# INVESTIGATION OF A NEW CLASS OF BINARY SEARCH TREE INSERTION ALGORITHMS

by

Robert M. MacGregor

Memorandum No. ERL-M523

June 1975

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

# INVESTIGATION OF A NEW CLASS OF BINARY SEARCH TREE INSERTION ALGORITHMS[†]

Robert M. MacGregor

Computer Science Division
Department of Electrical Engineering and Computer Sciences
and the Electronics Research Laboratory
University of California at Berkeley

June 1975

## Abstract

This paper introduces a new class of algorithms for insertion into a binary search tree. Called "compact-tree" algorithms, they are defined by the property that restructuring of the tree occurs only when the tree's overall height exceeds a specified bound, which is a function of the size of the tree.

First the algorithms are described in detail. Then the results of several empirical tests are discussed. The best of the compact-tree algorithms compares in some ways favorably, and in others unfavorably with the Adel'son-Velskii and Landis [1, p. 451] algorithm. Finally we have a couple of theoretical results, the principal one being that if keys are inserted into a tree in increasing (or decreasing) order, using a compact-tree algorithm, then the running time is proportional to $N \log N$, where $N$ is the number of keys inserted.

[†]Research sponsored by National Science Foundation Grant GJ-43227X.

## Description of Compact Trees

A binary search tree is a data structure which stores records in such a fashion that an individual record can be located relatively quickly. Each record is presumed to have a field containing a key which uniquely identifies that record. Given a binary search tree with $n$ nodes (records) and the key of some node in the tree, the corresponding record can generally be found in $O(\log_2 n)$ steps. Much of a binary search tree's value lies in the fact that a new node can be inserted into the tree, or an existing one deleted, with relative ease.

The keys of all records are embedded in a total ordering. A binary search tree is an extended binary tree containing keyed records such that, if the nodes are visited in symmetric order (post order), then the keys will be encountered in increasing order. Each node contains pointers to two sons, one or both of which may be null, and usually there is a header node which points to the root node of the tree.

Let P be a pointer to a node in the tree which is non-null. Then P.KEY denotes the key for that node, P.LLINK is a pointer to its left son, and P.RLINK is a pointer to its right son. The general procedure to find a node, given its key K, is as follows:

1) Set P equal to the root node.

2) IF K = P.KEY THEN stop, we have found the record.

3) IF K < P.KEY THEN P ← P.LLINK ELSE P ← P.RLINK.

4) IF P is non-null then go to step 2. Otherwise stop, because there is no record in the tree with key K.

We see that a pointer must travel down the tree to find a record. The distance it travels is bounded by the height (maximum depth) of the tree. Hence it is desirable to keep the height as small as possible.

Another measure is the "total path length" of a tree, which is the sum of the root-node distances. If we assume that nodes are accessed with equal probability, then the value $\frac{\text{total path length}}{\text{number of nodes}}$ gives us the average number of comparisons to find a node in the tree.

It would be nice to construct search trees which always have a minimal total path length. However, it can be shown that under repeated insertion or deletion of nodes, the amount of restructuring necessary to maintain minimality is prohibitive. Hence, we look for algorithms which can construct near-optimal trees. We will first look at the problem of inserting a node into a search tree.

The basic tree-insertion algorithm is as follows:

We wish to insert a record with key K into a non-empty tree.

1) Set P equal to the root node.

2) If K < P.KEY then P ← P.LLINK.

   If K > P.KEY then P ← P.RLINK.

   If K = P.KEY then the record is already in the tree, so we stop.

3) If P is non-null repeat step 2.

4) Otherwise, P is null so we have found a leaf. Create a node Q, set Q.KEY ← K, Q.LLINK ← Λ, Q.RLINK ← Λ (Λ is the null pointer) and place Q in this leaf. (This involves remembering where the father node is, and setting a pointer there to point to node Q.)

This algorithm does no restructuring, and the trees it grows have a total path length which can be very large if the order in which the keys are inserted is poor. In a worst case the keys may appear in ascending or descending order, and a completely degenerate tree (no node has two non-null sons) results. Several algorithms have been developed which eliminate this problem (see [2]) by restructuring the tree as

more nodes are inserted.

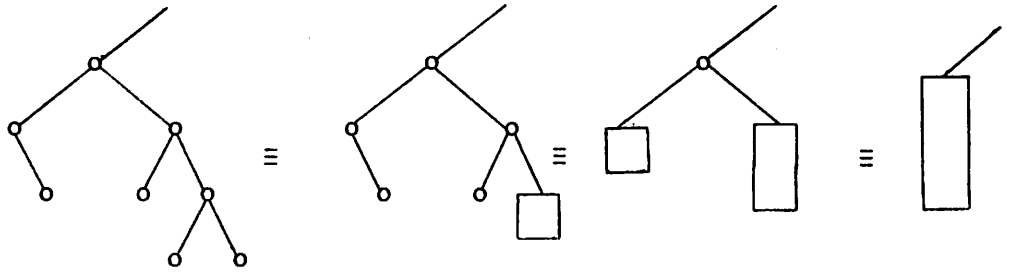
Possibly the most successful of these algorithms is the AVL algorithm invented by Adel'son-Velskii and Landis [1, pp. 451-469]. After inserting a node, it restructures the tree when necessary in order to keep the tree "AVL balanced," which means that for any node in the tree, the absolute difference between the heights of its left and right subtrees never exceeds one. It has been empirically observed that the restructuring, called rebalancing, occurs after about 50% of the insertions. Because rebalancing is fairly easy, this overhead is not excessive.

A new idea has been proposed which hoped to reduce this overhead by restructuring less frequently. The basic idea is as follows: Given a tree of size $n$, we devise a function $h(n)$ which is the maximum height which we will allow for a tree of $n$ nodes. If an insertion causes the height to exceed $h(n)$ we locally restructure the tree so that the overall height is again not more than $h(n)$ (note that $n \leftarrow n+1$ occurred upon insertion).

There are many ways in which this idea could be realized. It was the intention of this research to investigate and empirically test the more promising of these algorithms.

When describing tree operations it is necessary to develop some kind of shorthand representation for the trees. I shall use a form found in Knuth, Vol. III [1] which substitutes a rectangle for a subtree, and varies the lengths of the rectangles in proportion to the heights they represent. For convenience I shall omit specification of the null nodes.

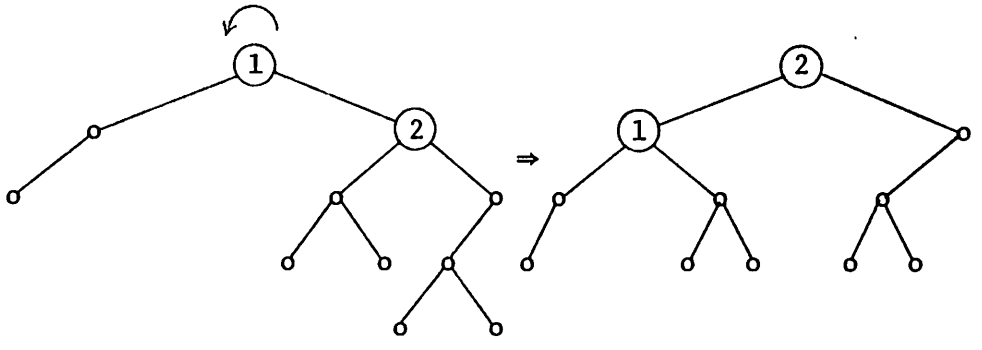
Example.



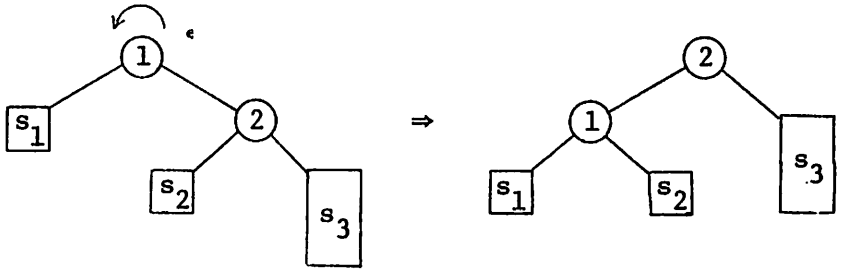
These represent successively more abstracted nodes of the same subtree.

The basic tree operations which we use to restructure a tree are called rotations. They come in four flavors: left single, left double, right single, and right double. They will be described by example.

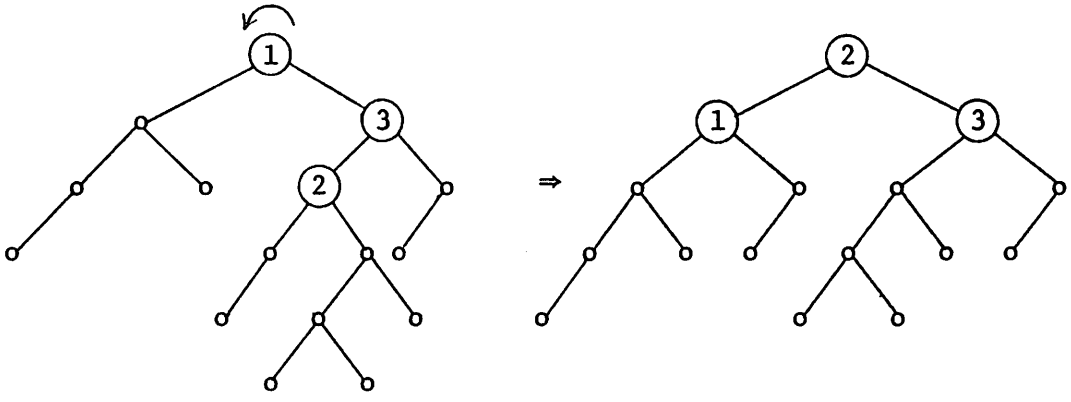
Left Single Rotation (at node 1)



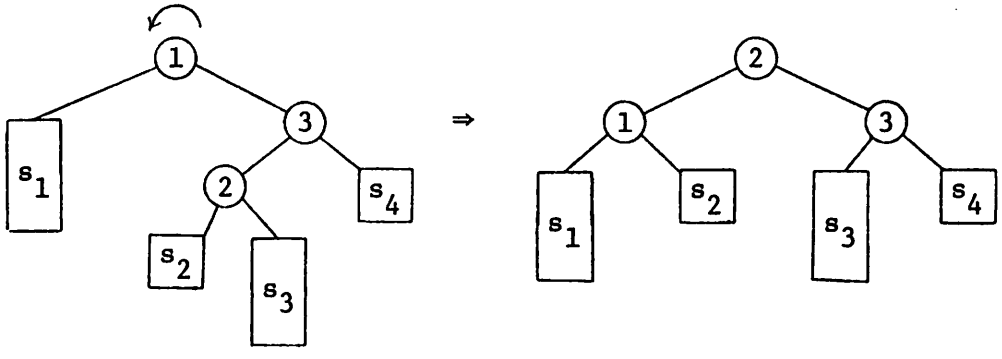

The general mechanism is clearer in the abstracted versions:



Left Double Rotation (at node 1)




or equivalently




The right rotations are mirror images of the corresponding left rotations. Notice that the left double rotation above is equivalent to performing a right single rotation at node 3, followed by a left single rotation at node 1. As seen in the examples, the rotations can sometimes be used to decrease the height of a subtree.

Let sub(p) denote the subtree rooted at node p. The following three conditions are necessary and sufficient for the existence of a rotation which reduces the height of sub(p):

Let sub(p) have height k.

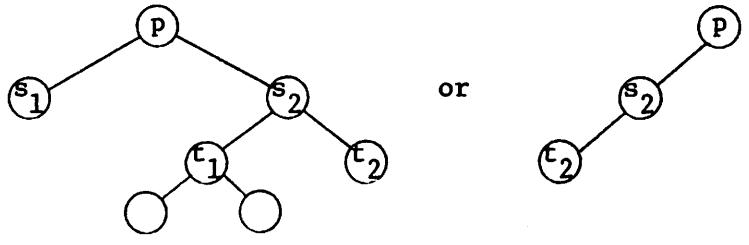
(i) p has two sons $s_1$ and $s_2$, and $s_2$ has two sons $t_1$ and $t_2$ ($s_1$ and/or $t_1$ may be null).

(ii) sub($s_1$) has height $\leq k-3$.

(iii) $sub(t_1)$ has height $\leq k-2$.

These conditions imply that $sub(t_2)$ has height $k-1$.

Example.

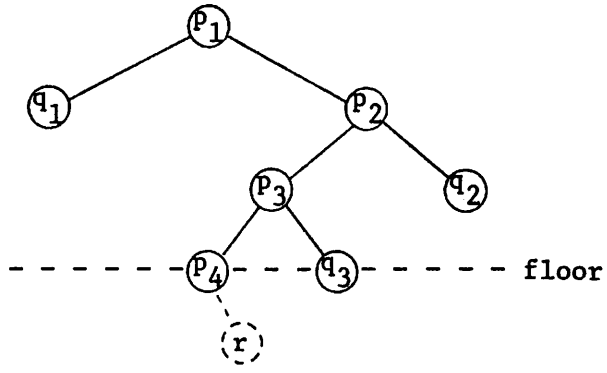



In the first instance above a left double rotation shortens the subtree, and in the second a right single does the trick.

We will refer to the deepest level of the tree which contains non-null nodes as the "floor" of the tree (the trees have the root on top and the floor on the bottom). A subtree "reaches the floor" if it has non-null nodes on that level.

Suppose a node $r$ is inserted into a tree and the height of the tree increases as a result. Let $p_1,p_2,\ldots,p_k,p_{k+1} = r$ be the nodes on the path from the root $p_1$ to $r$. Then $p_{j+1}$ is a son of $p_j$ for $j = 1,2,\ldots,k$. Let the "other son" of each $p_j$ be labelled $q_j$ ($p_k$ has none). Then a rotation at node $p_j$ exists which will shorten the tree if and only if $sub(q_j)$ doesn't reach the level which was the floor before $r$ was inserted. (In the example a left double rotation at $p_1$ or a right single rotation at $p_2$ will shorten the tree.) This suggests a method for reducing the height of a tree after an insertion.

Example




"Let MAXHT be the current height of the tree (a tree with one non-null node has height one). Insert node r. If the length of the path from the root to r equals MAXHT then the height has increased. In this case, look for a node $q_j$ as defined above such that sub($q_j$) didn't reach the floor. If we find one, then perform the appropriate rotation at $p_j$."

Our original aim was to reduce the amount of time spent restructuring the tree, and hence we will replace MAXHT with a function h(n) of the size of the tree, whose value determines the lowest level at which a node can be inserted without the occurrence of some restructuring. This still leaves a lot unspecified, including what we should do if a node $q_j$ with the desired properties cannot be found.

Relative to some fixed function h(n), a tree with n nodes is called "compact" if its height does not exceed h(n). For example, AVL trees are guaranteed to be compact relative to a function $h(n) \approx 1.44 \log_2 n$. The algorithms which perform restructuring only to restore the property of compactness we will call "compact tree algorithms." They don't perform any restructuring as long as the tree is compact.

We will first derive an expression for h(n). The height of an n-node tree with minimal total path length is $\lceil \log_2 n+1 \rceil$. Looking at

the inverse function, we see that for a minimal-tpℓ tree of height $k$, the number of nodes $n$ obeys the inequality $2^{k-1} < n+1 \leq 2^k$. Hence, if $n$ is increased by insertion, the height increases only if $n+1 > 2^k$, or $n > 2^k - 1$. Since we seek only to construct near-optimal rather than optimal trees, we will relax this bound, and instead allow the tree's height to increase whenver $n > a(b^k) - c$ becomes true, for some constants $a$, $b$ and $c$, $1 < b \leq 2$. Values for $a$, $b$ and $c$ can be found experimentally. Setting $h^{-1}(k) = a(b^k) - c$ we obtain $h(n) \approx \log_b n$.

We are now in a position to present the simplest of the compact tree algorithms, which we refer to as "compact-streamlined" (C-SL). Assume values for $a$, $b$ and $c$ have been chosen, so that the height function $h(n)$ is defined.

Compact-Streamlined Algorithm: A node is inserted using the basic tree-insertion algorithm. Pointers to nodes $p_1, p_2, \ldots, p_k$ are stacked as we travel the path to the point of insertion. $n \leftarrow n+1$ reflects the increase in the size of the tree. If $k \geq h(n)$ we must attempt to decrease the height of the tree. In this case we backtrack by unpopping the stack, enabling us to locate the nodes $q_{k-1}, q_{k-2}, \ldots, q_1$. At each node $q_j$ we perform a preorder search of $sub(q_j)$ to determine if it reaches the floor. If it doesn't, we rotate at $p_j$, reducing the height of the tree to the value it had before insertion. Otherwise we continue backtracking, unless $j = 1$. In this case we have failed to reduce the tree's height, and the tree ceases to be compact.

For this particular algorithm we choose not to let this bother us. In fact, because the size of $sub(q_j)$ tends to increase exponentially

as $j$ decreases, it is best to place a limit upon the distance we are allowed to backtrack. This increases the chance of failure. However, by careful choice of parameters we can limit the rate of failure, and produce a tree whose height is usually within the bounds we set.

Suppose we wish to guarantee that the search tree we are growing is always compact. An algorithm more complicated than the one just given exists which will guarantee compactness, using a height function $h(n) \approx 1.44 \log_2 n$.

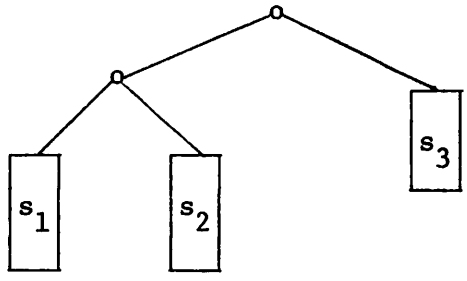
First we will describe a subroutine which is used to reduce the height of a tree (or subtree) by executing an appropriate sequence of rotations. Define a function $f(k)$ by

$$f(k) = \begin{cases} 0 & \text{if } k = 0 \\ 1 & \text{if } k = 1 \\ f(k-1) + f(k-2) + 1 & \text{if } k > 1 \end{cases}$$

Subroutine SHORTEN will reduce the height of a tree by one if it has height $k$ and obeys the inequality $n < f(k)$, where $n$ is the number of non-null nodes. We will show how to "shorten" any tree with height $k$ and $n = f(k) - 1$ nodes:

Trees with $f(1) - 1 = 0$ and $f(2) - 1 = 1$ nodes have heights 0 and 1 respectively, and hence already have height less than the value of f's argument. Otherwise we can assume $k \geq 3$ and $n \geq 3$.

Suppose both subtrees of the root have height $k - 1$. We know that one of the subtrees of the root must have less than half of the nodes, and since $f(k+1) \geq f(k)$ and $f(k) - 1 = f(k-1) + f(k-2)$ we know that it has less than $f(k-1)$ nodes. Hence we can recursively call SHORTEN to reduce its height to become $k - 2$. This gives us the figure (barring symmetries).




We will let "$s_i$" denote both a subtree and the number of nodes in that subtree. $s_1$ or $s_2$ and $s_3$ have height $k-2$.

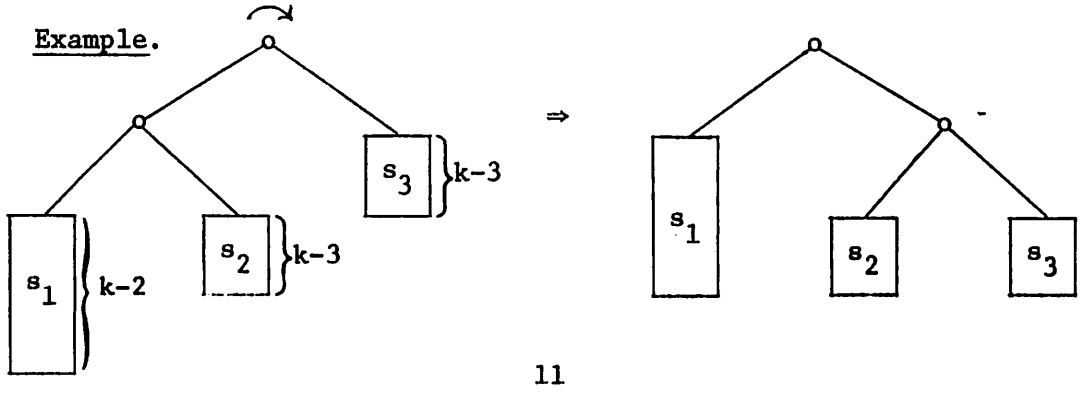
Suppose $s_3 \geq f(k-2)$. Then

$$\begin{aligned} s_1 + s_2 + 1 &= n - s_3 - 1 \\ &= f(k-1) + f(k-2) - (s_3+1) \\ &\leq f(k-1) - 1 \quad . \end{aligned}$$

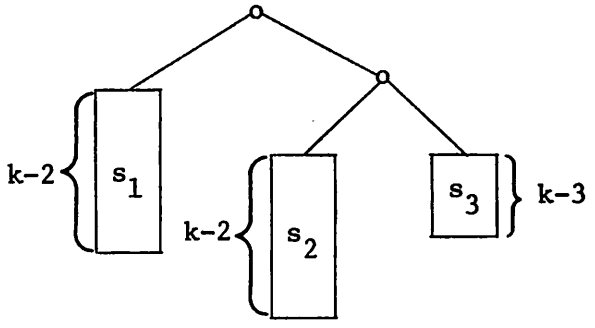
In this case we recursively call SHORTEN to reduce the height of the left (in this diagram) subtree.

Otherwise $s_3 < f(k-2)$, enabling us to call SHORTEN and reduce its height to $k-3$. If in fact $s_1$ has height $< k-2$ a right double rotation at the root will reduce the overall height to become $k-2$. If $s_2$ has height $< k-2$ a right single rotation at the root will do the same thing. If $s_1 < f(k-2)$ or $s_2 < f(k-2)$ then we call SHORTEN to reduce the height of one of these two subtrees, and then rotate at the root.

Example.



The remaining case has $s_3 < f(k-2)$, $s_1 \geq f(k-2)$ and $s_2 \geq f(k-2)$. Let $s_1 = f(k-2) + d$, $d \geq 0$. We perform a right single rotation at the root to get




Observe that

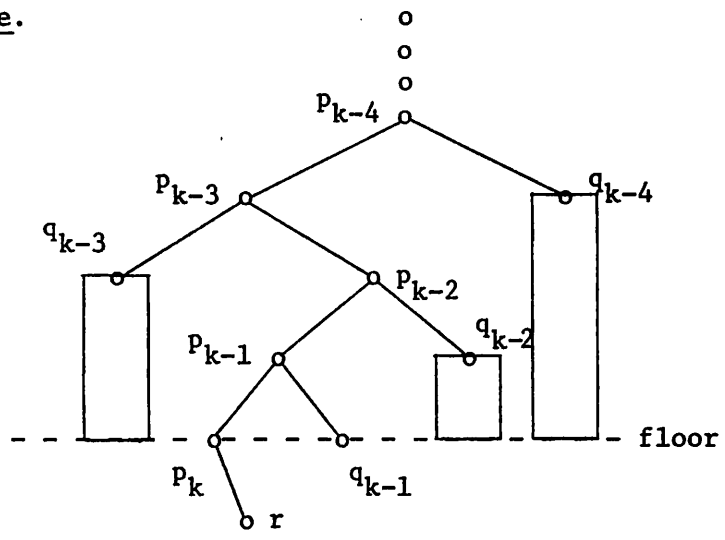
$$\begin{aligned} s_2 + s_3 + 1 &= n - s_1 - 1 \\ &= f(k-1) + f(k-2) - f(k-2) - d - 1 \\ &= f(k-1) - d - 1 \\ &< f(k-1) \ . \end{aligned}$$

This tells us that we can now recursively call SHORTEN to shorten the right subtree, finishing the job!

Thinking back a bit, we recall that while backtracking after an insertion to find a node $p_j$ where we could rotate, if $sub(q_j)$ reached the floor we had to continue backtracking. However, if we compute $n$ and $k$ for $sub(q_j)$ and discover that $n < f(k)$, then we could call SHORTEN to reduce the height of $sub(q_j)$, allowing us to rotate at $p_j$. We will refer to an algorithm which does this as "compact with shortening."

For this algorithm, the tree's height will increase upon insertion only if all of the $sub(q_j)$'s reach the floor, and none of them can be shortened by SHORTEN.

Example.




In this case we can determine a lower bound $g(k)$ on the number of nodes which must have been in the tree when $r$ was inserted. $\mathrm{Sub}(q_{k-i})$ has height $i$, so it must have $\geq f(i)$ nodes. Summing, we obtain

$$g(k) = \sum_{i=1}^{k-1} f(i) + k \ .$$

The recurrence relation for $g$ is

$$g(k) = g(k-1) + g(k-2) + 2$$

with $g(0) = 0$, $g(1) = 1$. If we allow the tree's height to increase from $k$ to $k+1$ if and only if its size becomes greater than $g(k)$, then we can guarantee that a backtrack search to absorb a newly inserted node will always succeed when it is attempted. $g(k) \approx (1.9)\cdot(1.618)^k - 2$, so a tree grown with the compact-with-shortening algorithm is compact with respect to the function $h(n) = g^{-1}(n) \approx 1.44 \log_2 n$.

## Empirical Results

We now turn to an evaluation of the empirical performance of compact-tree algorithms. As a standard of comparison, the basic tree insertion algorithm (using no restructuring) and the AVL algorithm were coded. Several different compact-tree algorithms, and some variations of them, were coded. The algorithms were used to construct 1000-node trees, using as input random permutations of the sequence of keys numbering one to 1000.

In most cases each algorithm built ten trees. The statistics exhibited a small enough variance to give reasonable confidence in the results. The graphs show only the range in which values for the total path length (tpℓ) fell. Generally most of the values fell slightly below the middle of this range.

Before we construct a tree with the compact-SL algorithm we must select values for the parameters A, B and C of the height function.

The most important parameter is B. It determines the asymptotic growth rate of the inverse of the height function ($h^{-1}(k) = A \cdot B^k + C$). Increasing B causes the algorithm to restructure more often, because it attempts to keep the tree at a given height during larger numbers of insertions. The value of B cannot profitably be increased beyond two, because that corresponds to the growth rate of a tree always having minimum tpℓ.

The parameter A must be such that the initial values $h^{-1}(0), h^{-1}(1), \ldots$ of the function are appropriate. The values shouldn't exceed those of a minimal tree: 0,1,3,7,15,... and it was found that the sequence of values 0,1,3,6,12 was about as high as could be used before a plateau in the improvement of the tpℓ occurred. This gives us a narrow range of values for A, given a fixed B, and the values

actually chosen were picked somewhat at random from this range.

The value of C insures that the first four values of the sequence after truncation are 0,1,2,6. C is somewhere around -1.0.

Trees produced with B taking values of 1.5, 1.618, 1.7, 1.75, 1.8 and 1.85 showed the expected inverse relation -- the average tpℓ improved (decreased) as B was increased, and the running time (the sum of the fetchs and stores) increased. Looking at the first graph (p.20) and comparing with the AVL algorithm, we see that their running times are approximately the same. However, the average tpℓ for the AVL trees was noticeably better -- within 3% of optimum, as opposed to tpℓ's around 10% of optimum for the compact trees.

The champion in the race to see who can build the fastest is the basic insertion algorithm. The C-SL algorithm was run with B = 2.0 to see how compact a tree it could produce. The resulting trees were almost as good as the AVL trees, but running time for the C-SL algorithm was substantially slower.

The compact algorithm with a SHORTENING subroutine was also coded. It was observed that with B = 1.618 (the maximum value which can guarantee success during backtracking) that the SHORTEN routine was only called two or three times during the entire 1000 insertions, and the resultant trees had tpℓ's about the same as those produced by C-SL with B = 1.618. This algorithm was also run with B = 2.0. The algorithm increased its use of the SHORTEN routine, but surprisingly the trees produced were actually poorer than those produced by C-SL with B = 2.0. We conclude that the SHORTEN routine is not useful.

The second graph helps to justify the choice of six as a limit on the amount of backtracking to allow the C-SL algorithm. It would be

expected that a small value would result in many failures during attempts to restructure, resulting in a poorer tree. On the other hand, large values would increase running times, and possibly waste time looking for a place to rotate when none existed.

The graph shows the results when trees were grown with a C-SL algorithm using a height function $h$ such that $h^{-1}(k) = 1.15*(1.8)^k - 0.6$. We see that as the value of the backtracking limit (BTLIM) was increased the average values for the $tp\ell$ decreased, and the variance in the $tp\ell$'s also decreased. In addition, the number of failures to find a point of rotation decreased. A pronounced tapering in the rate of improvement occurs around BTLIM = 5 or 6, and we have chosen six as the limit for subsequent uses of the C-SL algorithms.

For BTLIM = 6 the number of rebalancings averaged 246 and the number of failures averaged about thirty. The statistics on the number of rebalancings for this experiment exhibited low correlation with the resulting averages for $tp\ell$'s. Apparently when or where rebalancings occurred, rather than how many, determined the eventual shape of the tree.

During the experiments it was conjectured that how compact a tree was during the initial stages of insertion determined to a large degree how good the final tree would be. This was supported by an experiment which built trees using the basic insertion algorithm for the first 500 insertions, and tried to compact the tree using C-SL with B = 2.0 and full backtracking for the last 500 insertions. The average $tp\ell$'s of the trees were not as good as for trees produced using C-SL with B = 1.7, and the running time of this 'combination' algorithm was substantially greater.

It is natural to try a reverse experiment -- build the tree first

using a compact tree algorithm, and then finish off the insertions using the basic insertion algorithm. I used C-SL with $B = 1.7$ because it exhibited a very low failure rate in backtracking (averaging less than two failures per one hundred insertions). The results are shown in the third graph.

Observe that we can actually build a tree faster with this hybrid algorithm than if we use only basic insertion, and the resulting trees are more compact. It is not hard to guess why this should be so. The C-SL algorithm with $B = 1.7$ runs more slowly than basic insertion principally because of a large number of stores, most of which can be attributed to stacking of pointers to nodes along the path of insertion. Hence, if we only use C-SL in the beginning, we avoid stacking up long paths, and save most of the stores. On the other hand, because the basic insertion algorithm finds a better than random tree when it cuts in, it tends to insert the remaining nodes more evenly across the frontier of the tree. This results in the tree always being more compact than a random tree, which means that the average length of a path of insertion is decreased. Hence the basic insertion algorithm has a faster than usual rate of execution.

This hybrid algorithm might be a good choice for building a tree if the ratio of look-ups to insertions was expected to be relatively low. A similar hybrid combining the AVL algorithm with basic insertion might be expected to run even a touch faster, but once basic insertion commenced the balance fields would become useless, and the AVL algorithm could not again be applied to the same tree.

There was no obvious counterpart to the C-SL algorithm for the case of deletion. Intuitively, we could say that a compact tree algorithm

can work quickly because the point of insertion is also the place where a tree first increases its height. The effect of a deletion from a tree may be realized well below the point where a deletion actually occurred, and hence it is likely that only a rather contrived algorithm could find the places requiring restructuring after deletion.

We also ran our algorithms on 5000-node trees. The AVL algorithm built 5000-node trees which had tpℓ's averaging within 1% of optimum, so its performance in this department improved somewhat. Trees built with C-SL, setting B = 1.7, had tpℓ's around 11% above the optimum, only slightly worse than was the case for 1000-node trees. Rebalancing occurred about one-fifth of the time. C-SL built its trees slightly faster than did AVL.

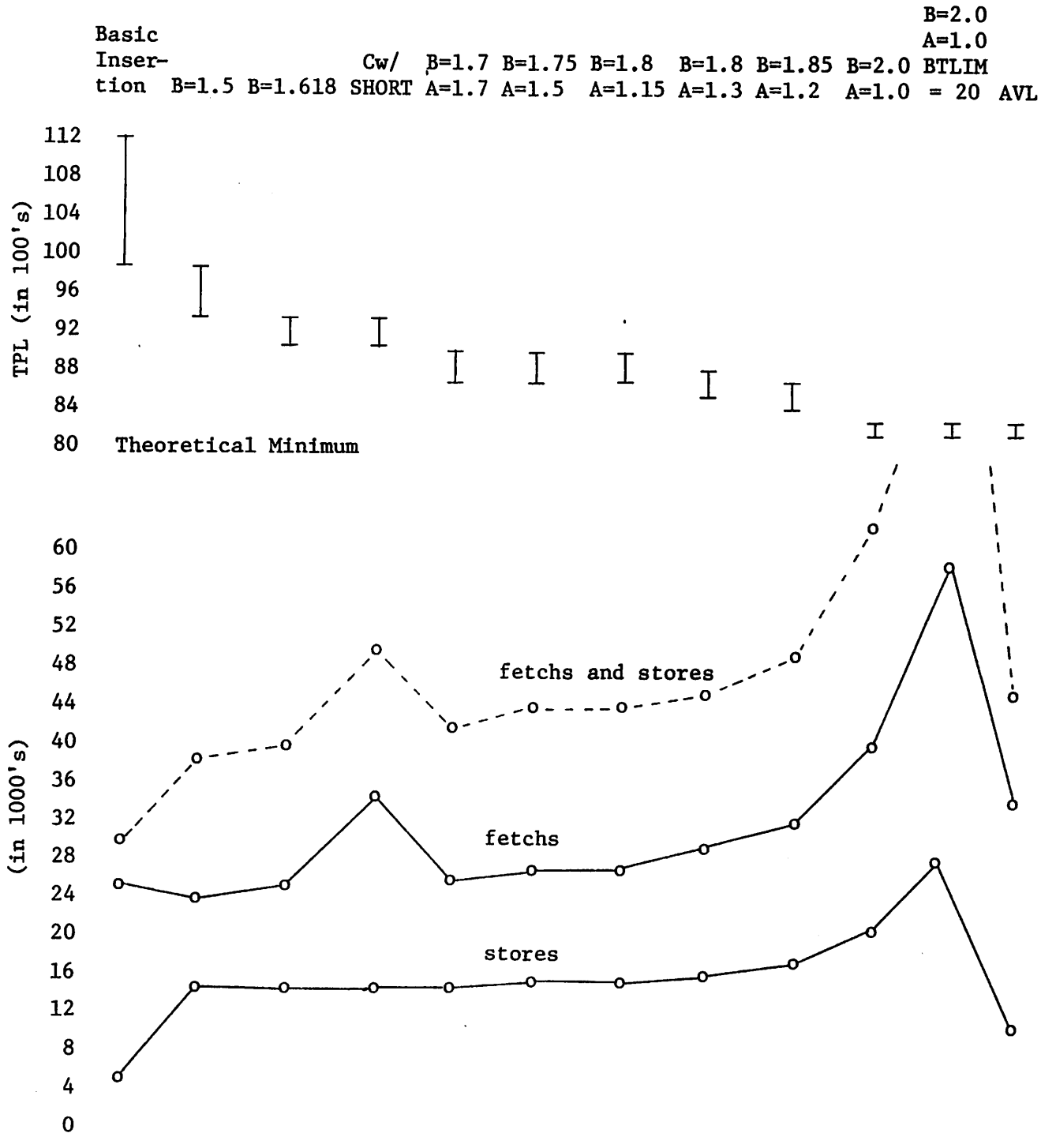
After building the 5000-node trees, we deleted them. The basic tree deletion algorithm was used on trees built by C-SL. We checked the tpℓ of the trees when only 1000 nodes were left and found an unexpected result. We predicted that after the random deletion of 4000 nodes with no restructuring we would be left with trees which were essentially random. In fact, the 1000-node trees left were still relatively compact, with tpℓ's averaging slightly over 9000. This is within 2% of the tpℓ's for trees grown from scratch, using C-SL with B = 1.7.

Basic deletion took only three-fifths as long to delete 5000-nodes as did AVL deletion. Hence the overall time to build and delete the trees using C-SL was about four-fifths of the time taken by the AVL algorithm.

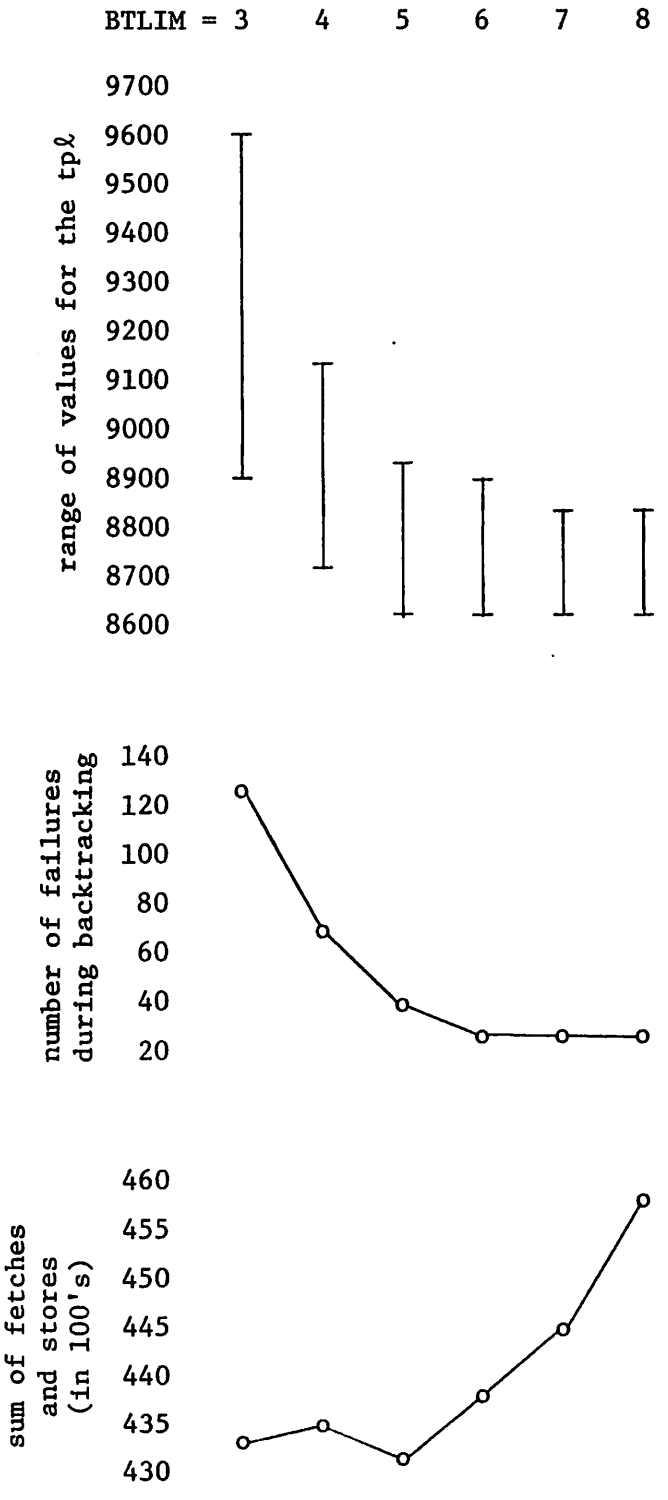
A brief trial of one other compact tree algorithm took place. It worked like C-SL except that it maintained the current balance (difference in heights between the right and left subtrees) of each node, so that no

backtracking was required to find the place to rotate for a rebalancing. When a choice of rotations at nodes along the path of insertion existed, then the rotation took place at the highest (nearest the root) of these nodes. It was found that the trees produced were similar to their C-SL counterparts in average tpℓ, but the added complexity and a slower running time made the algorithm non-competitive.

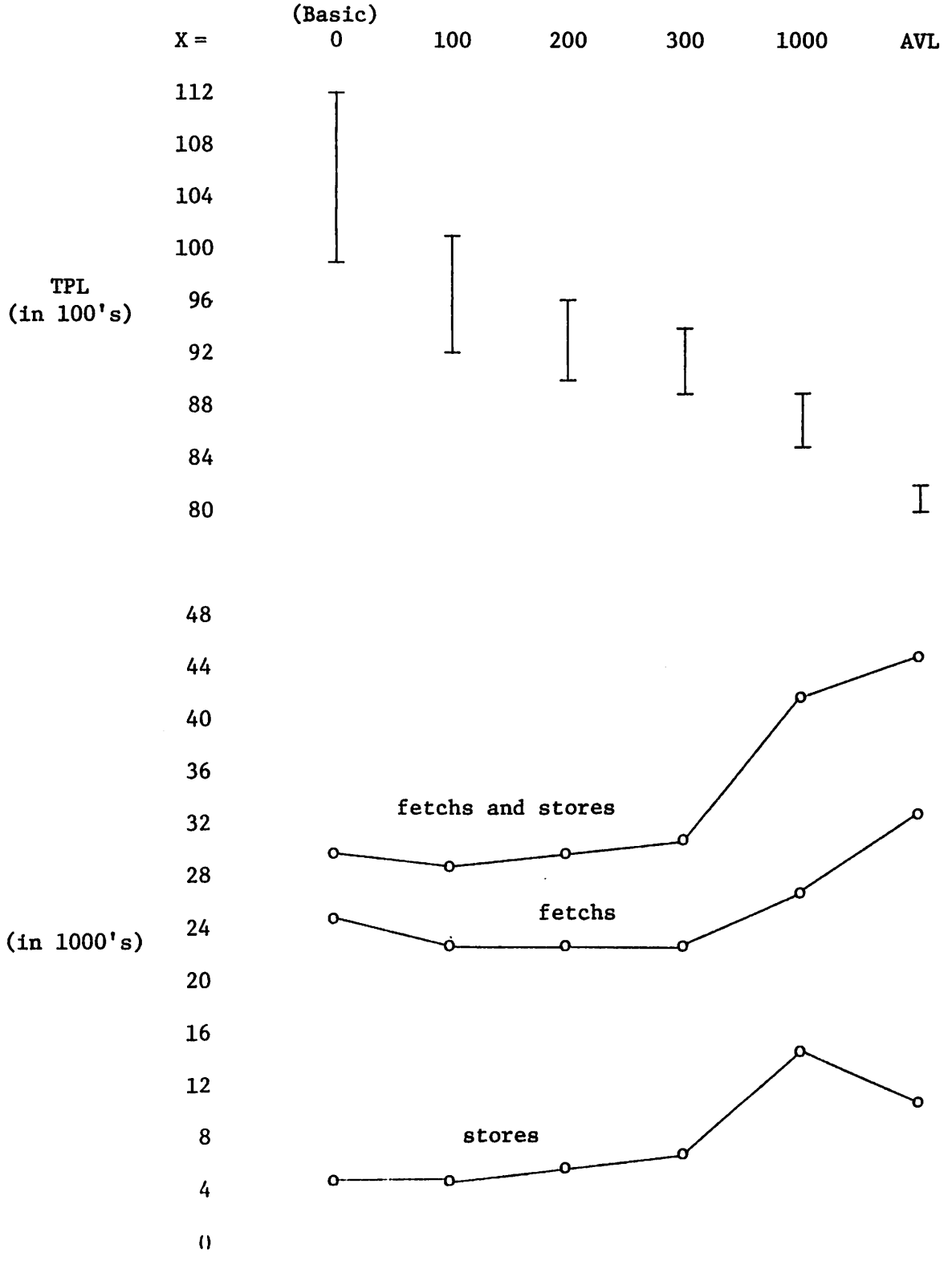
Comparison of Compact Tree Algorithms: Construction of 1000-node trees (BTLIM = 6)



Comparison of Cut-off Points for Backtracking: Construction of 1000-node trees using C-SL with B = 1.8, A = 1.15.



Speed Contest: Construction of 1000-node trees using C-SL with B = 1.7, A = 1.7, BTLIM = 6, for the first X insertions. BASIC ALG used for the last 1000-X insertions.



Conclusions

Let us refer to the hybrid algorithm, which follows initial use of the C-SL algorithm with use of the basic insertion algorithm, as algorithm H. The C-SL algorithm may be regarded as a variant of algorithm H. Algorithm H is the only compact-tree algorithm which might be used in practice, so we will evaluate its performance.

Advantages of Algorithm H: The program for it is simple. Essentially it just requires the addition to a program for basic insertion of a subroutine which performs backtracking and rotations, and a mechanism to stack the nodes along the path of insertion. In contrast, a program which performs insertions and deletions on an AVL tree is larger and more complicated.

Algorithm H is fast if the input data is reasonably random -- faster than straight basic insertion by a small amount, while producing a more compact tree.

C-SL is slower than algorithm H mainly because it must store pointers to all nodes along the path of insertion. However, it only uses the last five or six nodes while backtracking. One can imagine a piece of hardware which would keep in k registers only the last k items stored into it. With such a device, with k equal to five or six, we could eliminate most stores to memory from the C-SL algorithm, and hence increase its speed to about that for algorithm H, while improving the trees which were produced.

Algorithm H eliminates the need for "balance bits", or any other extraneous fields in the nodes of the tree. Under some circumstances this might result in a noticeable savings in space (e.g. if the tree-insertion program was to be written in a high-level language which

assigned a word of storage to each field.

Finally, algorithm H will operate on any binary search tree -- it doesn't require maintenance of a special underlying structure, as in AVL or "weight-balanced" trees.

Disadvantages of Algorithm H: There are two serious disadvantages to using algorithm H. The first is that the expected tpℓ of a tree grown by algorithm H is around 14% longer than for AVL trees. This increases the running time during "look-up" operations. Usually one would expect that the time spent looking up nodes in the tree would be much greater than the time spent inserting or deleting nodes.

The second disadvantage is that the performance of algorithm H is very sensitive to the order in which keys are inserted. If the incoming keys are ordered, or semi-ordered, the C-SL algorithm slows down quite a bit, spending a lot of time backtracking and restructuring. Also, there exist trees which have relatively large tpℓ's but aren't restructured by the C-SL algorithm (i.e. when all subtrees off of the path of insertion reach the floor, but each is rather sparsely filled). Finally, use of the basic insertion algorithm after initial use of C-SL relies on a semi-random sequence of insertions to produce a reasonably compact tree.

In contrast, the AVL algorithm is very well behaved under almost all sequences of insertions, and is guaranteed to produce trees with height less than or equal to about $1.44 \log_2 N$, given N insertions.

We conclude that for some specialized uses algorithm H might be the algorithm of choice.

## Appendix: Some Theoretical Results

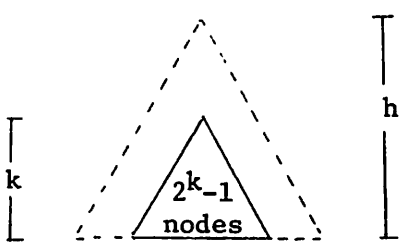
This section presents a few results concerning compact trees.

Theorem. Suppose an n-node tree is guaranteed to be compact with respect to a height function $h(n) \leq \log_b n + c_o$ for some constants $b$ and $c_o$. Then its total path length is bounded above by a function whose asymptotic growth is $O(n \log n)$.

Proof. Suppose $n = 2^k - 1$. Observe that if there are $m$ nodes on level $i$, then there must be at least $\lceil \frac{m}{2} \rceil$ nodes on level $i-1$, since each node has at most two sons. In a worst case we try to place as many nodes as possible on the bottom level. We can do no worse than to have $2^{k-1}$ nodes at the lowest level, $2^{k-2}$ on the second lowest, etc., noting that $1+2+4+\cdots+2^{k-1} = 2^k - 1$. (If the tree's height $h$ exceeds $k$, then we need $h-k$ more nodes to form a path from the root to the root of the subtree we have just built, but for simplicity we will disregard them.)

The total path length of the subtree we have just built is $(k-2)2^k + 2$. If the whole tree has height $h$ then we must add the amount $h-k$ for each node in this subtree. Hence the total path length for a compact tree with $2^k - 1$ nodes is less than

$$(k-2)2^k + 2 + (h-k)(2^k-1) = (h-2)2^k + 2 - h + k \quad .$$



Because $h \leq \log_b(2^k-1) + c_o$ and $h \geq k$ we have

$$\begin{aligned} tp\ell &\leq (k \log_b 2 + (c_o-2))2^k + 2 \\ &= (kc_1 + c_2)2^k + c_3 \qquad \text{for some constants } c_1, c_2, c_3. \end{aligned}$$

Setting $n = 2^k - 1$ gives $\log_2(n+1) = k$ so that

$$\begin{aligned} tp\ell &\leq (c_1 \log_2(n+1) + c_2)(n+1) + c_3 \\ &= O(n \log n) \quad . \end{aligned}$$

For $2^k - 1 < n < 2^{k+1} - 1$ the ratio $\frac{\text{worst case } tp\ell}{n}$ is less than for the bound derived for $n = 2^k - 1$, so that the total path length is always bounded by an $O(n \log n)$ function. □

Theorem. When keys are inserted in increasing order by a compact-tree algorithm, the running time is $O(n \log n)$ to insert $n$ nodes into an initially empty tree.

Proof. To prove this result we will keep track of the shape of the tree as it is being built. Assume a function $F$ exists such that we insert $F(k)$ nodes between each increase in the tree's height.

Lemma 1. Suppose we have inserted $n$ nodes into an initially empty tree using a compact tree algorithm which backtracks from the inserted node $r$ to find a place to rotate, and that the keys occurred in increasing order. Then insertion always takes place at the "lower right corner" of the tree. Let $p_1$ be the root node and let $p_{i+1}$ be the right son of node $p_i$, for $1 \leq i \leq k-1$ (there are $k$ levels in the tree). Let $s_i$ be the left subtree of $p_i$.

(*) Assume that if no rotation occurs at a node $p_i$ for all of the $F(k)$ insertions which occur while the tree has height $k$, then for the remainder of the insertions no rotations will take place at nodes $p_1, p_2, \ldots, p_i$.

We will show: The path $p_1, p_2, \ldots, p_k$ always extends to the floor of the tree, and there exist at any step variables $a$, $b$ and $c$ with $0 \leq a \leq b \leq c \leq k$ such that

(i) subtrees $s_1, s_2, \ldots, s_{a-1}$ don't reach the floor,

(ii) subtrees $s_a, s_{a+1}, \ldots, s_{b-1}$ don't reach the floor, but they reach level $k-1$ or level $k-2$,

(iii) subtree $s_b$ reaches the floor (unless $b = k$),

(iv) subtrees $s_{b+1}, s_{b+2}, \ldots, s_{c-1}$ either reach the floor or reach level $k-1$,

(v) subtrees $s_c, s_{c+1}, \ldots, s_{k-1}$ all reach the floor.

Call a subtree "f-compact" if it can't be shortened by SHORTEN because its number of nodes $n$ and height $h$ obey the inequality $n \geq f(h)$. Then

(vi) all subtrees $s_i$ are f-compact, $1 \leq i \leq k-1$.

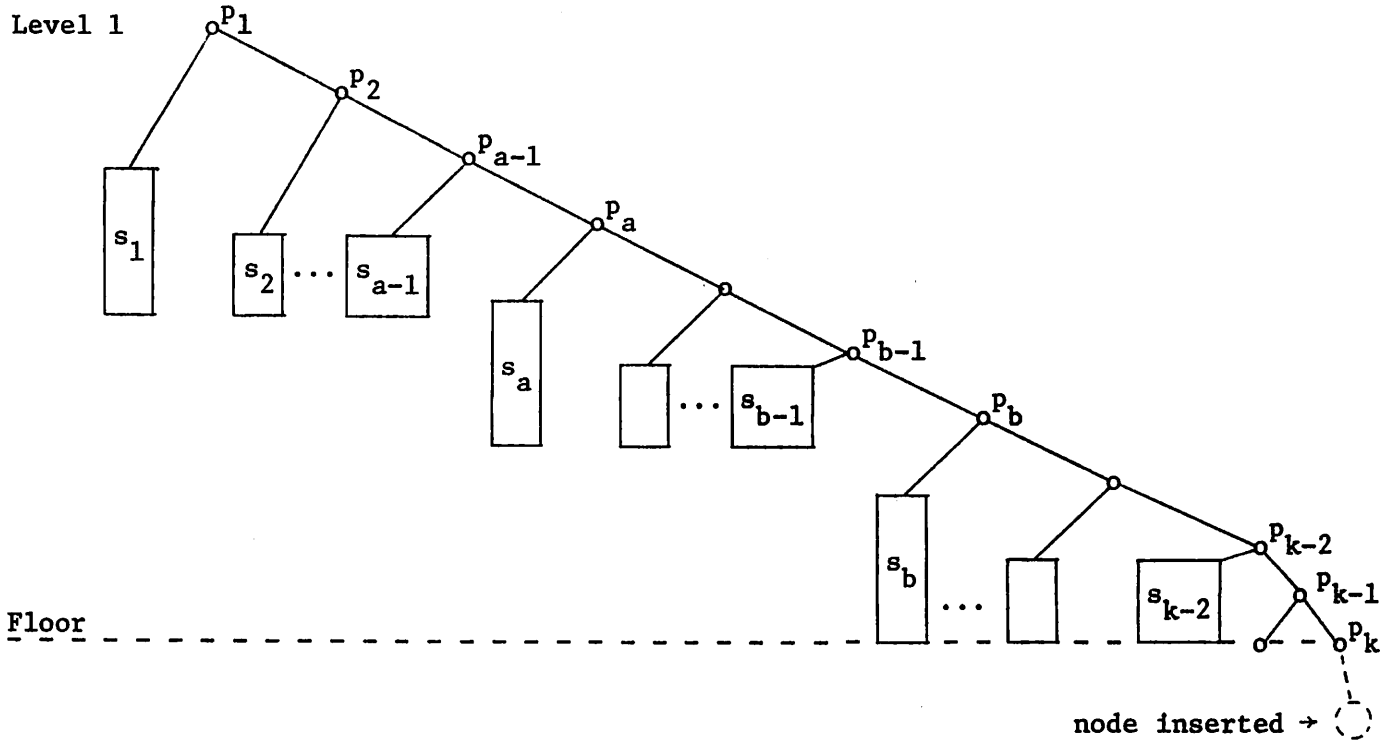
Let $\boxed{s_i}$ denote a (f-compact) subtree. (f is defined on p.10.)

The proof of the lemma is by induction on $n$, the number of nodes in the tree.

Basis. $n = 1$. Set $a = b = c = 1$ and $k = 1$.

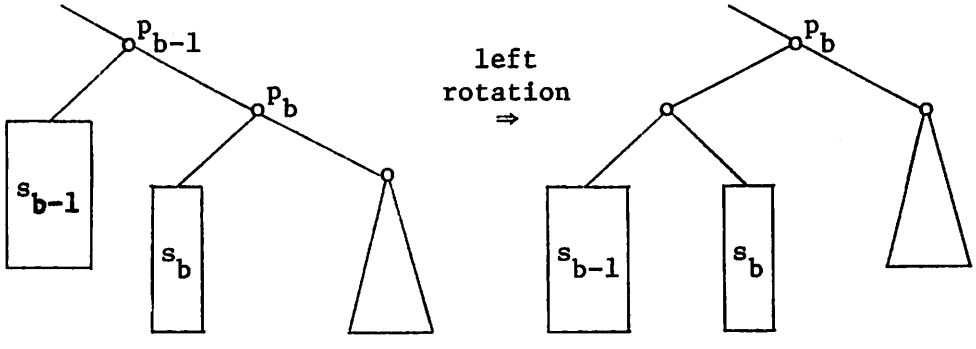
Induction Step. Let $G(k)$ be the function such that the tree's height grows from $k$ to $k+1$ when $n = G(k+1)$ $(G(k+1) = G(k) + F(k))$. Assume that the tree has $n$ nodes, height $k$, and $G(k) \leq n < G(k+1)$. We observe the result when a node is added to the right end of the tree,

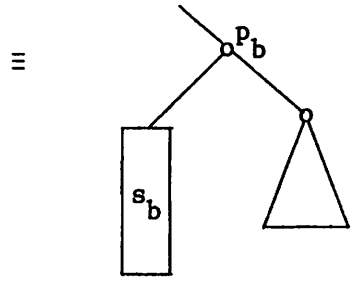
i.e. $n \leftarrow n+1$.




This is a picture of

Case 1. $n+1 < G(k+1)$, $a < b = c$. A node is added to the right of $p_k$, and we must rotate. Because $s_b, s_{b+1}, \ldots, s_{k-1}$ are f-compact and reach the floor we backtrack to $p_{b-1}$ and rotate there. If $s_b$ has height $i-1$ and $s_{b-1}$ has height $\geq i-2$, then we know that the number





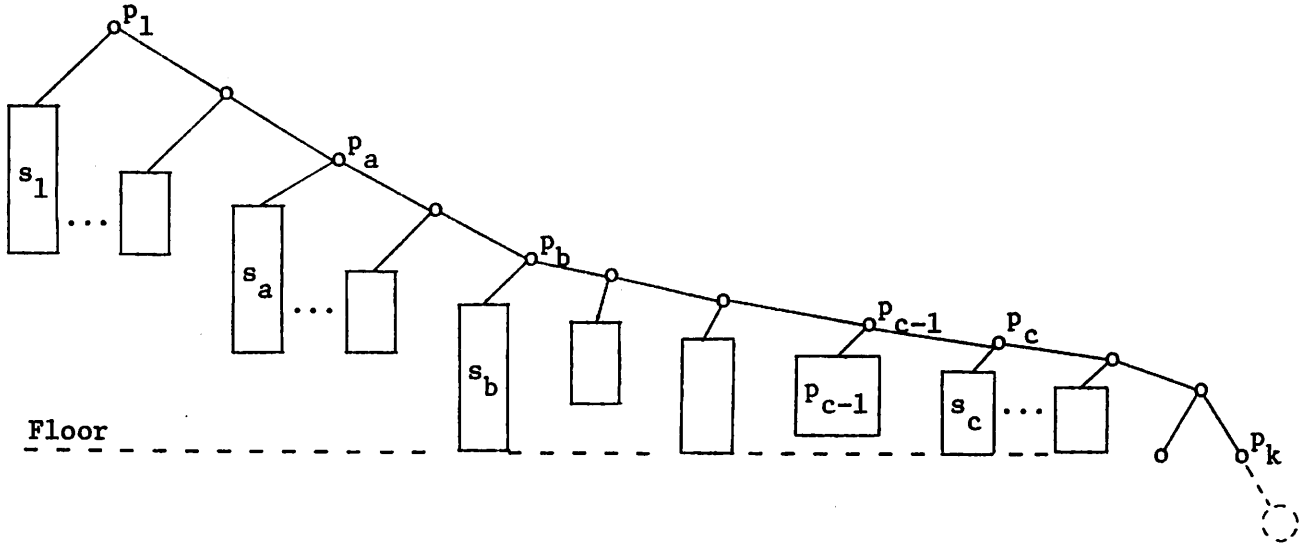

after relabeling

of nodes in the new subtree $s_b$ (in the above frame) is $\geq$ $f(i-1)+f(i-2)+1 = f(i)$. (As drawn $s_{b-1}$ appears to have height $i-1$.) Hence the new subtree $s_b$ with height $i$ is f-compact, as claimed. We must set $b \leftarrow b-1$. Because the subtrees $s_{c+1}, s_{c+2}, \ldots, s_{k-1}$ represented by the triangle all left the floor, we set $c \leftarrow k$.

Now the picture belongs to

Case 2. $n+1 < G(k+1)$, $b < c-1$

Example.




Here the rotation takes place at $p_{c-1}$ (variable $c$ is always chosen so that $s_{c-1}$ doesn't reach the floor, except possibly when $b = c$). As in Case 1, the subtree which is the new $s_c$ is f-compact and reaches

the floor. If it happens that $s_{b+1}, s_{b+2}, \ldots, s_k$ all reach the floor we set $c \leftarrow b$ and are in Case 1 unless $n+1 = G(k+1)$. Otherwise set $c \leftarrow k$. We stay in Case 2 if $n+1 < G(k+1)$.

Case 3. $n+1 = G(k+1)$. In this case the height of the tree increases upon insertion, so $k \leftarrow k+1$. Label the new node $p_k$ and drop the floor one level. Set $a \leftarrow b$ and then $b \leftarrow k$, $c \leftarrow k$ and we land in Case 1.

Case 4. $n+1 < G(k+1)$, $a = b = c$. If this case occurs then we rotate after insertion at $p_{a-1}$, but we can't guarantee that the new $s_a$ is f-compact. Observe however that $p_b$ is always the top-most $p_i$ where a rotation occurred while we inserted nodes at a fixed level, and in Case 3 we set $a \leftarrow b$. Hence while inserting at which we now call level $k-1$, a rotation at what is now labeled $p_{a-1}$ never occurred. We now make use of the assumption (*) made earlier to assert that no rotations will occur at $p_1, p_2, \ldots, p_{a-1}$. Hence Case 4 can't happen.

This proves Lemma 1.

Now we would like to justify the assumption (*).

Lemma 2. Given the conditions of the theorem, then (*) is true.

This involves a counting argument.

We inspect what happens to a part of the tree which initially looks like

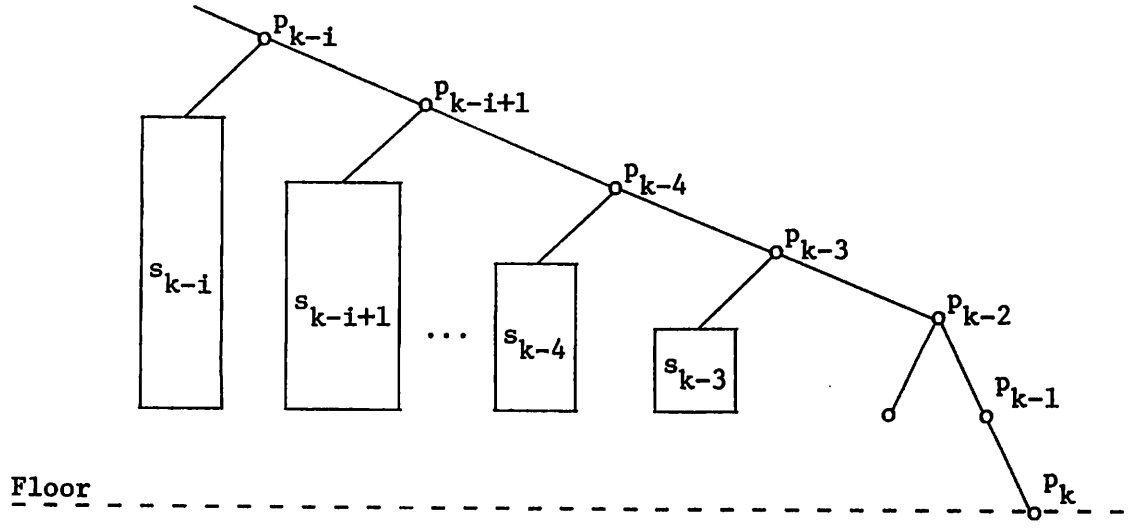



Fig. 1

Induction Hypothesis: After $2^i-1$ insertions to the right of $p_k$ the resultant tree has $s_{k-i}, s_{k-i+1}, \ldots, s_{k-1}$ all reaching the floor, and $2^{i-j}$ rotations occurred at each node $p_{k-j}$, $0 < j \leq i$. No rotations have occurred at $p_{k-j}$ if $j > i$.

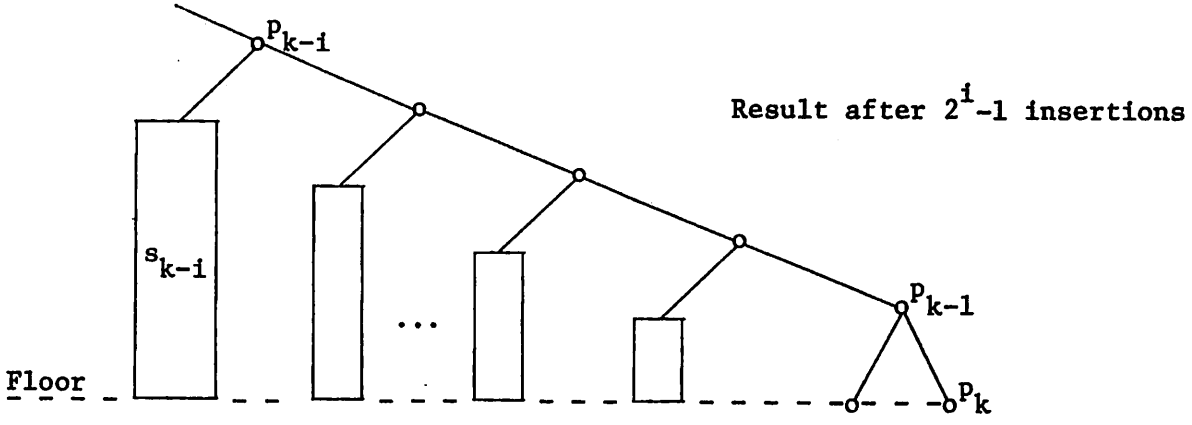
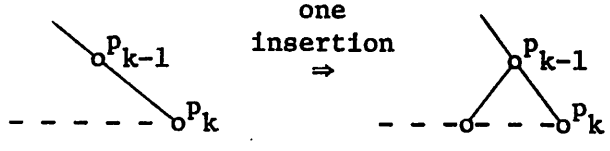



Fig. 2

We prove the hypothesis by induction on $i$.

Basis. $i = 1$



We see that $2^1 - 1 = 1$ insertion produces the desired figure, and we rotated $2^{i-i} = 1$ time at node $p_{k-i}$.

Induction Step. $i > 1$

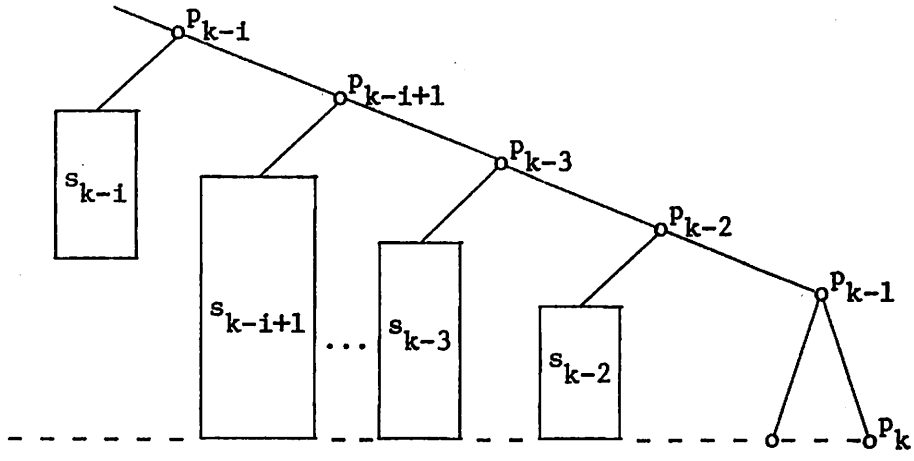



Fig. 3

We assume by hypothesis that Fig. 3 is produced from Fig. 1 after $2^{i-1} - 1$ insertions.

One more insertion produces Fig. 4 (after relabeling).

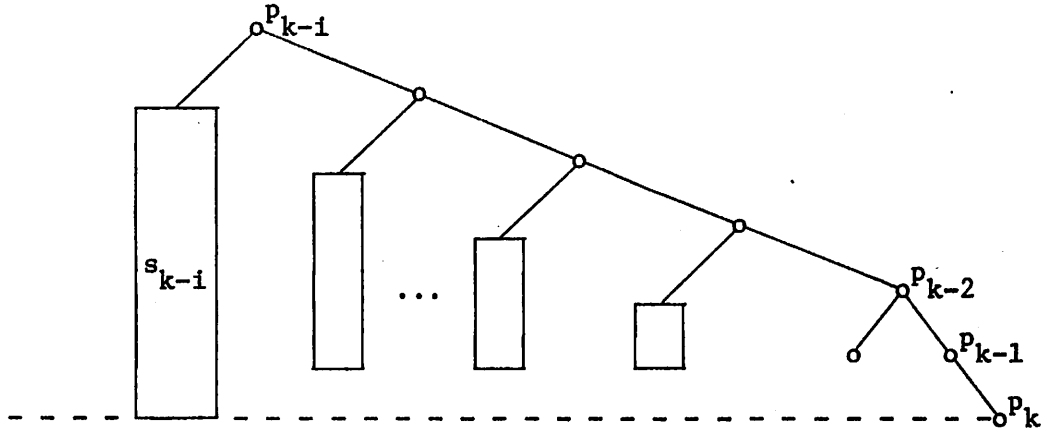



Fig. 4

Finally, $2^{i-1}-1$ more insertions produce Fig. 2. The total is $(2^{i-1}-1)+1+(2^{i-1}-1) = 2^i-1$ insertions. □

At node $p_{k-i}$ we had $2^{i-1} = 1$ rotation and at each node $p_{k-j}$, for $0 < j < i$, there were $2 \cdot 2^{i-1-j} = 2^{i-j}$ rotations. □

This finishes the induction.

Starting with $n+1 = G(k)$, one insertion produces the result in Fig. 1, and $(2^{i-1}-1)+1 = 2^{i-1}$ more insertions produce a rotation at $p_{k-i}$. There are $F(k)$ insertions at this level. Hence we don't rotate at $p_{k-i}$ if and only if $F(k) \leq 2^{i-1}$. At the next level we don't rotate at this node if and only if $F(k+1) \leq 2^i$. Hence Lemma 2 is true if $F(k+1) \leq 2F(k)$.

For a compact algorithm we set $G(k) = a \cdot b^k - c$, $1 < b \leq 2$. Then $F(k) = G(k+1) - G(b) = a(b^{k+1} - b^k) = (ab-b)b^k$. Since $b \leq 2$ implies $F(k+1) \leq 2F(k)$, we are done.

We note that if we were using the compact-with-shortening algorithm then $G(k) = g(k)$ and $F(k) = f(k)+1$.

The result of the previous theorem tells us that the time spent on inserting the nodes to build an n-node tree is $O(n \log n)$. If we can show that the number of rotations and the number of nodes visited during backtracking are both $O(n \log n)$, then the time spent on restructuring is also.

Each time we performed a rotation at node $p_j$ we first had to visit all of the nodes in $sub(p_j)$, and hence we will just count the number of visits. $Sub(p_{k-j})$ has height $j+1$ and hence has $\leq 2^{j+1}-1$ nodes.

At level $k$ we insert $F(k)$ nodes. Suppose we never rotate at $p_{k-i}$ (possibly $i = k$), then the most nodes we could have visited is

$$\leq \sum_{j=0}^{i-1} (2^{j+1}-1)(\#\text{ rotations at } p_{k-j})$$

$$= \sum_{j=0}^{i-1} (2^{j+1}-1)(2^{i-j}) \leq \sum_{j=0}^{i-1} 2^{i+1} = i \cdot 2^{i+1} \quad .$$

We know $F(k) \leq 2^{i-1}$, implying $\log_2 F(k) \leq i-1$ or $i \geq 1+\log_2 F(k)$. We can make $i$ as small as possible, so set $i = 1+\lceil \log_2 F(k) \rceil$. Then the number of nodes visited at level $k$ is

$$\leq (1+\lceil \log_2 F(k) \rceil) \cdot 2^{(2+\lceil \log_2 F(k) \rceil)}$$

$$\leq c_1 F(k) \log F(k) \quad \text{for some constant } c_1 \ .$$

We know that $F(k) \leq a \cdot b^k$ for some constants $a$ and $b$, $b \leq 2$. Then the total number of nodes visited during backtracking while building a tree of $n$ nodes is

$$\leq c_1 \sum_{k=0}^{m} ab^k \lceil \log_2 ab^k \rceil$$

$$\leq c_2 \sum_{k=1}^{m} kb^k$$

$$\leq c_3 m b^m \quad \text{for some constants } c_2 \text{ and } c_3$$

and a value $m \approx \log_b n$. Therefore the total number of visits is $O(n \log n)$. □

Acknowledgments

I would like to thank Professor R.M. Karp for providing the initial impetus for this paper, and for many valuable discussions thereafter.

## References

[1] Knuth, D.E. (1973). The Art of Computer Programming, Vol. III. Addison-Wesley Publishing Co., Reading, Mass.

[2] Walker, Aldon N. "An Investigation and Implementation of Some Binary Search Tree Algorithms." Computer Science Technical Report No. 74/8, Department of Applied Mathematics, McMaster University, Hamilton, Ontario, Canada.