University of Alberta

Asymmetry in Binary Search Tree Update Algorithms

by

Joseph Culberson and Patricia A. Evans

Technical Report TR 94–09
May 1994

DEPARTMENT OF COMPUTING SCIENCE
The University of Alberta
Edmonton, Alberta, Canada

Asymmetry in Binary Search Tree Update Algorithms

Joseph C. Culberson * Patricia A. Evans †

May 17, 1994

Abstract

In this paper we explore the relationship between asymmetries in deletion algorithms used in updating binary search trees, and the resulting long term behavior of the search trees. We show that even what would appear to be negligible asymmetric effects accumulate to cause long term degeneration. This persists even in the face of other effects that would appear to counteract the long term effects. On the other hand, eliminating the asymmetry completely seems to give us trees that have a smaller IPL than is expected for trees built by a random sequence of insertions. But even then there are surprises in that the backbone becomes longer than expected.

1 Introduction

Binary search trees are one of the oldest and most frequently used data structures for solving the dictionary and other problems [2, 11, 6, 9]. The average case efficiency of these structures has been well studied, when only insertions are involved. The usual insertion algorithm simply inserts new values at the leaf, although insertion at the root has also been studied [10].

However, developing and analyzing deletion algorithms, and their interaction with insertion algorithms, has proven more difficult [6, 9, 7, 5, 1, 4, 3].

*Supported by Natural Sciences and Engineering Research Council Grant No. OGP8053. Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, T6G 2H1. email:joe@cs.ualberta.ca

†NSERC Undergraduate Summer Student Research Award. This was work done during the summer and fall of 1990.

Although there are many algorithms that achieve efficient operation by enforcing balancing constraints, we are here looking at the long term behavior of trees using algorithms that do not use such mechanisms.

The purpose of this paper is a study of the effects of asymmetry in deletion algorithms. In [4, 3] it was shown that the algorithm proposed by Hibbard [6] causes long term deterioration, leading to trees in which the average depth of a node is $O(\sqrt{n})$. Simulations show that by making the algorithm symmetric by randomly choosing between mirror versions of Hibbard's algorithm, or the improved algorithm proposed by Knuth [9], leads to trees with $O(\log n)$ depth, where the constant factor is actually improved over that of a tree generated by a random insertion sequence. However, no analysis has been presented confirming these observations.

In this paper we attempt to shed more light on the effects of asymmetry by making further modifications to these algorithms. We improve Knuth's algorithm further in the same sense he improved Hibbard's. We systematically modify the algorithms so that the asymmetry evident in these algorithms appears to be greatly reduced, although not eliminated. We hasten to point out that these algorithms are designed for purposes of studying trees, not as practical algorithms, although the symmetric forms do seem to achieve a high level of search efficiency. The surprising and paradoxical results that we achieve show that, although the rate at which the trees degenerate is reduced, after sufficiently long sequences of updates the trees are far more imbalanced than the ones produced by the usual algorithms. Symmetric versions of our algorithms do not degenerate, verifying that it is indeed the asymmetry that causes the deterioration.

2 Algorithms and Definitions

A *binary tree* is a (possibly empty) collection of nodes, consisting of one distinguished node, called the *root*, and two subtrees, referred to as the *left* and *right* subtrees. A *binary search tree* is a binary tree in which each node contains a real number called the *key*, such that all keys in the left subtree of any node are less than the key of the node, and all the keys in the right subtree are larger. If both subtrees of a node are empty, the node is referred to as a *leaf*. In describing the following algorithms, we use a number of terms that are defined in an appendix.

The insertion algorithm that we use is the usual one, in which a search is made for the key, and when it fails, the key is inserted as a leaf where the search terminated.

The problems facing a designer of a deletion algorithm stems from the requirement of maintaining order in the structure when keys from non-leaf nodes are to be deleted. In these circumstances restructuring of the tree and reassignment of keys to nodes is generally necessary. To delete a key in a leaf, all that is necessary is to remove the node. The natural choices arise when we wish to delete a non-leaf element.

Hibbard's Algorithm

In Hibbard's algorithm [6] when a non-leaf key is to be deleted, the key is replaced by the key in the successor node, if the successor exists. Then the successor node is deleted, and its right subtree, if it exists, is re-attached to the successor's parent in its place. This action is illustrated in figure 1. If there is no successor, then the node is deleted, and its left subtree if it exists, is re-attached to the node's parent.

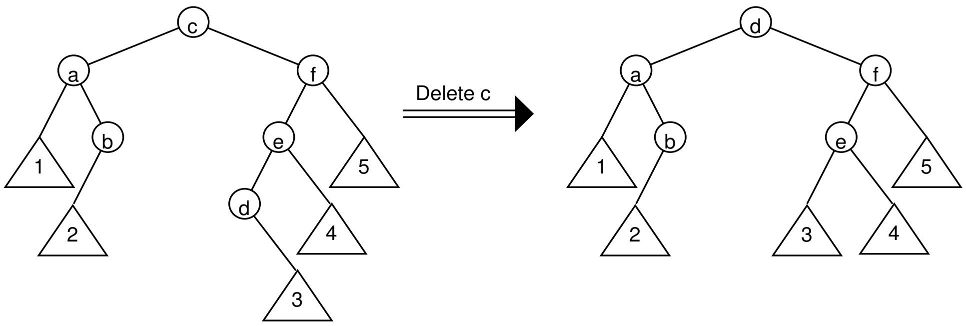

Figure 1: Deletion of 'c' using Hibbard's Algorithm

This algorithm has the remarkable property that after inserting $n + 1$ elements in random order, and then deleting one element selected at random from the tree, the probability distribution of *shapes* of trees is the same as if the remaining n elements had been inserted in random order [6]. Unfortunately, as observed by Knott [8] the distribution of shapes is destroyed with the next random insertion. The problem is that the distribution of trees over $n+1$ elements is not the same as the distribution obtained by inserting n elements selected randomly from the set of $n + 1$ elements. The studies cited in the introduction show that using this algorithm for long sequences of updates causes the tree to seriously deteriorate, so that nodes are at an average depth of $O(\sqrt{n})$.

Knuth's Algorithm

Knuth [9] observed that if the left subtree of the node whose key is to be deleted is empty, then instead of using the successor key as a replacement, it is always at least as effective to delete the node, and bring up its right subtree as a replacement. However, this algorithm does not improve the long term behavior of the tree very much, and the average depth is still $O(\sqrt{n})$ [3]. In this paper we sometimes refer to this as algorithm A.

Algorithm B

This is a modification of Knuth's algorithm such that if

$$depth(node, successor) > depth(node, predecessor)$$

then the key from the node in the successor path whose depth is one greater than the depth of the predecessor is used as the replacement. The replacement node is then deleted, and its subtrees are reattached as illustrated in figure 2.

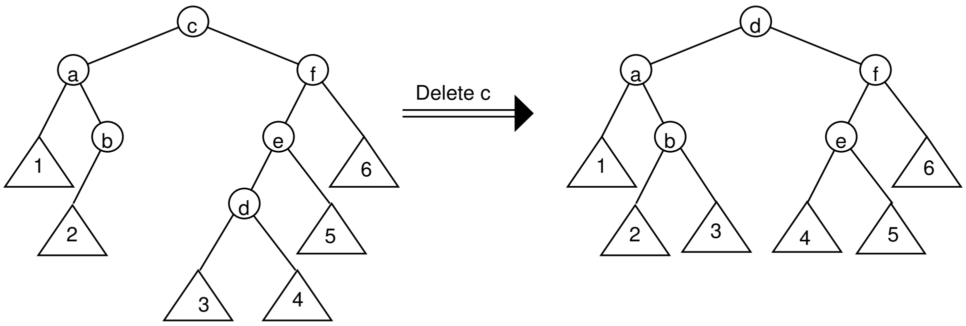

Figure 2: Deletion of 'c' using Algorithm B

The number of nodes reduced in depth is always greater than or equal to the number reduced in depth by Knuth's algorithm, and thus for any single deletion, this algorithm is superior to Knuth's. It was in fact designed as a natural extension to Knuth's idea.

Algorithm B is highly asymmetric. If we think of the node whose key is scheduled for deletion as a fixed point in the structure, then whenever both subtrees exist, and the successor key is not used as the replacement key, some nodes from the right subtree are moved to the left subtree. Such movement never occurs in the opposite direction.

Algorithm C

Algorithm C is a more symmetric version of B where, if

$$depth(node, successor) < depth(node, predecessor)$$

then the key in the node in the predecessor path whose depth is one greater than the depth of the successor is used as the replacement. The replacement node is then deleted, and its subtrees are reattached in a fashion symmetric to algorithm B. The only asymmetric bias that remains occurs when the successor and predecessor are at the same depth, in which case the key from the successor is always used as the replacement. To enable this reduction in asymmetry, the guaranteed improvement over A for a single deletion is relinquished.

We might expect that the small bias in this algorithm would lead to only slight imbalances in the asymptotic results. Surely the shifting of nodes within the domain as in [4] would be offset by the reshuffling effects, and besides, if the right subtree of a node tends to be smaller than the left, then the successor would be at a shallower depth. Thus, we *might* expect that the skewing evidenced in [4] would be stopped before it could cause the extensive deterioration demonstrated there.

Algorithm D

Algorithm D is a modification of Knuth's algorithm (A) wherein the deeper of the predecessor or successor is used as the replacement key unless they are at equal depth, in which case the successor is used. Alternatively, this algorithm can be seen as a modification of C, in which the reshuffling of subtrees has largely been eliminated.

Although this algorithm has asymmetry similar to C, it should deteriorate faster and to a worse state than C, as D has none of the symmetric rebalancing employed by C beyond the selection of the deeper node as the replacement value. On the other hand, we might expect it to be better than Knuth's algorithm, since it seems to eliminate most of the asymmetry evident in it.

Algorithms E and F

In [5, 4] symmetric versions of the Hibbard and Knuth algorithms exhibited much better behavior than the asymmetric versions. These symmetric versions are designed by making a second copy of the algorithm, except that the roles of left and right, and successor and predecessor are reversed. Then whenever a deletion is to be made, one of the two versions is chosen randomly to perform the deletion. This effectively prevents any long term bias from skewing the tree. Algorithms E and F are the symmetric forms of C and D respectively. E and F are actually implemented by simply flipping a coin when the successor and predecessor are at the same depth, since this was the only point of asymmetry in algorithms C and D. These algorithms should exhibit no left to right skewing, and so we hope to isolate the effects of the extensions to Knuth's algorithm from the skewing effects.

3 Simulation Results

Algorithms A through D were simulated for tree sizes 512, 1024, 2048, and 4096 nodes. For each simulation, a random binary search tree was constructed and subjected to repeated updates, each composed of a random deletion followed by a random insertion. Each tree size and algorithm pair was simulated 10 times. Each tree simulation was continued until deterioration had stopped and a slightly oscillating steady state was observed. However, given the long lead times, we cannot guarantee that the final steady state was reached. Simulations from an initial state of a left linear tree are observed to come down and meet the observed results from the simulations which start with a random tree, so the observed state at which the simulations were concluded is supported by observations as being the steady state of the tree.

In order to determine the behavior of the trees as updates are performed, the following measures, defined in appendix I, were taken. The internal path length (IPL) gives us the measure of overall efficiency of the tree. The skewness measure gives us one measure of left right imbalance, as does the length of the backbone and forebone. Since the algorithms depend heavily upon the relative lengths of the predecessor and successor paths, it seemed reasonable to compute the total length over all nodes of the predecessor and successor paths. However, the following relationships can be proven to hold (see appendix II)

Lemma 3.1 *The total length of all successor paths is $n - |backbone|$, the total length of all predecessor paths is $n - |forebone|$ and thus the total depth of all predecessor and successor paths is $2n - 1 - |shell|$.*

This indicates that the length of the backbone and forebone are important measures.

Ranking the algorithms by IPL we find that $A < B < C < D$; that is, A produces the most efficient trees, while the successive *improvements* in B and C actually make the long term results increasingly worse. Algorithm D, which has very little asymmetry, but does not have the rebalancing effects present in C, produces the worst trees after sufficiently many updates. Typical of these are the results from trees of 1024 nodes displayed in figure 3. Note that not only does algorithm B have a larger IPL than does A, but that it also degenerates more quickly. This is not surprising given the high asymmetry of algorithm B. Notice also that both algorithms C and D have greater initial improvement than the others, probably because the skewing effects take much longer to produce an effect, thus allowing the rebalancing effects longer to improve the tree.

The same ranking occurs when we take the skewness measure, and when we examine the backbone length versus forebone length, displayed as $(|backbone| - |forebone|)/|shell|$ in figure 4, showing that indeed the degeneration is closely related to the asymmetry of the algorithm.

Algorithm D removes most of the rebalancing effects of algorithm C, and as one might expect, the tree deteriorates even further under D than under C. It is still surprising nevertheless that given the small degree of asymmetry the tree becomes so degenerate.

Size	A	B	C	D
512	5704.9	7438.9	9101.8	11096.1
1024	14496.1	23115.6	33021.2	37878.5
2048	37806.5	72878.2	122330.7	130484.3
4096	98220.1	319805.8	462874.8	533726.5

Table 1: Average IPL of the Asymmetric Algorithms

In table 1 we show the final IPL of the algorithms for the various sizes. For reference the expected IPL given trees generated by a random insertion sequence are included in table 3. We do not have enough points to verify

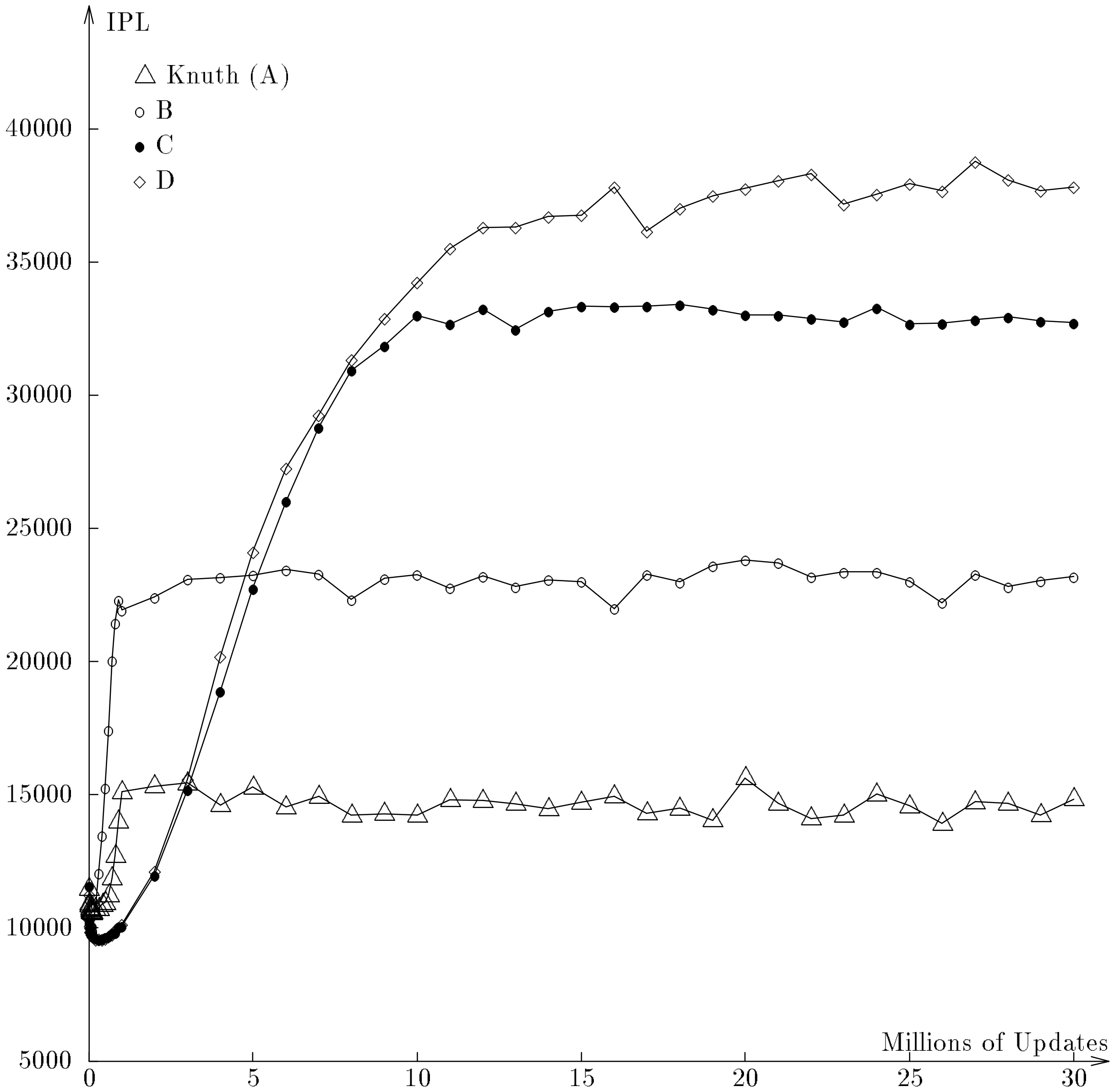

Figure 3: Comparison of IPL on $n = 1024$ for A, B, C, D

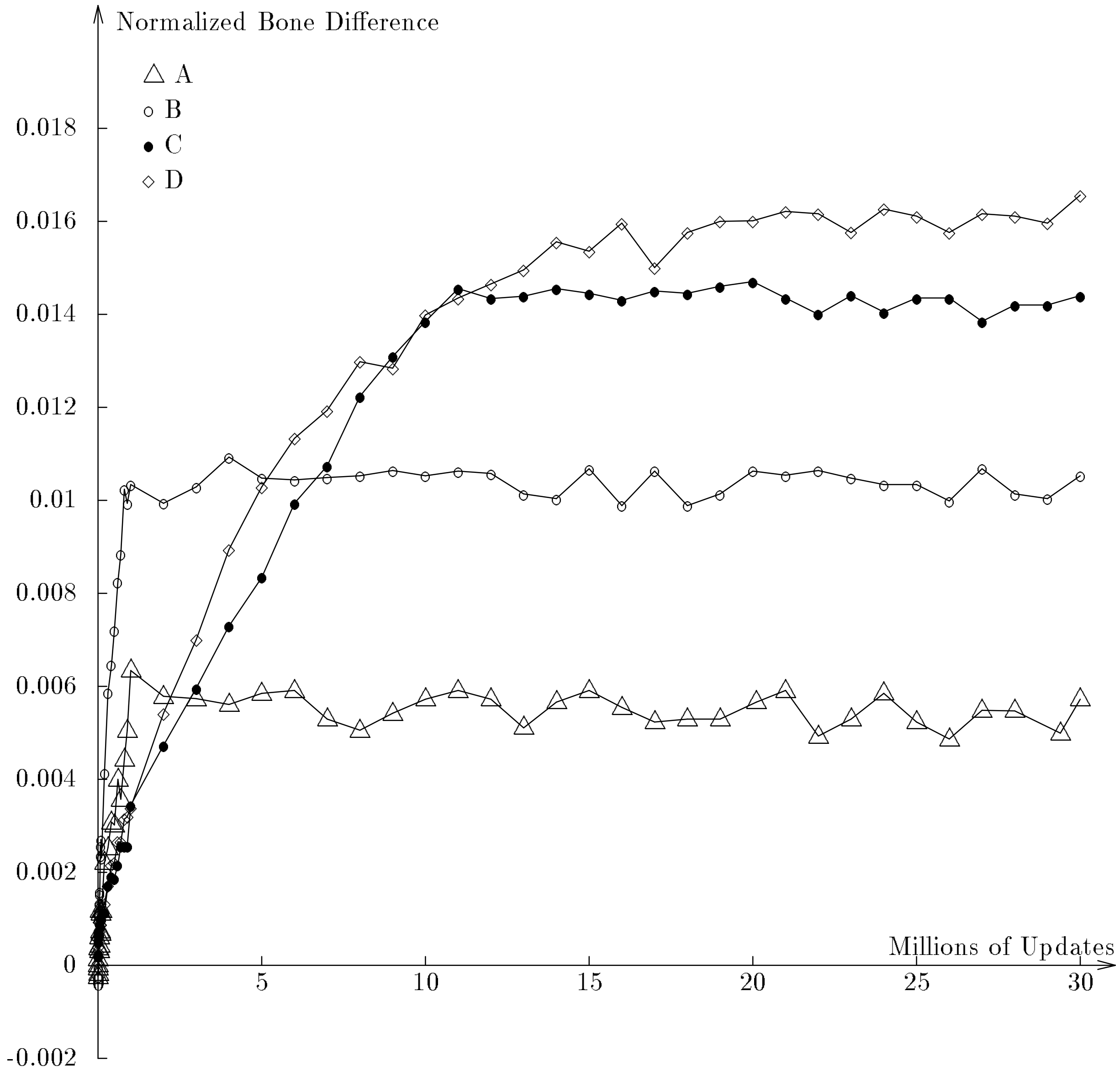

Figure 4: Comparison of Normalized Bone Differences for A, B, C, D

precisely how fast the IPL is growing with n for these algorithms, but clearly it is much faster than the $0.266n\sqrt{n}$ conjectured for Hibbard and A in [3]. In table 2 we give the leading coefficients from regression for each of the algorithms, where the fit is to $n^{3/2} + n \log_2 n + c$. Note the close agreement of A with the conjectured value. It seems clear however, from the data in table 1 that algorithms B through D are growing faster than a constant times the rate of A.

Algorithm	A	B	C	D
Coefficient	0.261	2.99	3.49	4.54

Table 2: Regression Coefficient of $n^{3/2}$

As mentioned above, symmetric versions of C and D, called E and F respectively, yield improved IPL and show no signs of skewing. In [3], the IPL for the symmetric Knuth algorithm for $n = 512$ and 1024 is given as 4332.47 and 9840.40 respectively. Comparing the results in table 3, these algorithms appear to be better than even Knuth's algorithm in terms of the efficiency of the resulting trees. In fact, these results are very close to optimal balanced trees. [1]

Size	IPL E	IPL F	Expected IPL
512	3845.27	3849.67	4945.75
1024	8706.25	8716.42	11297.83
2048	19490.10	19514.67	25420.13
4096	43434.36	44268.76	56502.52

Table 3: Comparison of IPL from Symmetric Algorithms

On the other hand, the average length of the backbone (and the forebone) increase over the expected length of a random tree by a factor of 1.2 to 1.3,

[1] 512 and 1024 were run for 100 million updates, and the means computed from sampling conducted every 10 thousand updates over the last 50 million. 2048 ran for 150 million updates and sampling was from last 20 million. 4096 ran for 300 million and sampling was from the last 30 million. It is not absolutely clear that the absolute minimum was obtained in the latter cases, but the values appeared to be reasonably constant. The same sampling was done for the backbone measurement in table 4.

as is evident in table 4. This result is surprising and just a little cautionary.

Size	Backbone E	Backbone F	Expected (H_n)
512	8.64	8.59	6.82
1024	9.62	9.58	7.51
2048	10.55	10.77	8.20
4096	11.68	11.06	8.90

Table 4: Comparison of Backbones from Symmetric Algorithms

Is it possible that for sufficiently large trees (presumably extremely large) the effect of lengthening the shell of the tree could outweigh the rebalancing effects and produce trees that are less efficient than random trees? Are there other surprises waiting for us in the various symmetric deletion algorithms that are so far believed to produce efficient trees?

Appendix I: Definitions

If v is a node in a binary tree, then we refer to the left subtree as $l(v)$ and the right as $r(v)$.

- Backbone — the path from the root of the tree to the smallest valued element in the tree, situated at the extreme left of the tree, including the root.
- Forebone — the path from the root of the tree to the largest valued element in the tree, situated at the extreme right of the tree, including the root.
- Shell — the shell of a tree is composed of the forebone and backbone of the tree.
- Kernel — all nodes that are not part of the shell of the tree.
- Successor — the successor of a given node is the node in the given node's right subtree which has the least value.
- Successor path — the path from a given node to its successor. All nodes in this path are contained in the right subtree of the given node. Does not include the given node.

- Predecessor — the predecessor of a given node is the node in the given node's left subtree which has the greatest value.
- Predecessor path — the path from a given node to its predecessor. All nodes in this path are contained in the left subtree of the given node. Does not include the given node.
- Depth(v,w) — the number of nodes in a path from v to w, including w but not v, where v is an ancestor of w. If v is not specified, it is assumed to be the root of the tree. Is equivalent to $|path(v, w)|$ where $path(v, w)$ is the path from v to w.
- Local subtree — the subtree with the given node as its root.
- Local characteristics — the characteristics, such as node distribution, of the local subtree.
- Internal path length (IPL) — the total over all nodes v of $depth(root, v)$.
- Harmonic n — defined as $H_n = \sum_{k=1}^{n} 1/k$.
- Skewness — a measure of the normalized relative balance of a tree. Calculated as :

  $$\frac{(sleft - sright)}{(sleft + sright)}$$

  where

  $$sleft = \sum_{v \in T} \frac{|l(v)|}{2^{depth(root,v)}}$$

  $$sright = \sum_{v \in T} \frac{|r(v)|}{2^{depth(root,v)}}$$

  This measure tells us if the tree is skewed to the left or right. A value of zero implies the tree is left–right balanced, but not necessarily of small IPL.
- Update — an update consists of one deletion followed by one insertion. For our purposes, the key to be deleted is selected randomly and equiprobably over the set of n nodes in the tree. The new key for the insertion is chosen randomly using the uniform distribution over the interval (0,1).

Appendix II: Proof outline of lemma 3.1

We list here the lemmas that lead to the result stated in the paper relating the shell to the predecessor and successor paths.

Lemma 3.2 *Every node in a binary tree is part of at most one predecessor path and one successor path.*

Lemma 3.3 *Every node on the forebone of a tree is not part of a predecessor path, and every node on the backbone of a tree is not part of a successor path.*

Lemma 3.4 *Except the root, all nodes in the backbone are part of exactly one predecessor path, and all nodes in the forebone are part of exactly one successor path. The root is not part of either type of path.*

Lemma 3.5 *All nodes in the kernel of a tree are part of exactly one predecessor path and one successor path.*

Lemma 3.6 *The total length of all predecessor paths is* $n - |forebone|$

Proof: All the nodes in the tree (n of them) can be divided into three groups, those internal, those on the forebone, and those on the backbone (except the root, which is on the forebone). All internal nodes and those on the backbone, except the root, are part of exactly one predecessor path, and thus each contribute 1 to the total predecessor depth. All nodes on the forebone, including the root, are not part of any predecessor path, thus they do not contribute to the total predecessor depth. Thus we have :

$$\sum_v depth(v, predecessor) = |kernel| + |backbone| - 1$$

But $|kernel| = n - |backbone| - |forebone| + 1$ as we cannot count the root twice.
So, $\sum_v depth(v, predecessor) = n - |forebone|$. ■

By symmetry we have

Lemma 3.7 *The total length of all successor paths is* $n - |backbone|$*.*

Combining the last two lemmas, and noting that the root occurs in both the forebone and the backbone,

THEOREM **3.1** *The total depth of all predecessor and successor paths is* $2n - 1 - |shell|$*.*

References

[1] Ricardo A. Baeza-Yates. Analysis of algorithms in search trees. Master's thesis, Universidad de Chile, Santiago, Chile, January 1985.

[2] A. D. Booth and A. J. T. Colin. On the efficiency of a new method of dictionary construction. *Information and Control*, 3:327–334, 1960.

[3] Joseph Culberson and J. Ian Munro. Analysis of the standard deletion algorithms in exact fit domain binary search trees. *Algorithmica*, 6:295–311, 1990.

[4] Joseph C. Culberson and J. Ian Munro. Explaining the behavior of binary search trees under prolonged updates: A model and simulations. *The Computer Journal*, 32(1):68–75, February 1989.

[5] Jeffery L. Eppinger. An empirical study of insertion and deletion in binary trees. *Communications of the ACM*, 26, September 1983.

[6] Thomas N. Hibbard. Some combinatorial properties of certain trees with applications to searching and sorting. *Journal of the ACM*, 9(1):13–28, January 1962.

[7] Arne T. Jonassen and Donald E. Knuth. A trivial algorithm whose analysis isn't. *Journal of Computer and System Sciences*, 16:301–322, 1978.

[8] Gary D. Knott. *Deletion in Binary Storage Trees*. PhD thesis, Stanford University, May 1975. Avail. as Tech. Rep. STAN-CS-75-491.

[9] D. E. Knuth. *Sorting and Searching*, volume III of *The Art of Computer Programming*. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1973.

[10] C. J. Stephenson. A method for constructing binary search trees by making insertions at the root. *International Journal of Computer and Information Sciences*, 9:15–29, 1980.

[11] P. F. Windley. Trees, forests and rearranging. *The Computer Journal*, 3:84–88, July 1960.