibbard originally showed that his algorithm yielded random trees after a deletion, although further insertions were later shown to change the distribution. If RR were to be applied symmetrically, then I believe that it would change the internal path length very little. The reason is that almost no rebalancing ever occurs during deletion. However, other effects due to redistribution of values throughout the tree may now become predominant. Consider, for example, the unexplained humping that seems to be produced by algorithm 2.

There are many other possible algorithms, and most of them would appear to be bad. One such algorithm is to simply connect the father of the deleted element to the left son, and make the right son of the deleted element the right son of the predecessor. This algorithm would appear to be left asymmetric, although the limiting case is by no means clear. Applied symmetrically, the question is 'just how bad would the tree get?'

Another interesting possibility is to make the choice between mirror images of any asymmetric algorithm dependent on the relation of the deleted node to its father, alternating at the root. For example, we could choose the left asymmetric version when the element to be deleted was the left son of its father, and the right asymmetric version otherwise. These algorithms I would call semi-symmetric, since they appear to lead to symmetric trees which are the nonetheless bad. Analyzing the algorithm just described in the manner of chapter 4, it would appear that the resulting tree would achieve a zig-zag shape, since alternating subtrees would be emptied.

I also think that further extensions to the theory of exact fit domains, and other small domains slightly larger than the size of the tree, might well lead to further interesting insights.

Pictures and Profiles

On the following pages are several drawings representing trees and their profiles, as described in chapters 4 and 5. The first eight pages each consists of a drawing of a 32 node tree and its profile. The profiles are scaled so that the bar of maximum height is always drawn the same height on the page, so the scale is different for each profile. The dotted line represents the balance height; that is, the height using the appropriate scale which would represent the balanced tree. Thus, the lower the dotted line, the more unbalanced is the corresponding tree. The trees are drawn with a factor of $\frac{1}{2}$, as explained in chapter 4. The vertical distances between nodes in each drawing are constant, but may differ between different drawings, since the diagrams are scaled so that they are always the same height.

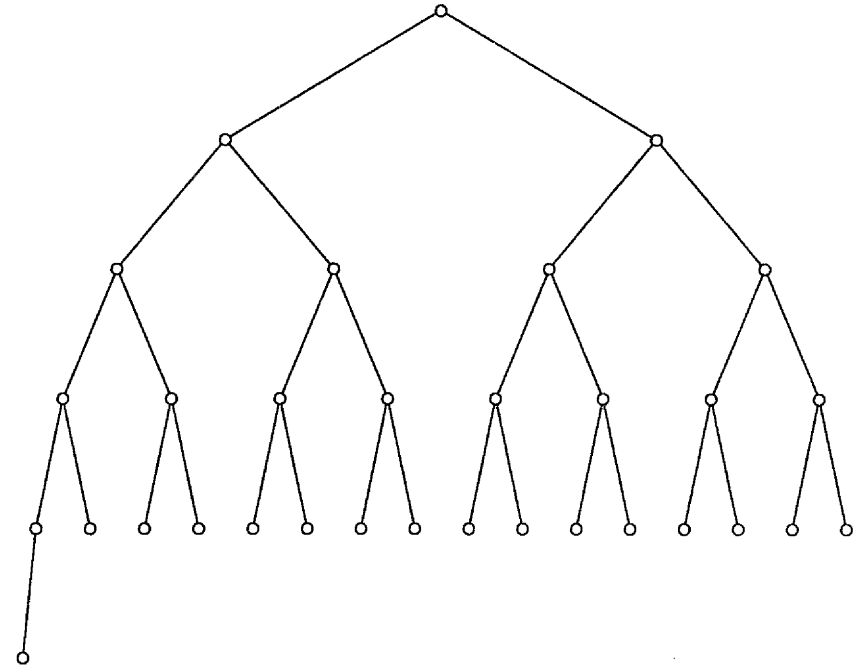
The first drawing represents a balanced tree, and is included for reference purposes. Notice that the extra node in excess of a full tree must be at the extreme left of the tree in order that the profile be flat.

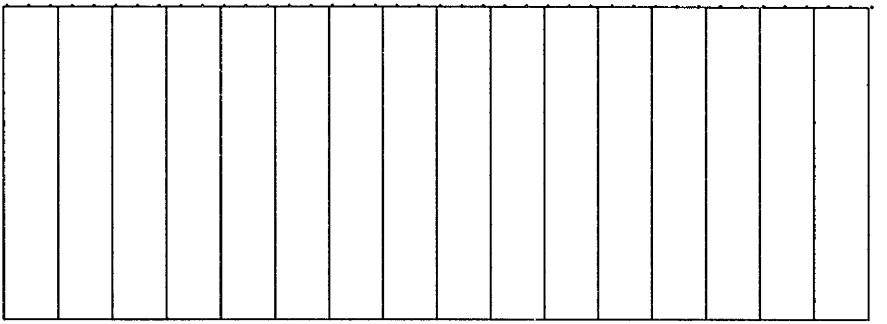
The seven following trees were all obtained using the same initial insertion sequence. The first of these diagrams shows the tree as it appeared before any updates were performed. Following this, the tree is shown as it appeared after being acted upon by each of the updating algorithms 1 through 4. The next diagram shows the tree resulting from the same initial insertion sequence, but with the minimal rebalancing scheme described in chapter 2 applied after each insertion. The final tree shows this last tree after being acted upon by algorithm 5.

These drawings are just snapshots of the trees at a particular point during their evolution, and so cannot be taken as representative of the average shape of the trees over time. However, they do graphically portray the results of applying an unsymmetric algorithm to a given tree. They also show the correspondence between the profile of a tree and its shape.

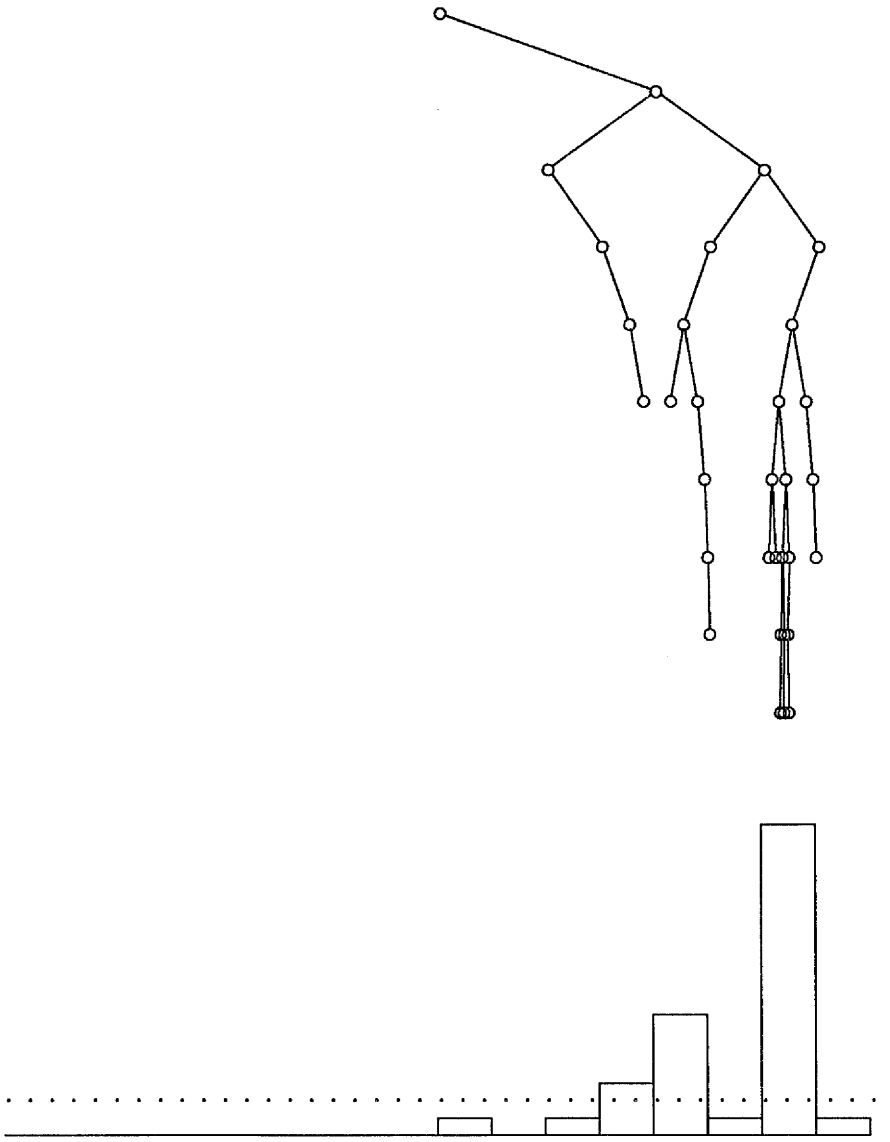
Following the drawings of the trees, are profiles which represent the average values for each of the five algorithms for each of the different trees used in the simulation. For com-

pleteness, profiles for average initial trees are also included. Those labeled as algorithm X were generated using the standard insertion algorithm, while those labeled as algorithm Y resulted from insertions which included the minimal rebalancing scheme.

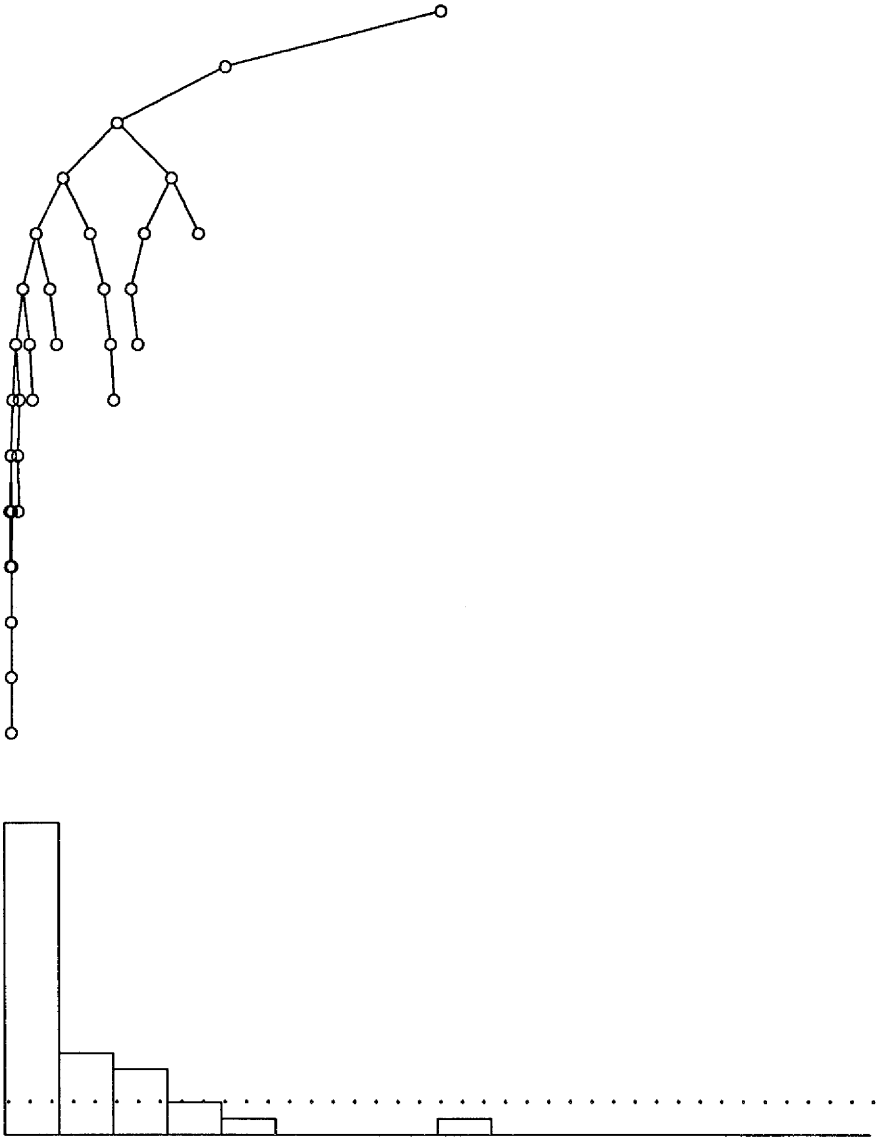




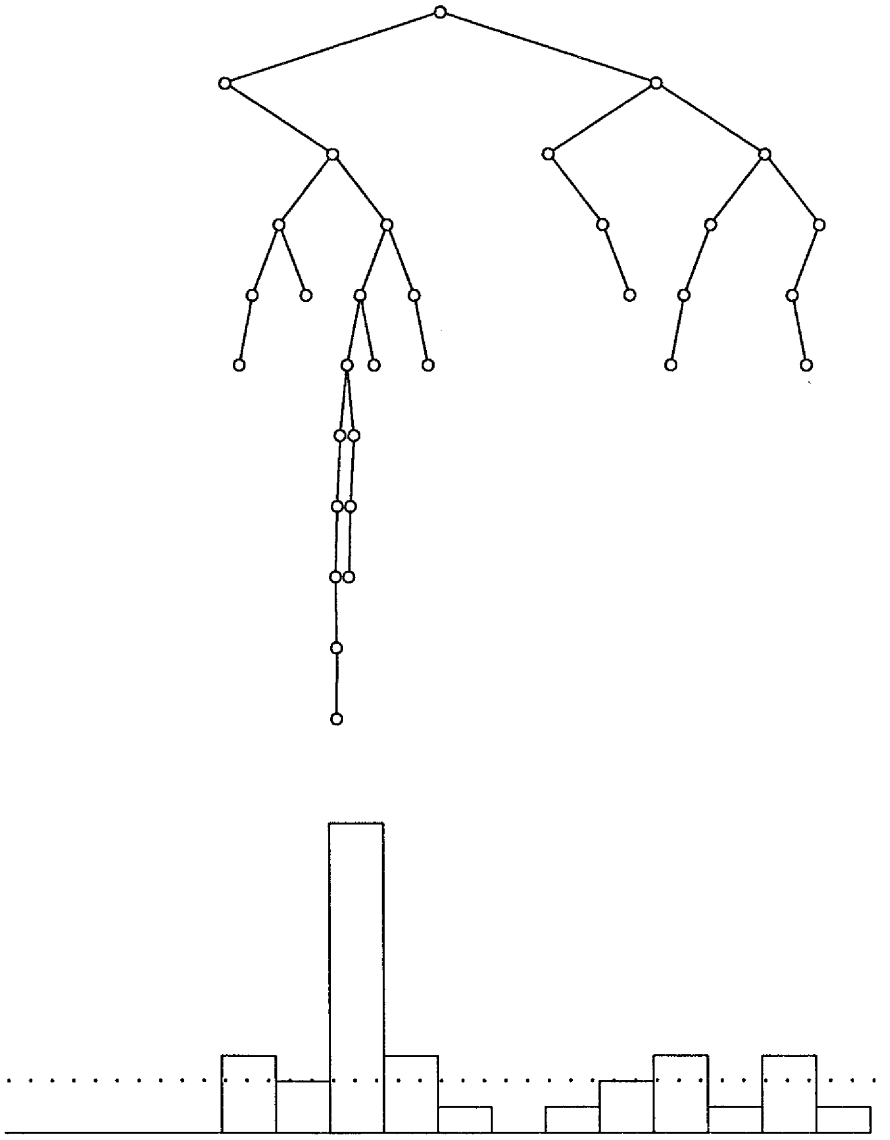
Balanced Tree of 32 Nodes



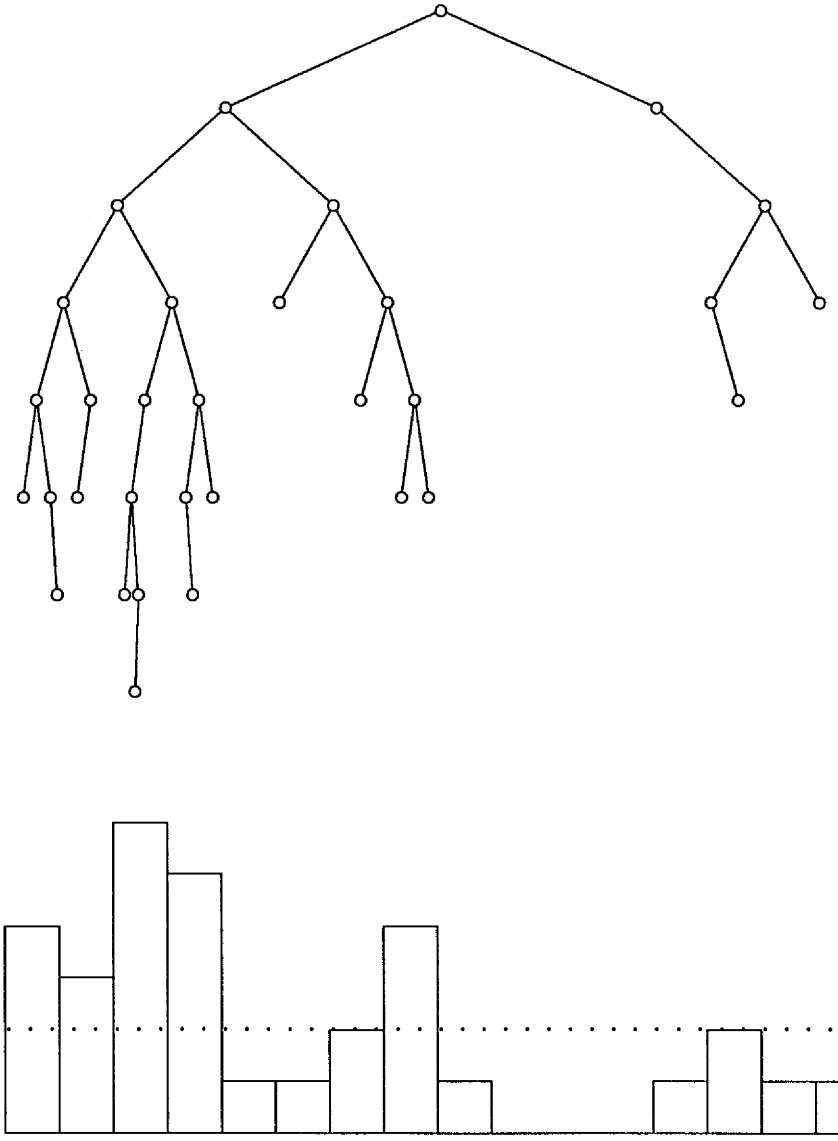
Initial Tree of 32 Nodes



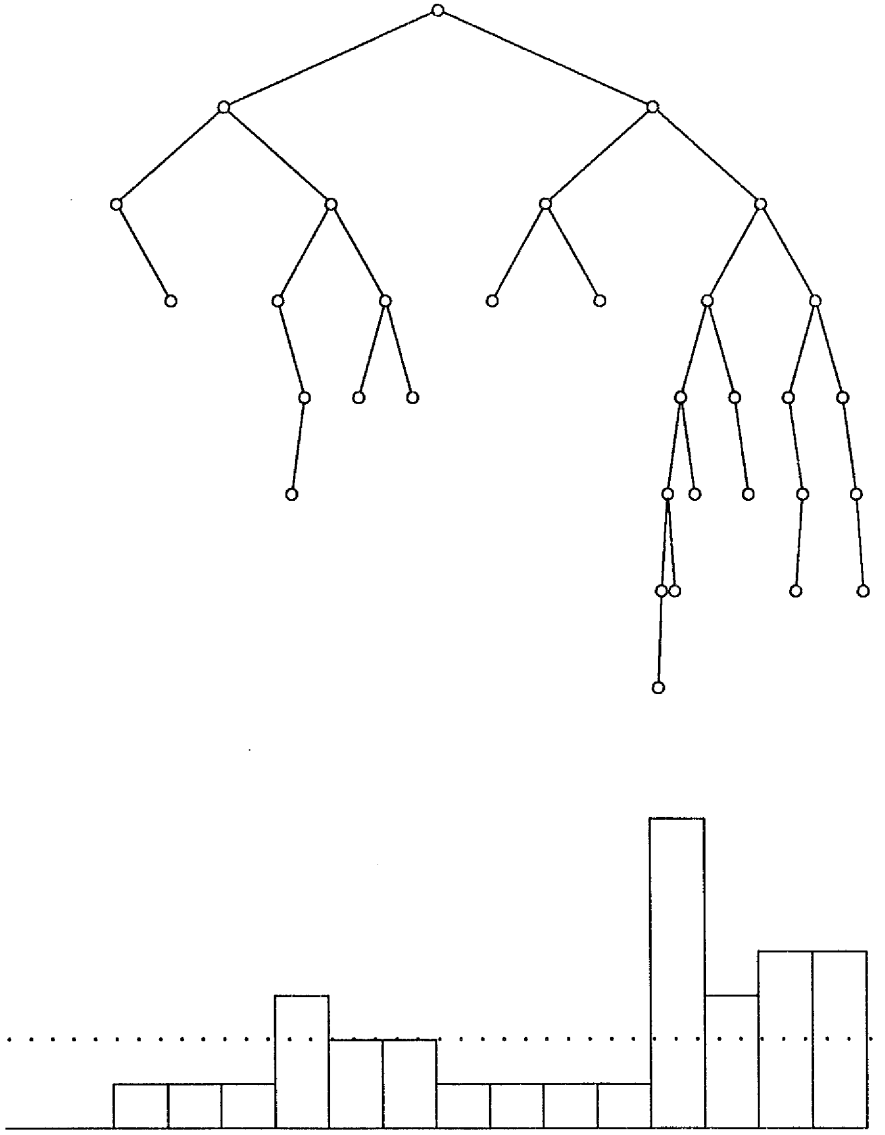
Same Tree After 5000 Updates Using Alg. 1



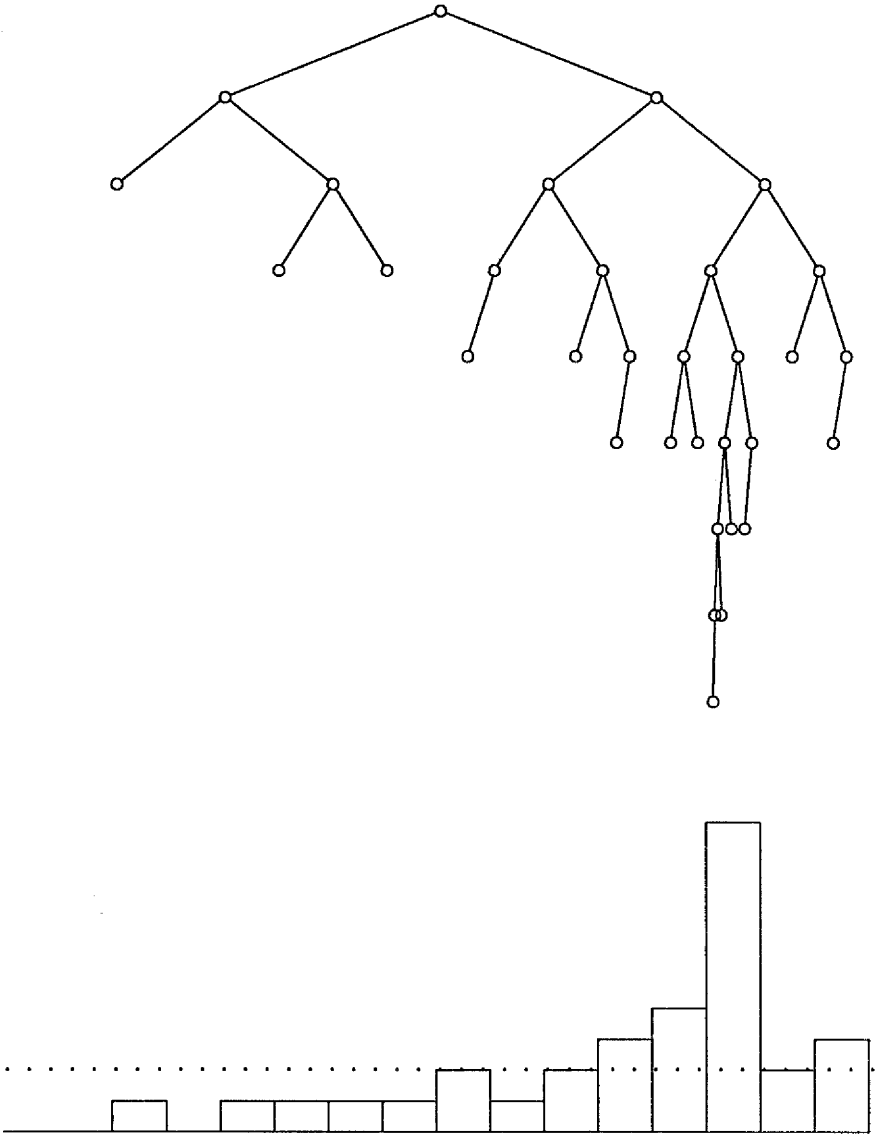
Same Tree After 5000 Updates Using Alg. 2



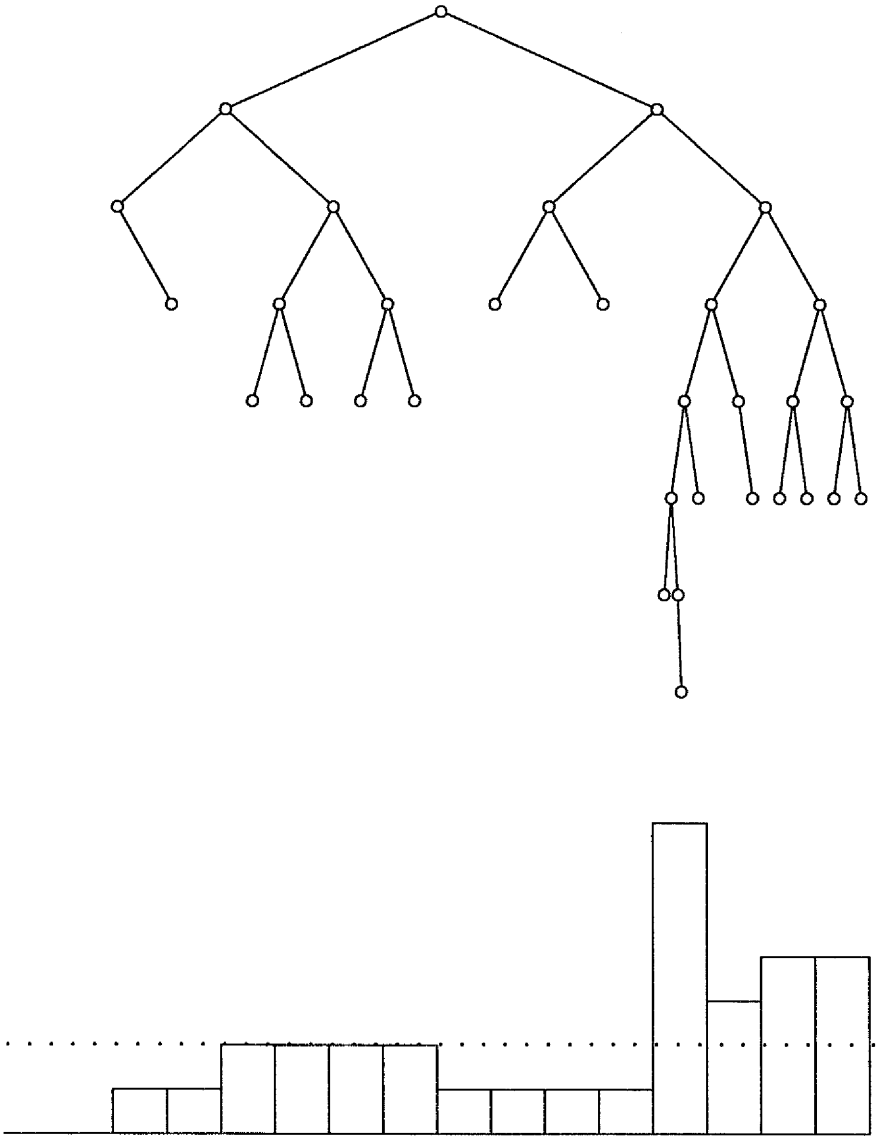
Same Tree After 7500 Updates Using Alg. 3



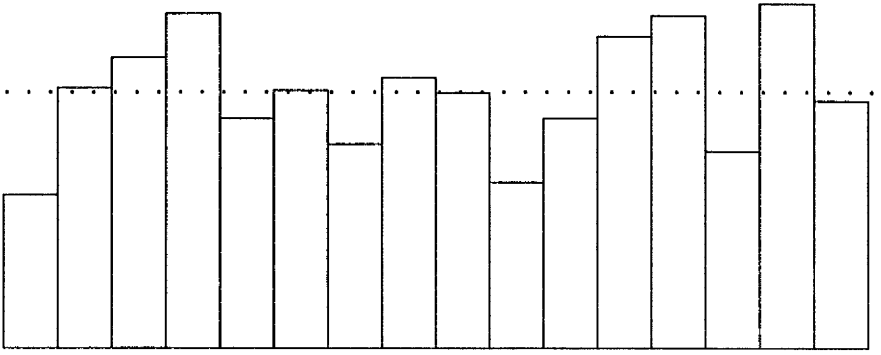
Same Tree After 7500 Updates Using Alg. 4



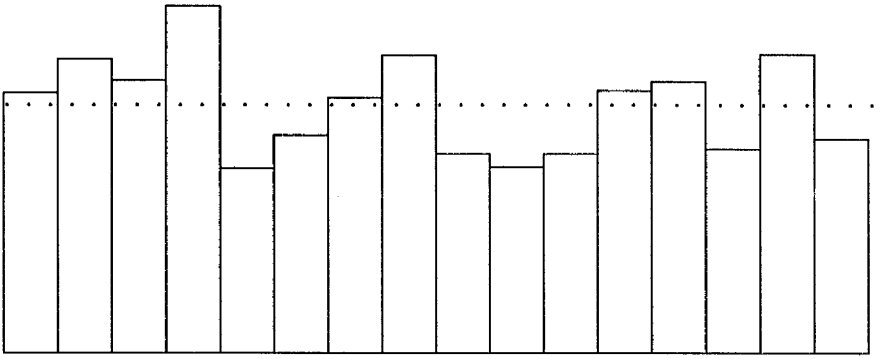
Initial Tree Using Rebalance of 3 at Leaves



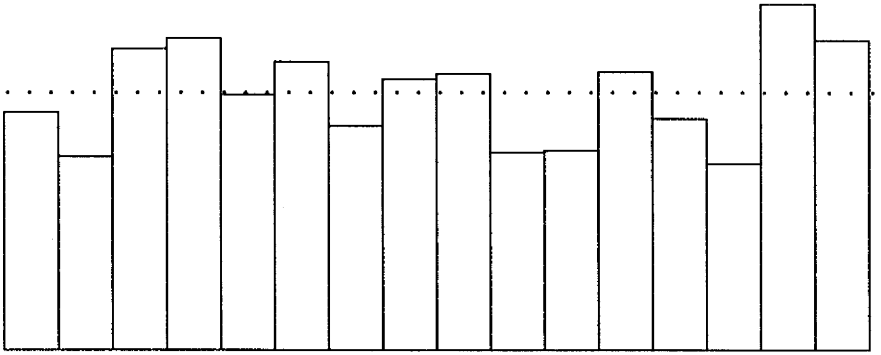
Tree After 7500 Updates Using Alg. 5



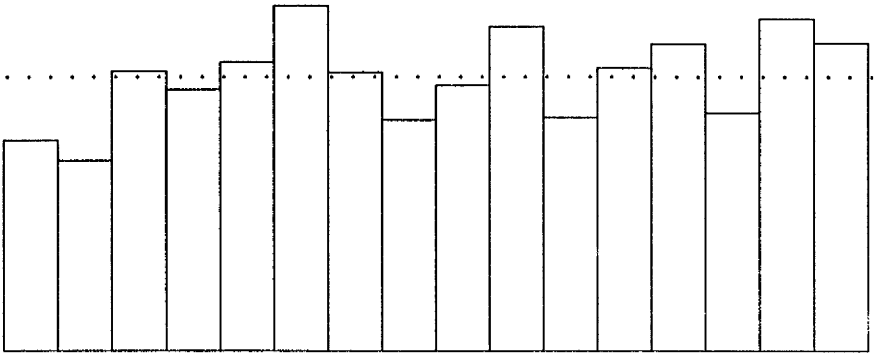
Average 50 Runs: 32 Nodes 0 Updates Using Alg. X



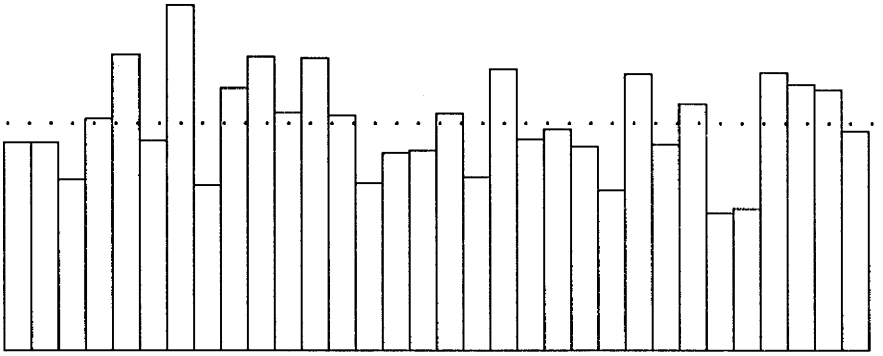
Average 50 Runs: 64 Nodes 0 Updates Using Alg. X



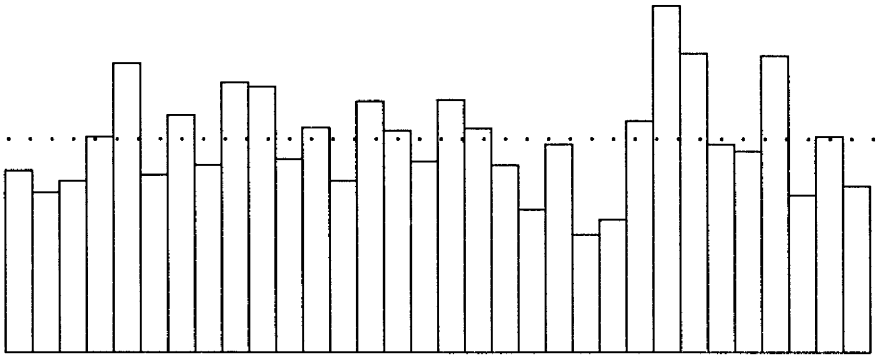
Average 50 Runs: 128 Nodes 0 Updates Using Alg. X



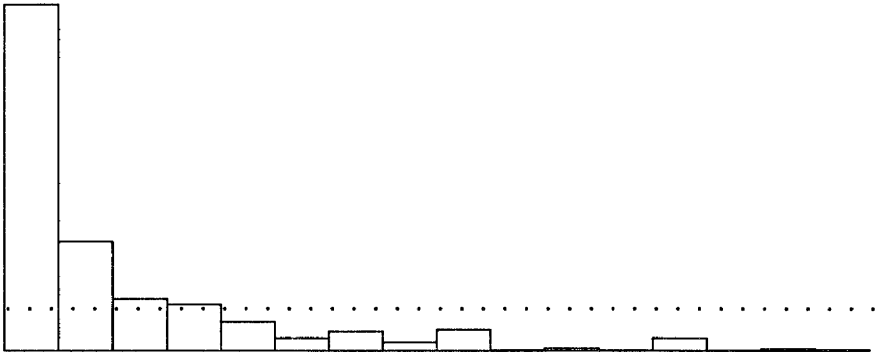
Average 50 Runs: 256 Nodes 0 Updates Using Alg. X



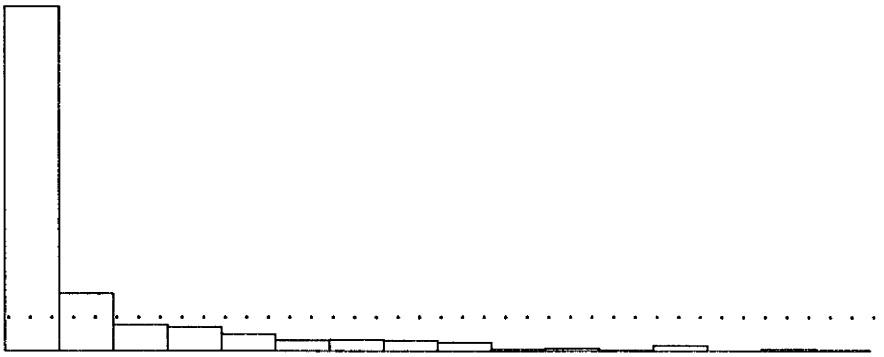
Average 50 Runs: 512 Nodes 0 Updates Using Alg. X



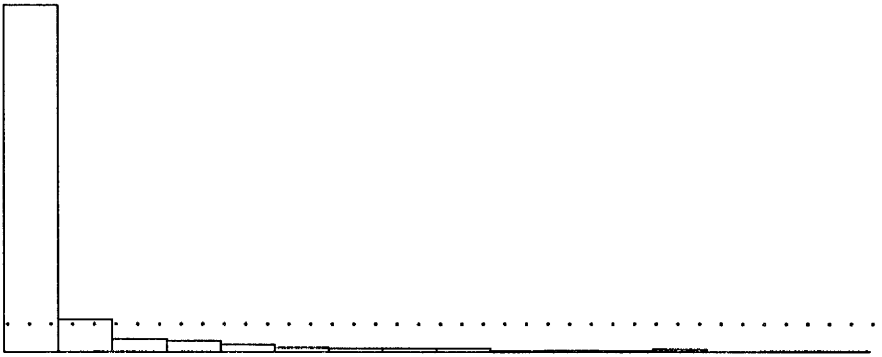
Average 50 Runs: 1024 Nodes 0 Updates Using Alg. X



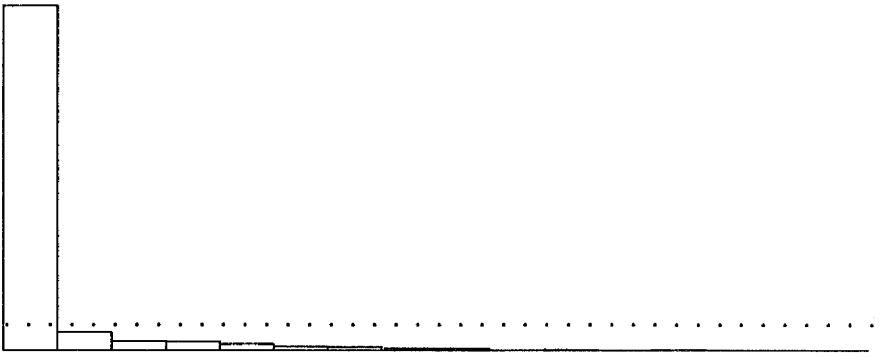
Average 50 Runs: 32 Nodes 5000 Updates Using Alg. 1



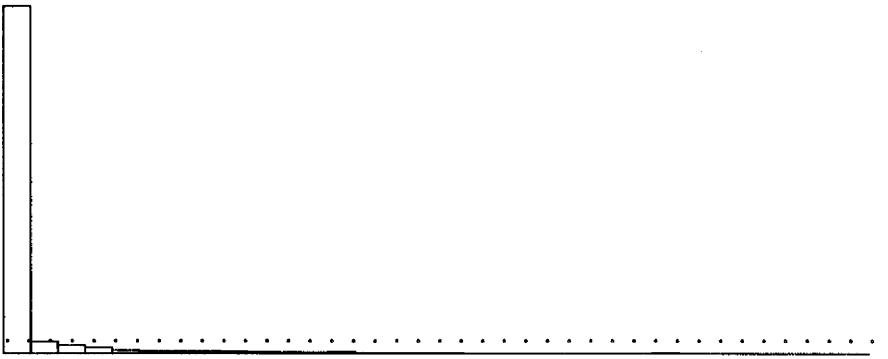
Average 50 Runs: 64 Nodes 20000 Updates Using Alg. 1



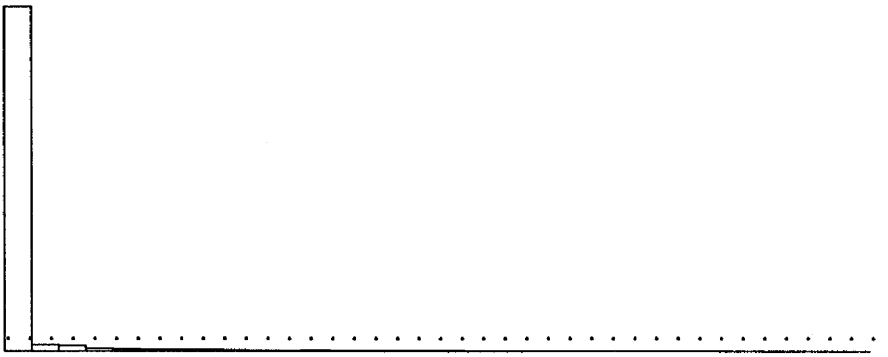
Average 50 Runs: 128 Nodes 30000 Updates Using Alg. 1



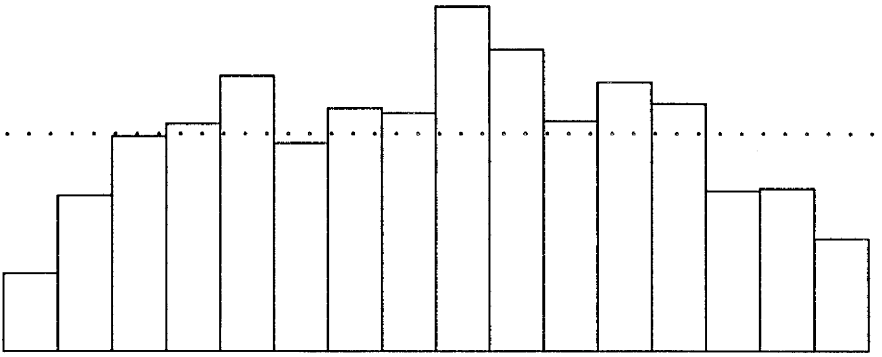
Average 50 Runs: 256 Nodes 120000 Updates Using Alg. 1



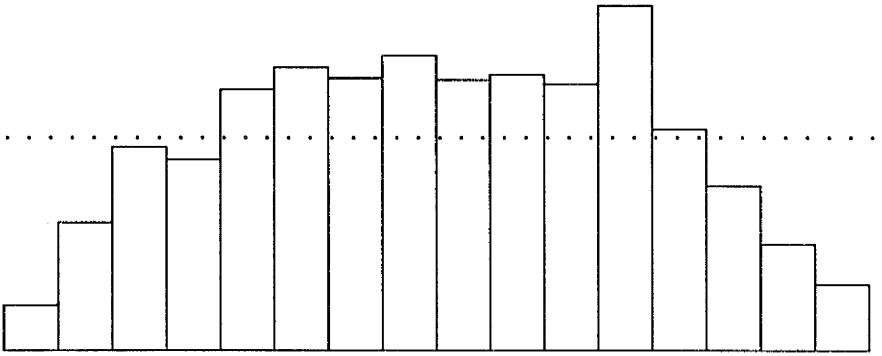
Average 50 Runs: 512 Nodes 450000 Updates Using Alg. 1



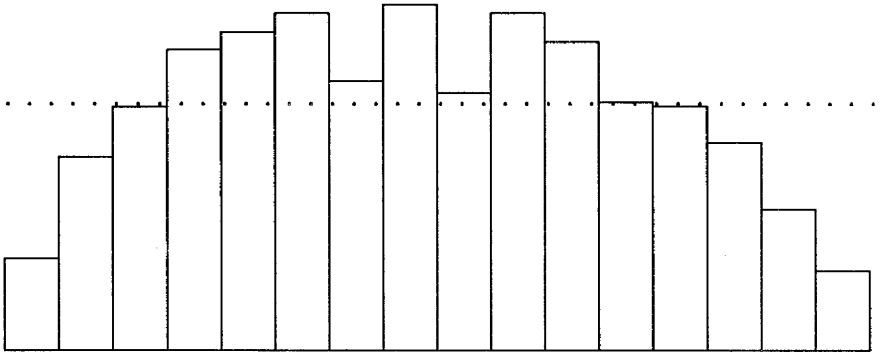
Average 50 Runs: 1024 Nodes 2000000 Updates Using Alg. 1



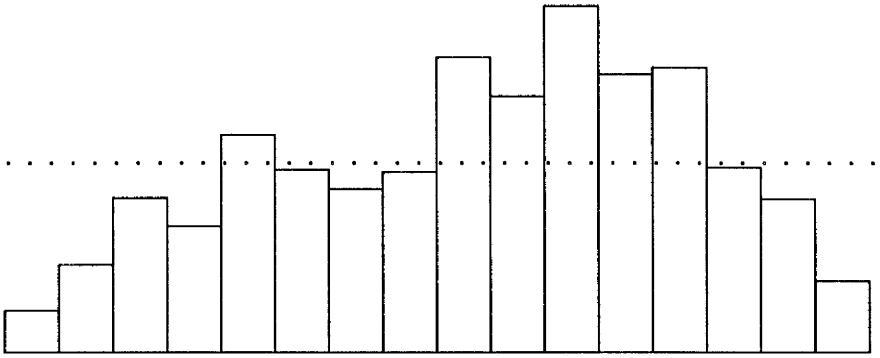
Average 50 Runs: 32 Nodes 5000 Updates Using Alg. 2



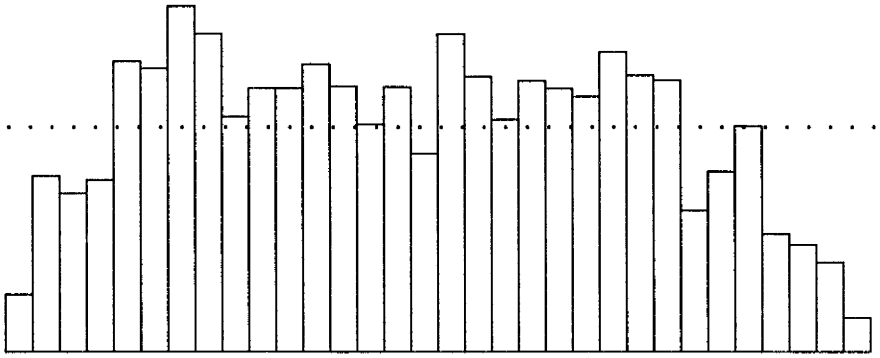
Average 50 Runs: 64 Nodes 20000 Updates Using Alg. 2



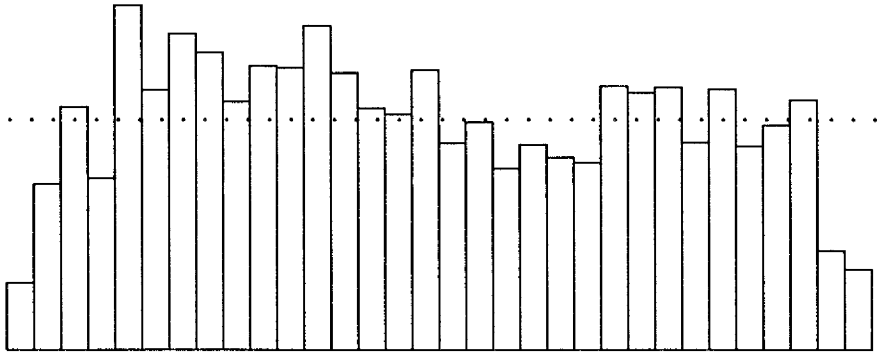
Average 50 Runs: 128 Nodes 30000 Updates Using Alg. 2



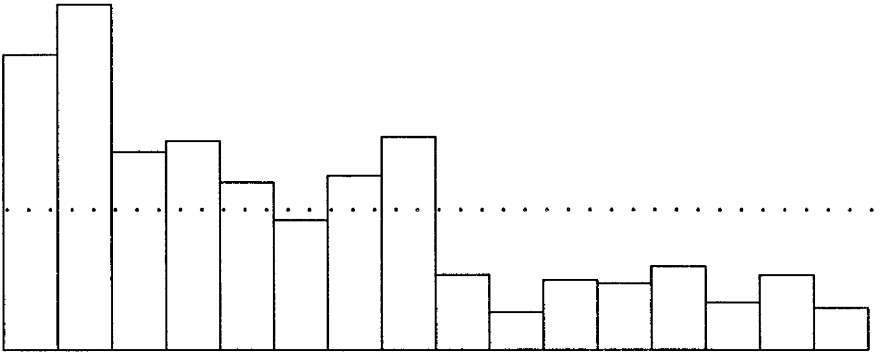
Average 50 Runs: 256 Nodes 150000 Updates Using Alg. 2



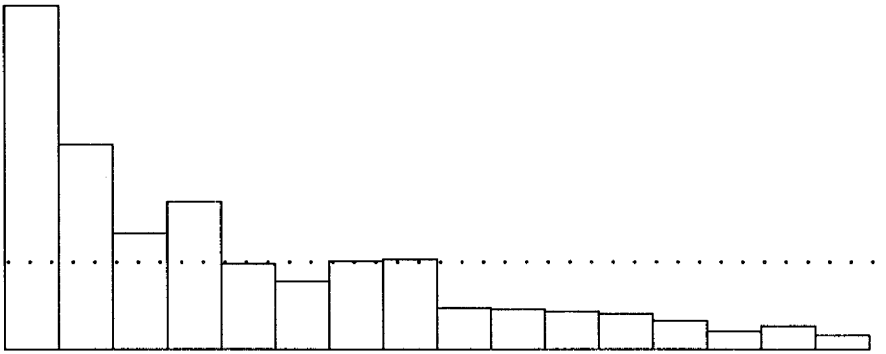
Average 50 Runs: 512 Nodes 750000 Updates Using Alg. 2



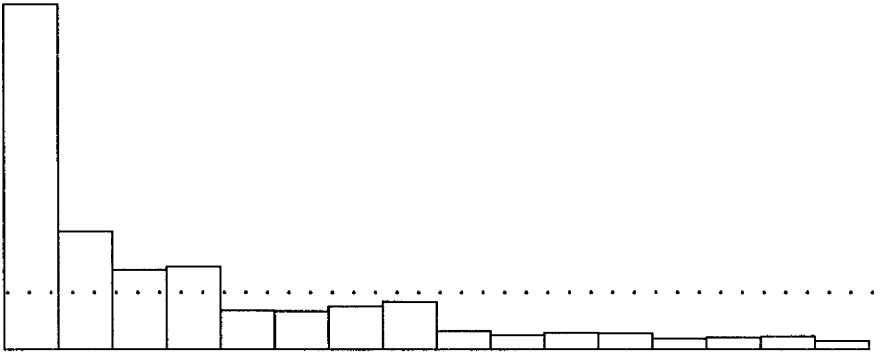
Average 50 Runs: 1024 Nodes 2000000 Updates Using Alg. 2



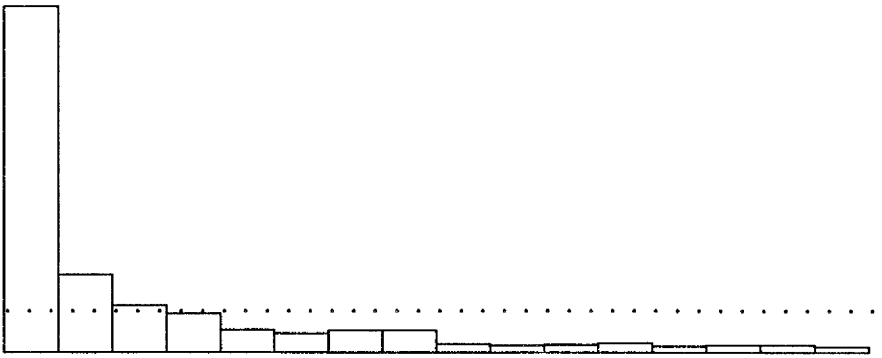
Average 50 Runs: 32 Nodes 7500 Updates Using Alg. 3



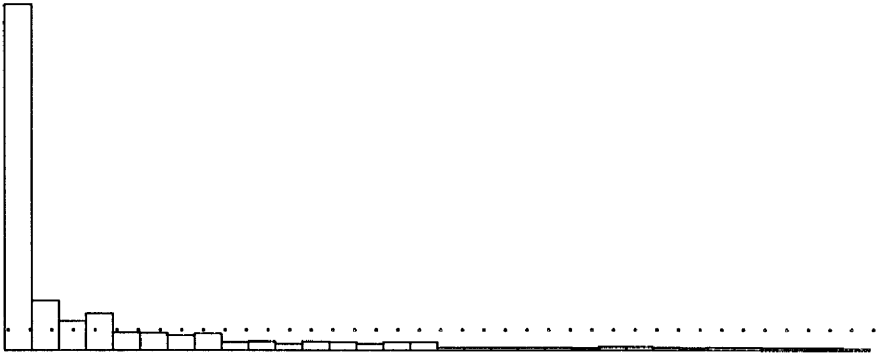
Average 50 Runs: 64 Nodes 50000 Updates Using Alg. 3



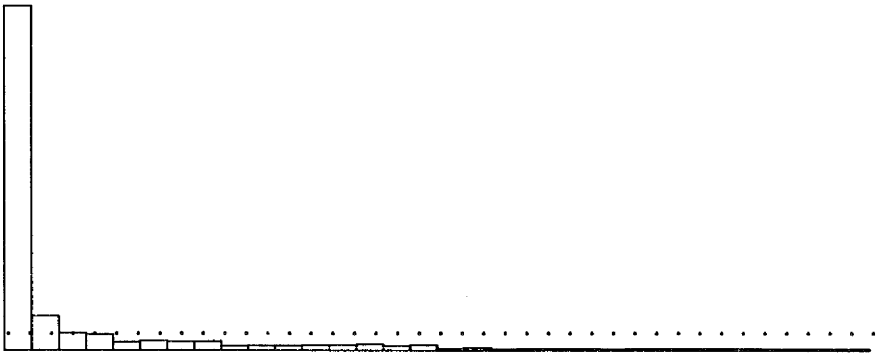
Average 50 Runs: 128 Nodes 80000 Updates Using Alg. 3



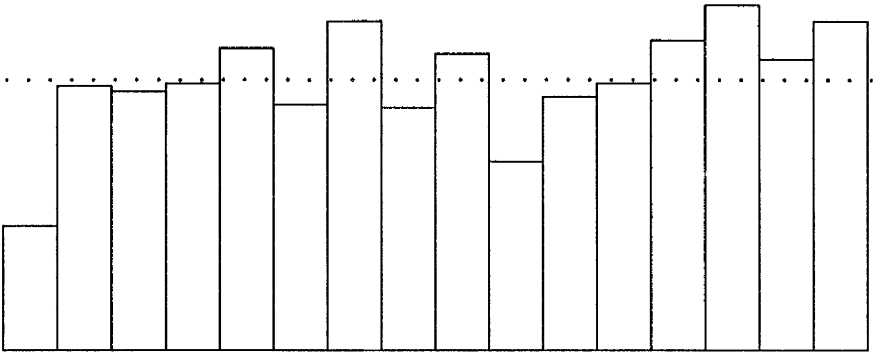
Average 50 Runs: 256 Nodes 150000 Updates Using Alg. 3



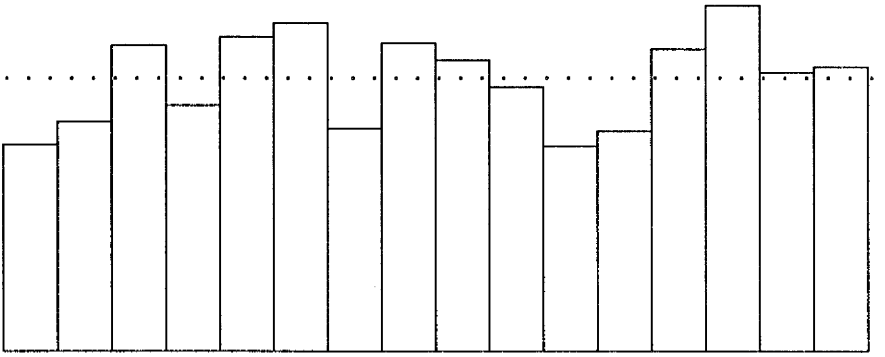
Average 50 Runs: 512 Nodes 500000 Updates Using Alg. 3



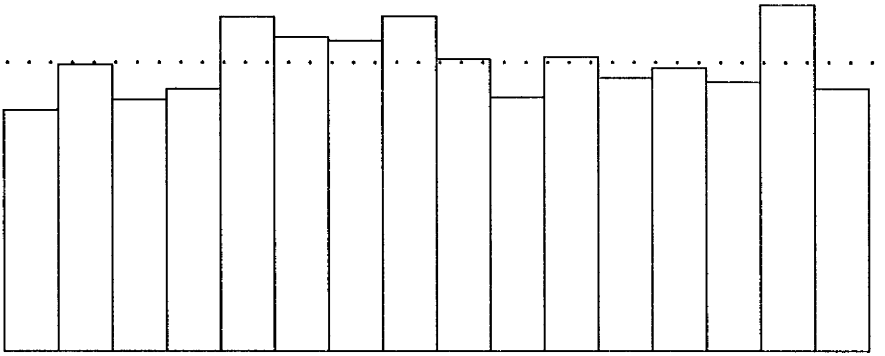
Average 50 Runs: 1024 Nodes 2000000 Updates Using Alg. 3



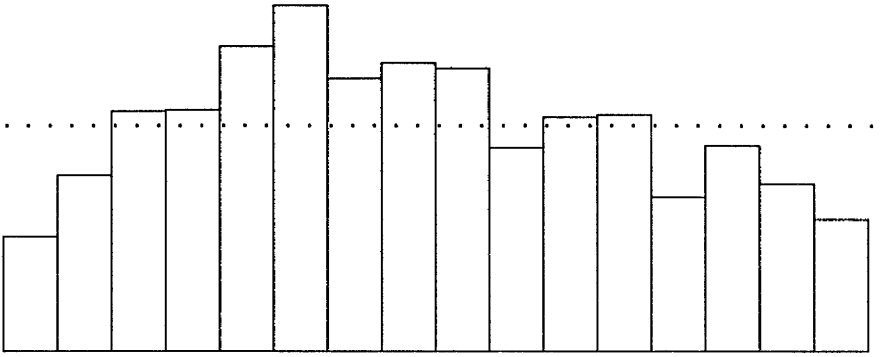
Average 50 Runs: 32 Nodes 7500 Updates Using Alg. 4



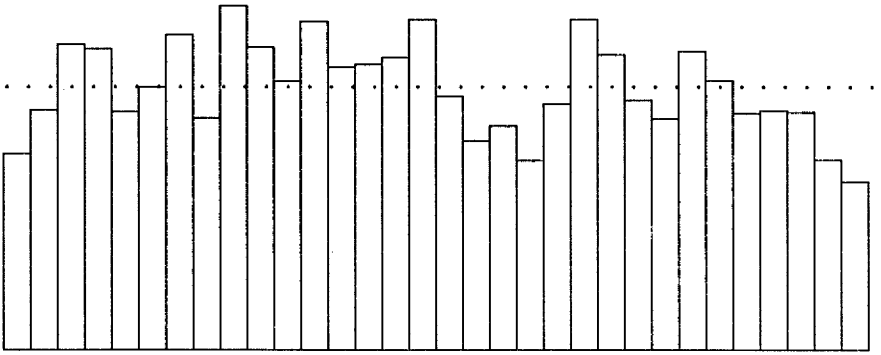
Average 50 Runs: 64 Nodes 50000 Updates Using Alg. 4



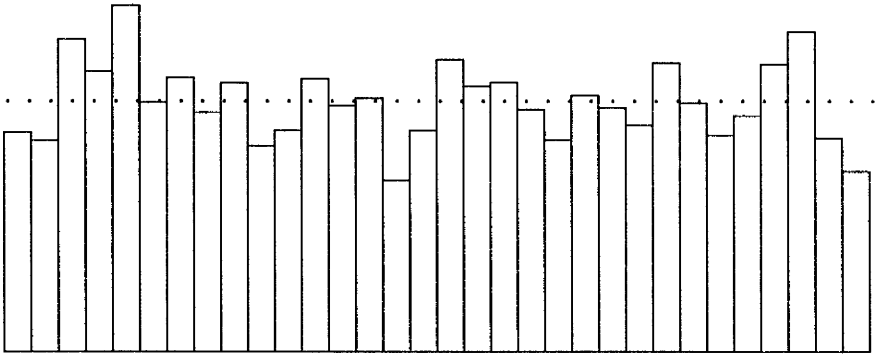
Average 50 Runs: 128 Nodes 100000 Updates Using Alg. 4



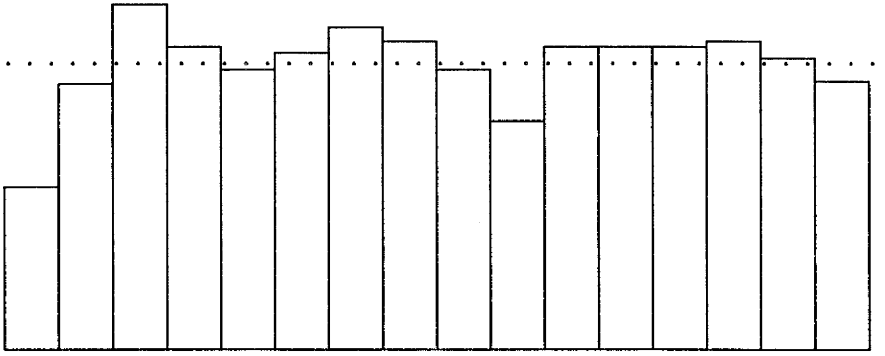
Average 50 Runs: 256 Nodes 200000 Updates Using Alg. 4



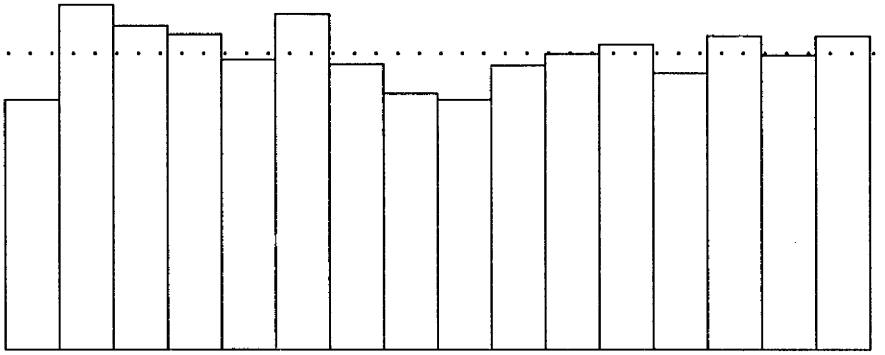
Average 50 Runs: 512 Nodes 800000 Updates Using Alg. 4



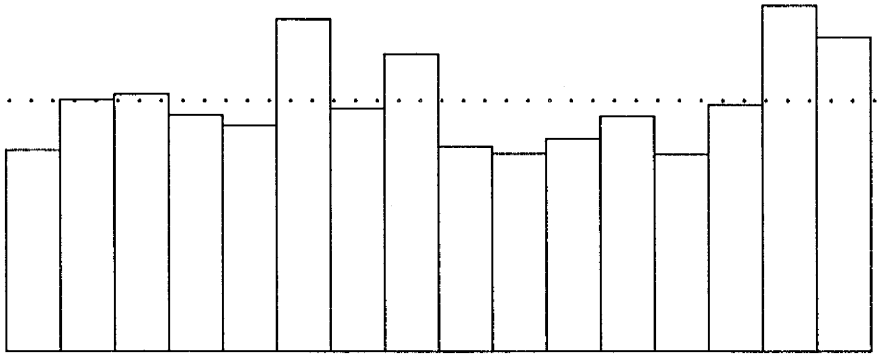
Average 50 Runs: 1024 Nodes 2000000 Updates Using Alg. 4



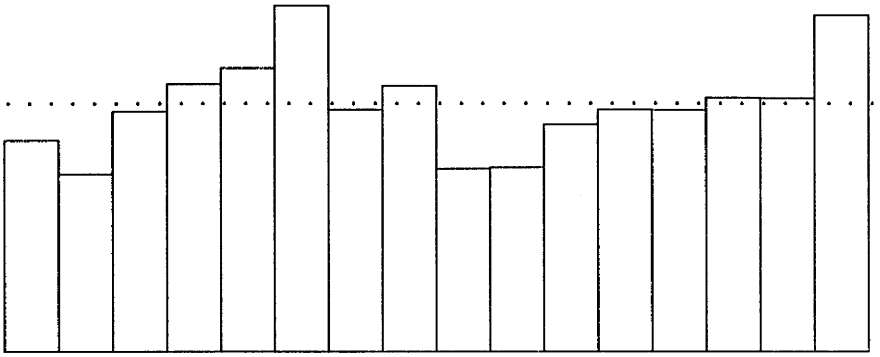
Average 50 Runs: 32 Nodes 0 Updates Using Alg. Y



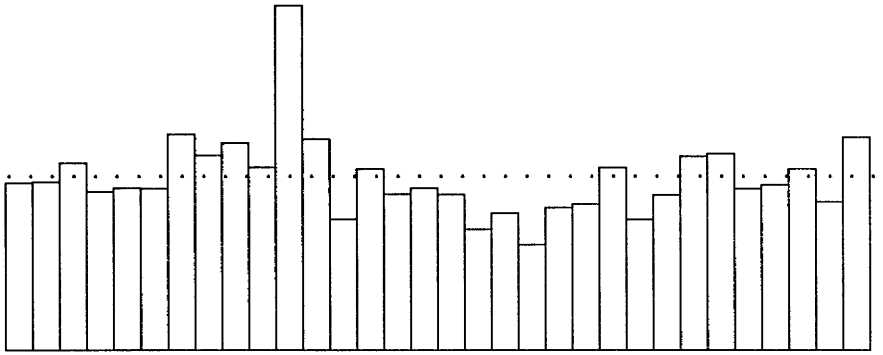
Average 50 Runs: 64 Nodes 0 Updates Using Alg. Y



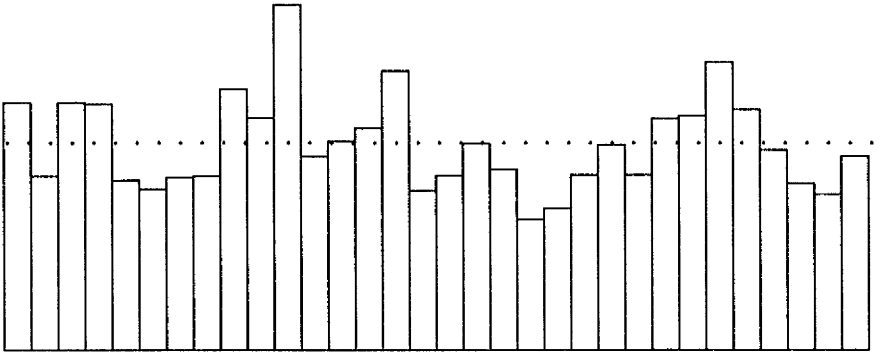
Average 50 Runs: 128 Nodes 0 Updates Using Alg. Y



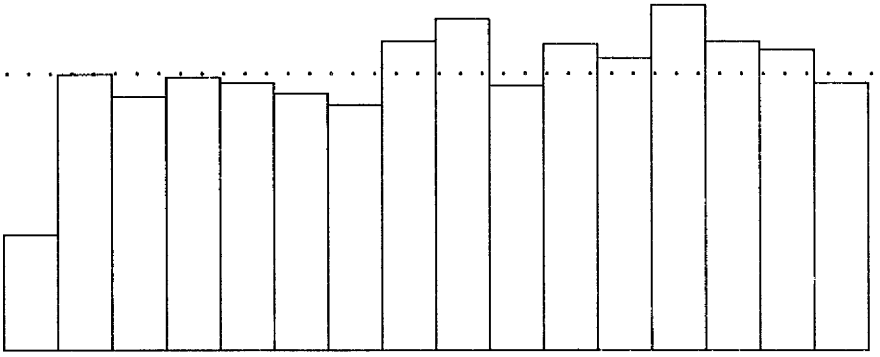
Average 50 Runs: 256 Nodes 0 Updates Using Alg. Y



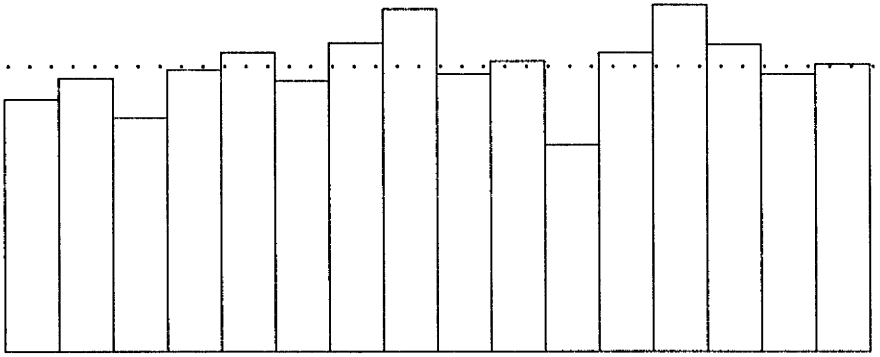
Average 50 Runs: 512 Nodes 0 Updates Using Alg. Y



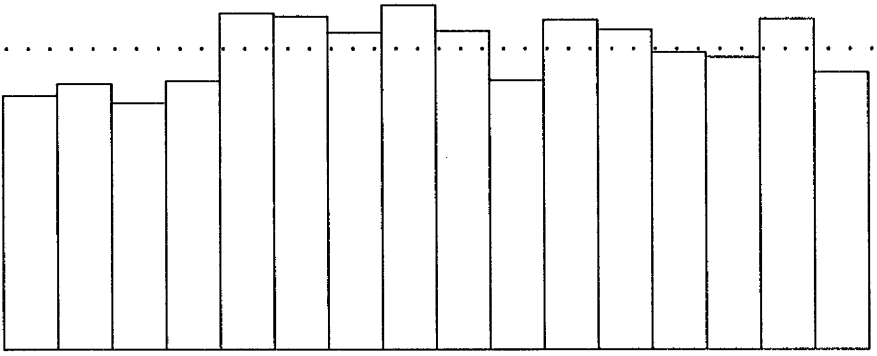
Average 50 Runs: 1024 Nodes 0 Updates Using Alg. Y



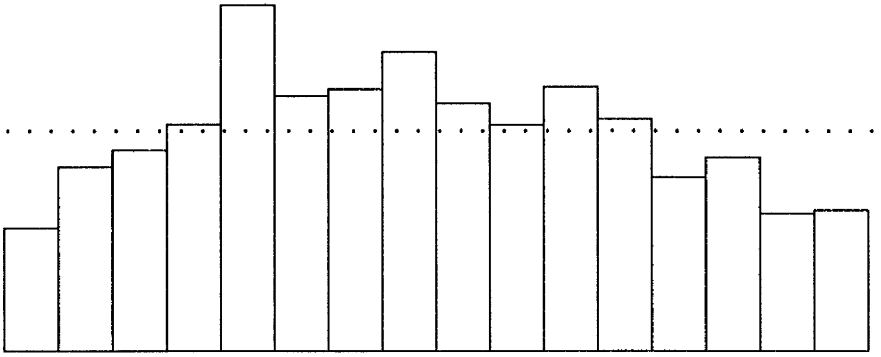
Average 50 Runs: 32 Nodes 7500 Updates Using Alg. 5



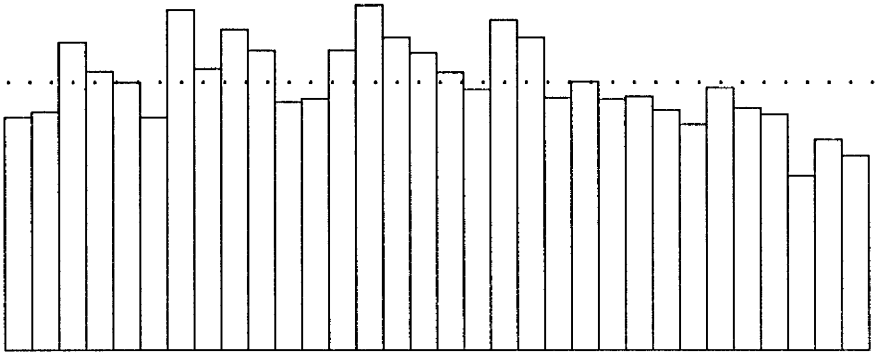
Average 50 Runs: 64 Nodes 50000 Updates Using Alg. 5



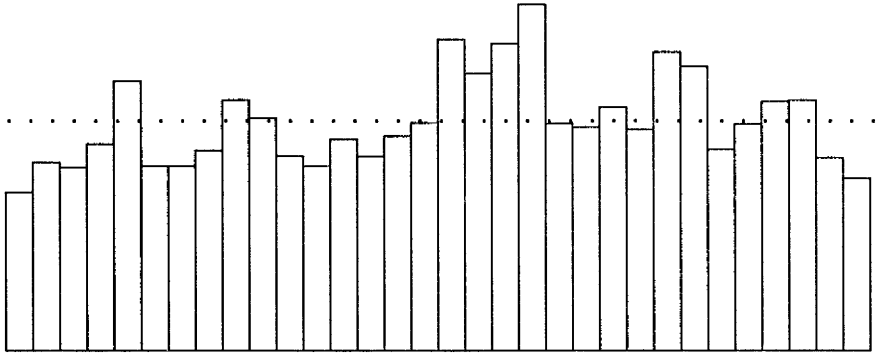
Average 50 Runs: 128 Nodes 100000 Updates Using Alg. 5



Average 50 Runs: 256 Nodes 200000 Updates Using Alg. 5



Average 50 Runs: 512 Nodes 800000 Updates Using Alg. 5



Average 50 Runs: 1024 Nodes 2000000 Updates Using Alg. 5

Graphs

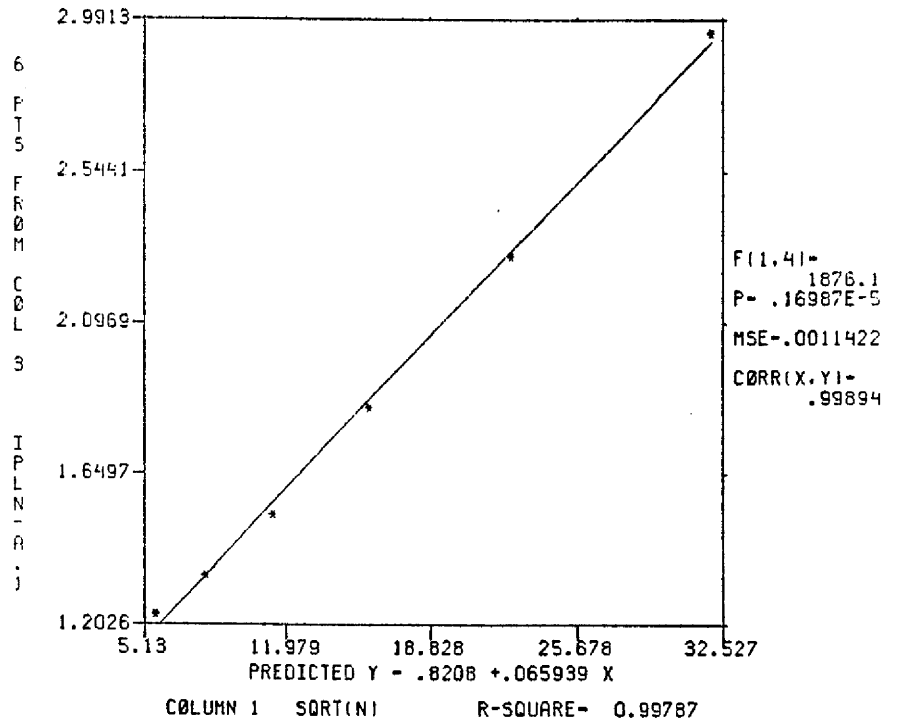
On the following pages are some graphs showing how algorithms 1 and 3 changed the trees throughout the updating process. First, there is a graph showing how the ratio of the internal path length to the expected initial path length corresponds to the square root of the number of nodes in the tree, for each of the two asymmetric algorithms. These plots were generated by a polynomial regression package. Note that the coefficients are not precisely those given in chapter five, which were obtained using a multiple regression package. There, the inverse of the variance was used as a weight for the average path length, while unweighted values were used here.

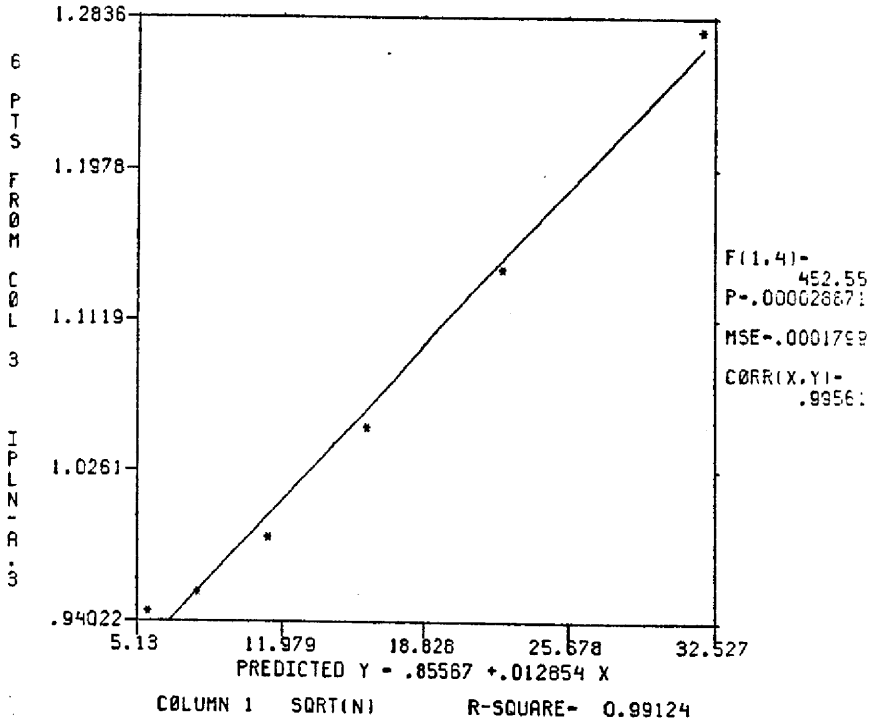
The next two graphs show the change in the average internal path length as the number of updates is increased. The path length is again given as a multiple of the expected initial length. The number of iterations has been divided by the square of the number of nodes, as it is in all the following graphs. Notice that there is virtually no initial improvement in the path length generated by algorithm 1, and that, for all trees, the length increases from the initial value. Algorithm 3, on the other hand, shows significant initial improvement, and the smaller trees seem never to get worse than the initial ones.

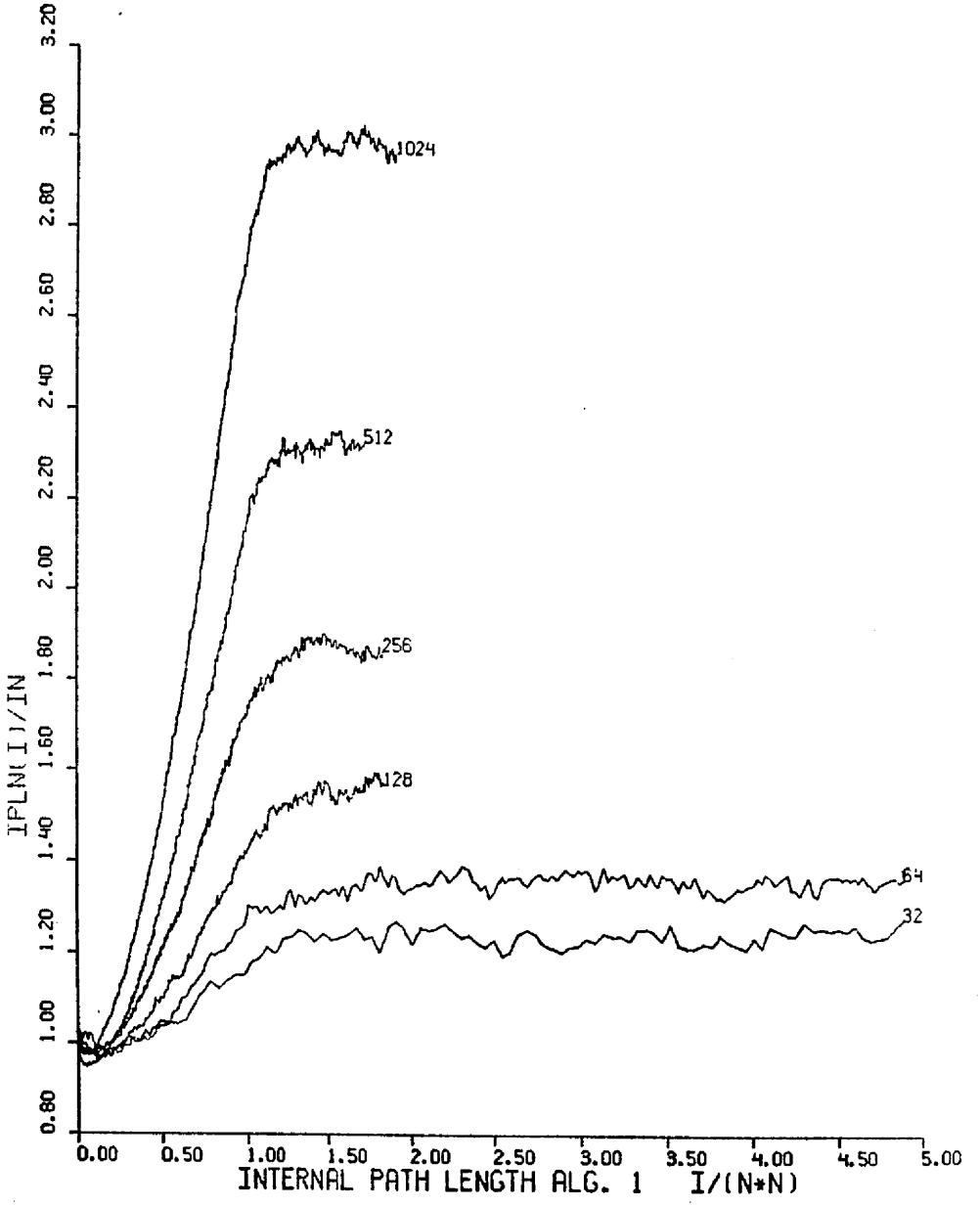
The changes in the lengths of the backbones as updates are performed using 1 and 3, are plotted next. The normalizing value Hn is the harmonic number, and as explained in chapter 4, it is the expected initial length of the backbone.

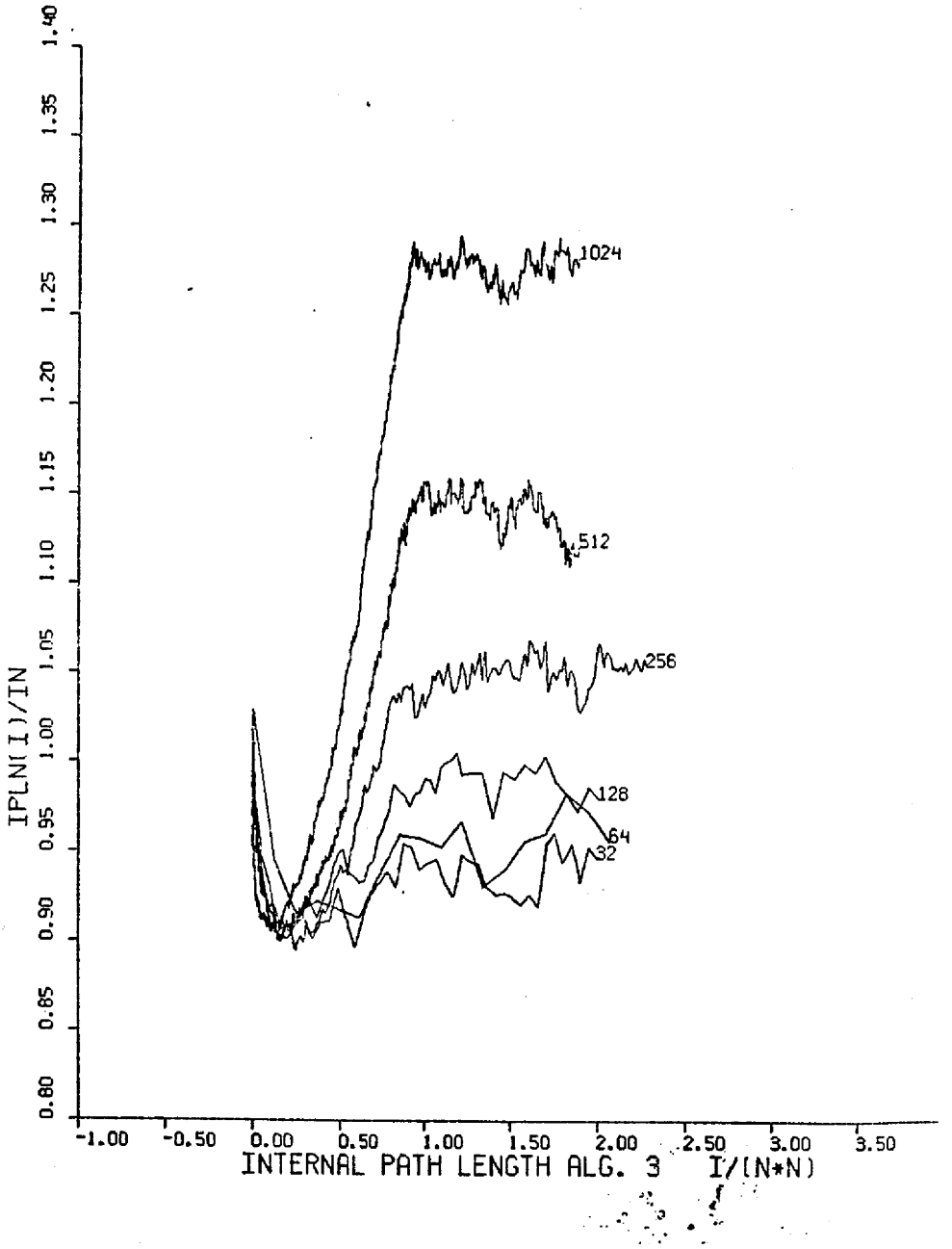
Finally, the changes in the relative balance and the relative skew (at $X=2$) for the two algorithms are graphed.

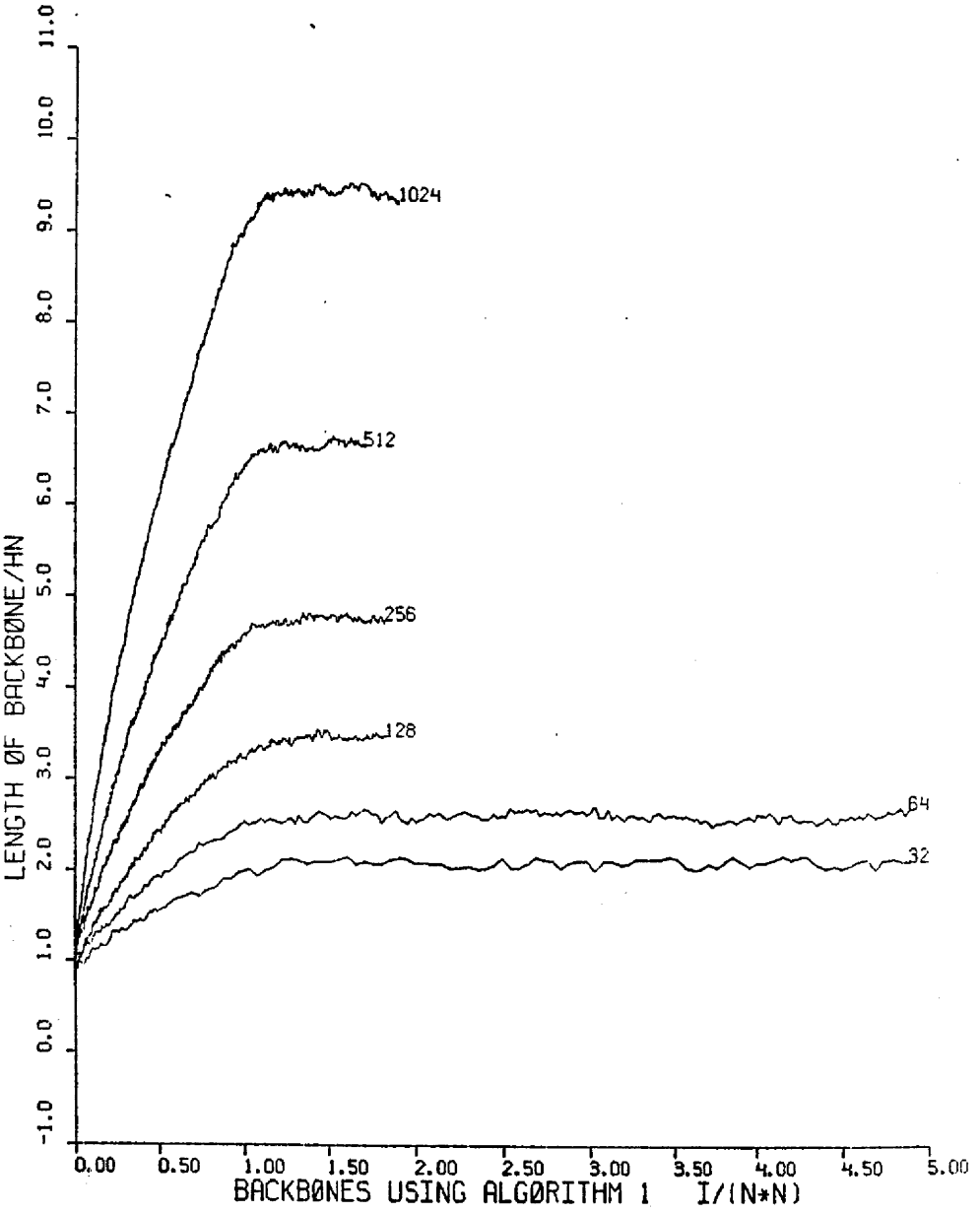
Similar graphs for the symmetric algorithms are not included, as in those the asymptotic changes tended to be small, while the variability of the trees was relatively large, which resulted in an undecipherable jumble of lines when plotted.

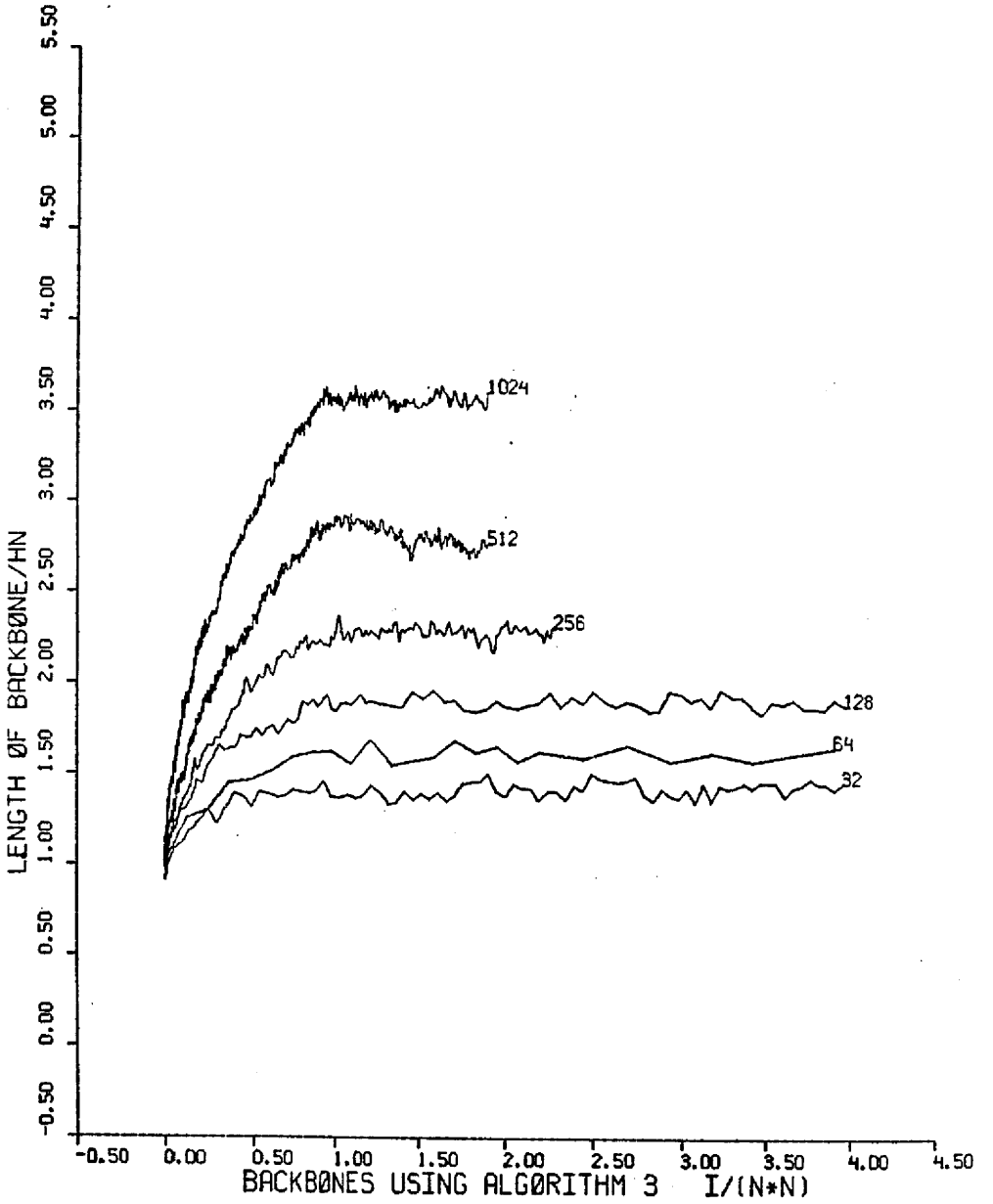

2.9913
2.5441
2.0969
1.6497
1.2026
6 PTS FROM COL 3 IPLN-A.J
5.13
11.979
18.828
25.678
32.527
PREDICTED Y = .8208 +.065939 X
F(1,4)=
1876.1
P= .16987E-5
MSE=.0011422
CORR(X,Y)=
.99894
COLUMN 1 SQRT(N)
R-SQUARE= 0.99787

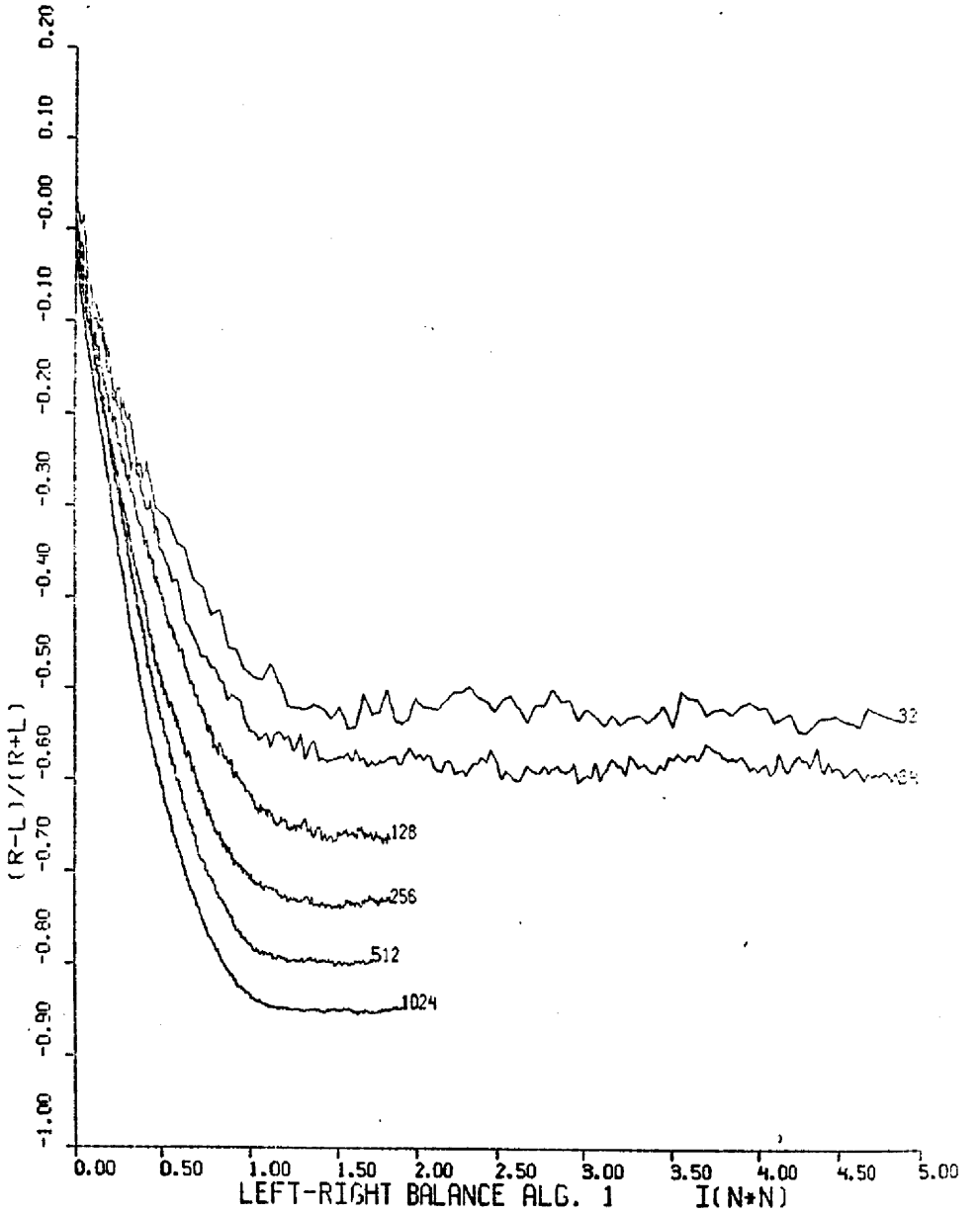

1.2836
1.1978
1.1119
1.0261
.94022
6 PTS FROM COL 3 IPLN-A:3
5.13
11.979
18.828
25.678
32.527
PREDICTED Y = .85567 +.012854 X
COLUMN 1 SQRT(N)
R-SQUARE= 0.99124
F(1,4)=
452.55
P=.00002887
MSE=.000179
CORR(X,Y)=
.9956

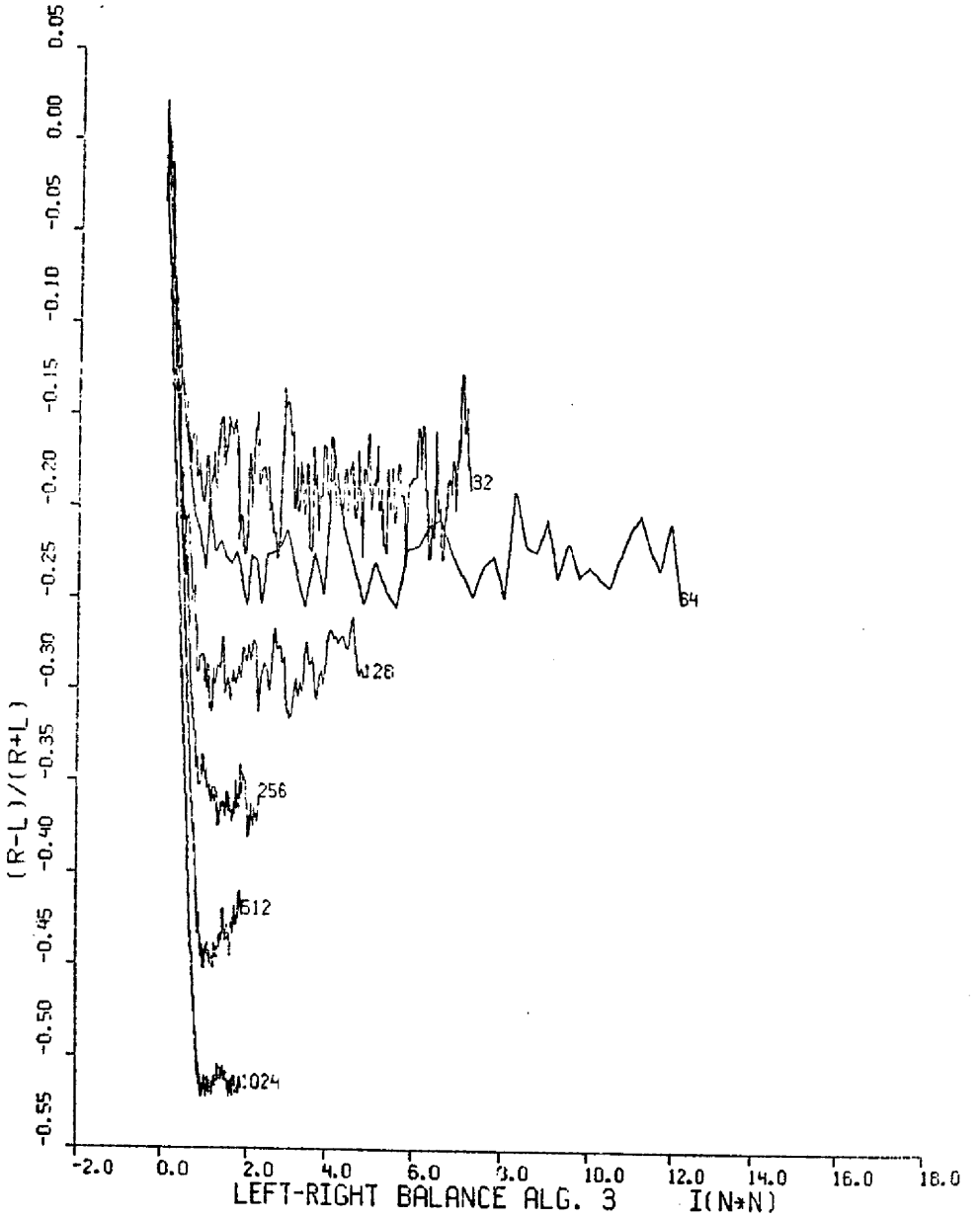

IPLN(I)/IN
0.80
1.00
1.20
1.40
1.60
1.80
2.00
2.20
2.40
2.60
2.80
3.00
3.20
1024
512
256
128
64
32
0.00
0.50
1.00
1.50
2.00
2.50
3.00
3.50
4.00
4.50
5.00
INTERNAL PATH LENGTH ALG. 1 I/(N*N)

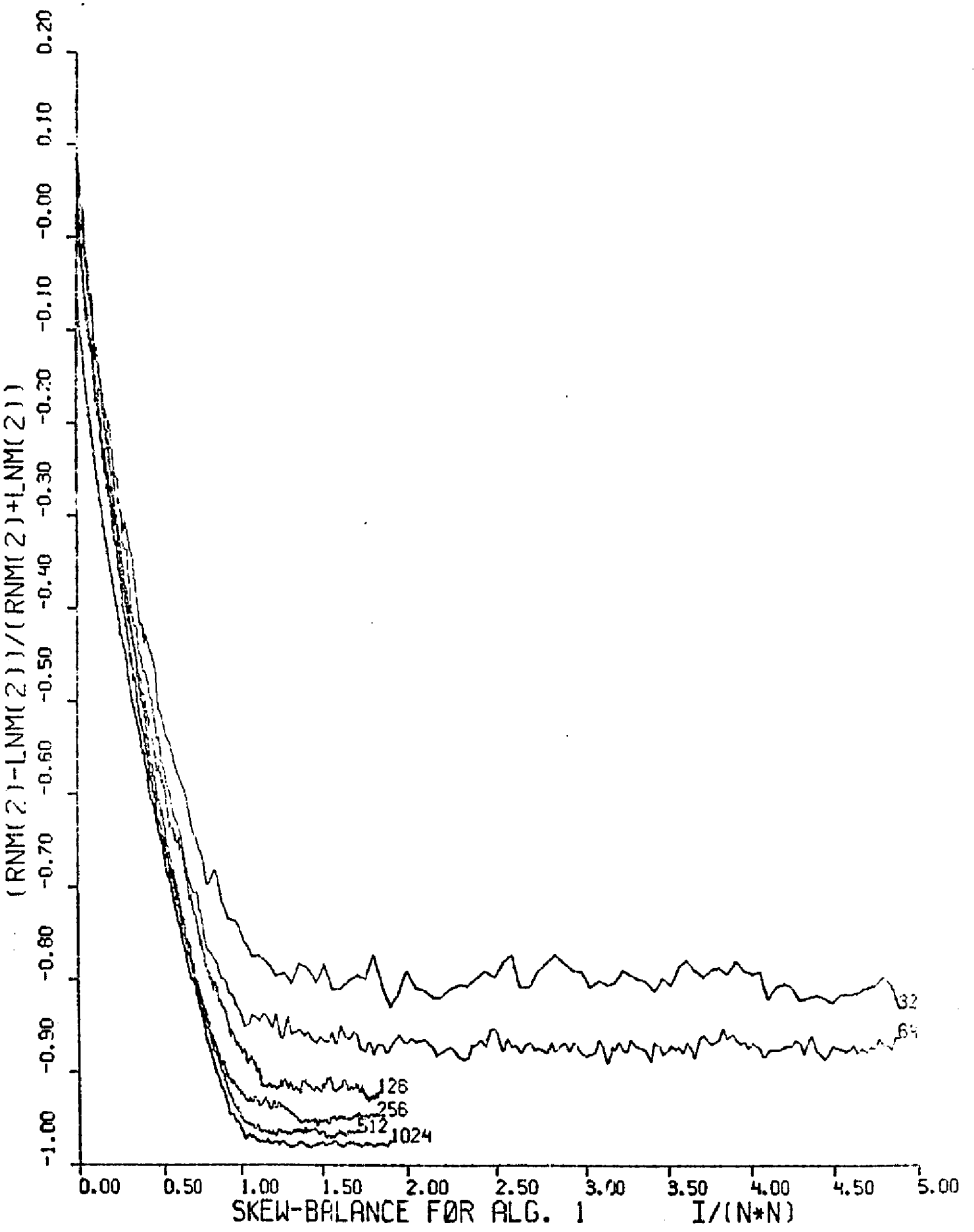

IPLN(I)/IN
0.80
0.85
0.90
0.95
1.00
1.05
1.10
1.15
1.20
1.25
1.30
1.35
1.40
1024
512
256
128
64
32
-1.00
-0.50
0.00
0.50
1.00
1.50
2.00
2.50
3.00
3.50
INTERNAL PATH LENGTH ALG. 3 I/(N*N)

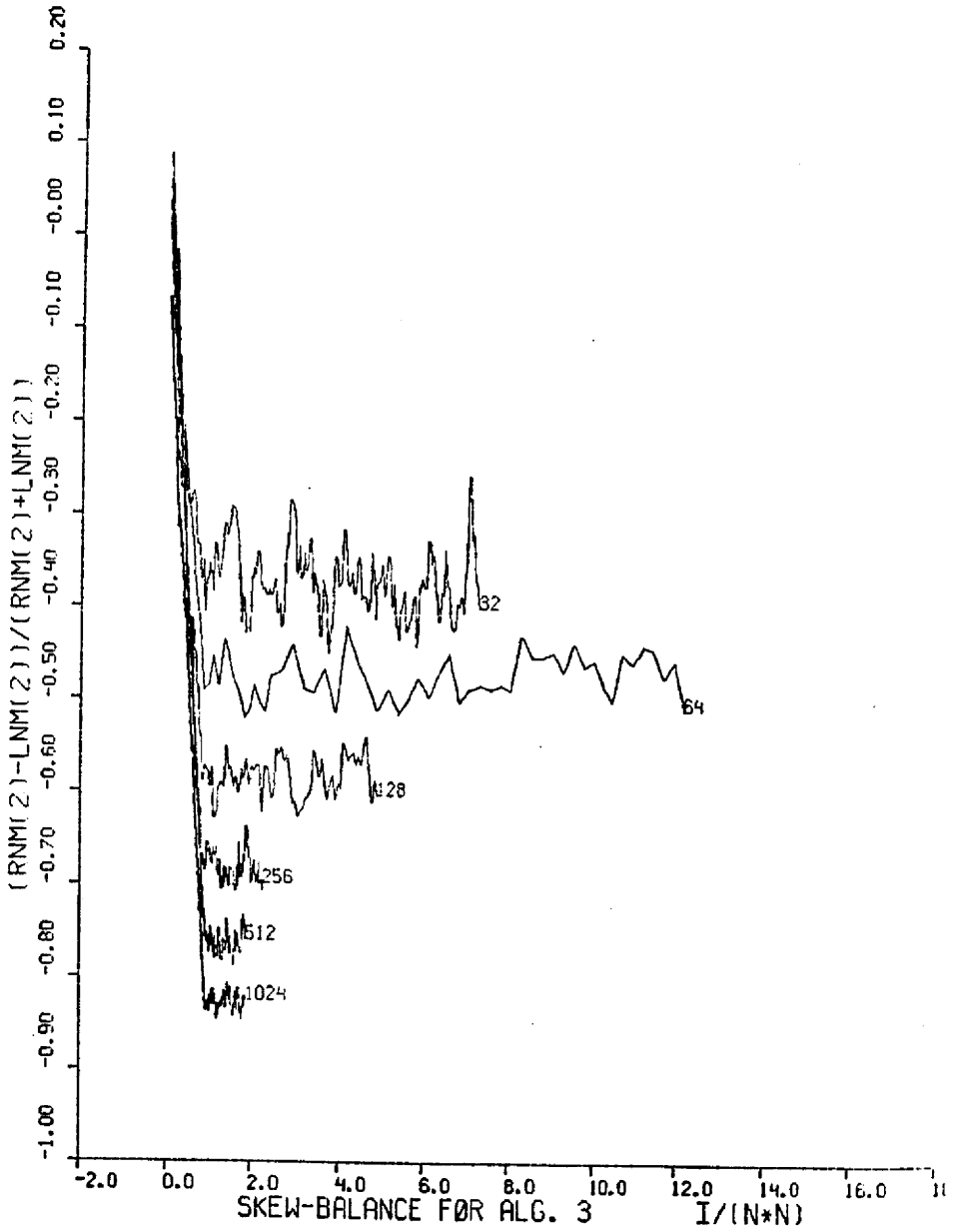

LENGTH ØF BACKBØNE/HN
-1.0
0.0
1.0
2.0
3.0
4.0
5.0
6.0
7.0
8.0
9.0
10.0
11.0
1024
512
256
128
64
32
0.00
0.50
1.00
1.50
2.00
2.50
3.00
3.50
4.00
4.50
5.00
BACKBØNES USING ALGØRITHM 1 I/(N*N)


LENGTH ØF BACKBØNE/HN
-0.50
0.00
0.50
1.00
1.50
2.00
2.50
3.00
3.50
4.00
4.50
5.00
5.50
1024
512
256
128
64
32
-0.50
0.00
0.50
1.00
1.50
2.00
2.50
3.00
3.50
4.00
4.50
BACKBØNES USING ALGØRITHM 3 I/(N*N)


0.20
0.10
-0.00
-0.10
-0.20
-0.30
-0.40
-0.50
-0.60
-0.70
-0.80
-0.90
-1.00
(R-L)/(R+L)
32
128
256
512
1024
0.00
0.50
1.00
1.50
2.00
2.50
3.00
3.50
4.00
4.50
5.00
LEFT-RIGHT BALANCE ALG. 1
I(N*N)


0.05
0.00
-0.05
-0.10
-0.15
-0.20
-0.25
-0.30
-0.35
-0.40
-0.45
-0.50
-0.55
(R-L)/(R+L)
32
64
128
256
512
1024
-2.0
0.0
2.0
4.0
6.0
8.0
10.0
12.0
14.0
16.0
18.0
LEFT-RIGHT BALANCE ALG. 3
I(N*N)


(RNM(2)-LNM(2))/(RNM(2)+LNM(2))
0.20
0.10
-0.00
-0.10
-0.20
-0.30
-0.40
-0.50
-0.60
-0.70
-0.80
-0.90
-1.00
0.00
0.50
1.00
1.50
2.00
2.50
3.00
3.50
4.00
4.50
5.00
SKEW-BALANCE FØR ALG. 1
I/(N*N)
128
256
512
1024


(RNM(2)-LNM(2))/(RNM(2)+LNM(2))
0.20
0.10
-0.00
-0.10
-0.20
-0.30
-0.40
-0.50
-0.60
-0.70
-0.80
-0.90
-1.00
32
64
128
256
512
1024
-2.0
0.0
2.0
4.0
6.0
8.0
10.0
12.0
14.0
16.0
SKEW-BALANCE FØR ALG. 3
I/(N*N)

References

[E] Jeffery L. Eppinger, *An Empirical Study of Insertion and Deletion in Binary Trees.* **CACM Vol 26 No. 9** Sept 1983

[H] Thomas N. Hibbard, *Some Combinatorial Properties of Certain Trees with Applications to Searching and Sorting.* **Journal of Association of Computing Machinery** 9(1):13-28, Jan. 1962

[K] Gary D. Knot, *Deletion in Binary Storage Trees.* Ph.D thesis, Stanford University, May 1975 STAN-CS-75-491

[JK] Arne T. Jonassen and Donald E. Knuth, *A Trivial Algorithm Whose Analysis Isn't.* **Journal of Computer and System Sciences** 16, pp 301-322, (1978)

[K1] Donald E. Knuth, *The Art of Computer Programming.* Volume I: *Fundamental Algorithms.* Addison-Wesley, 1968

[K2] Donald E. Knuth, *The Art of Computer Programming.* Volume II: *Seminumerical Algorithms.* Addison-Wesley, 1969

[K3] Donald E. Knuth, *The Art of Computer Programming.* Volume III: *Searching and Sorting.* Addison-Wesley, 1973