SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 23(11), 1267–1283 (NOVEMBER 1993)

ISA[*k*] Trees: a Class of Binary Search Trees with Minimal or Near Minimal Internal Path Length

FARIS N. ABUALI AND ROGER L. WAINWRIGHT
Department of Mathematical and Computer Sciences, The University of Tulsa, 600 South College Avenue, Tulsa, Oklahoma 74104-3189, U.S.A.

SUMMARY

In recent years several authors have investigated binary search trees with minimal internal path length. In this paper we propose relaxing the requirement of inserting all nodes on one level before going to the next level. This leads to a new class of binary search trees called ISA[*k*] trees. We investigated the average locate cost per node, average shift cost per node, total insertion cost, and average successful search cost for ISA[*k*] trees. We also present an insertion algorithm with associated predecessor and successor functions for ISA[*k*] trees. For large binary search trees (over 160 nodes) our results suggest the use of ISA[2] or ISA[3] trees for best performance.

KEY WORDS Binary search trees Inorder shifting Internal path length Optimal searching

INTRODUCTION

Binary search trees are very popular and useful data structures for efficiently searching, inserting and deleting data items. Unfortunately, binary search trees can have a worst case search path length of $n-1$, where n is the number of nodes in the tree. However, a best case search path length of $\log_2 n$ is possible for a balanced tree. A binary search tree that grows in a random manner will have an expected search path length of $1{\cdot}386 \log_2 n$.[1] Thus methods that maintain balance in binary search trees have received considerable attention recently. These algorithms can offer on the average as much as 28 per cent better performance over randomly constructed binary search trees.

Binary search tree rebalancing strategies are either local or global. Local rebalancing strategies detect an imbalance at a particular node in the search tree and restores the balance of the tree to within the specified limits. The most widely used local rebalancing strategy is the ‘AVL rotations’ for height balanced trees. Other local rebalancing strategies, including BB trees and weight balanced trees, are given in Reference 2. A local strategy geared on the total internal path length of the tree is given in Reference 3. Global rebalancing strategies are generally much more expensive to perform than local rebalancing strategies. In the global case, generally all of the nodes of the tree are inspected in order to restore the tree to complete balance. Various global rebalancing strategies are given in References 4 and 5.

0038–0644/93/111267–17$13.50

Received 4 June 1992
Revised 24 May 1993

Gerasch[6] has introduced a balancing algorithm for binary search trees. His algorithm has a run-time performance that falls between that of the local and global rebalancing algorithms. The significance of Gerasch's balancing algorithm is that it produces trees that have minimum internal path length. His insertion algorithm maintains the minimal internal path length of displacing keys when necessary, in an inorder (infix) fashion, until a vacant position is located somewhere in the last level of the tree.The following definition comes from Reference 6.

Definition 1

A binary tree with n nodes ($n>0$) is defiined to be *nearly complete* to level L,if it is complete to level $L-1$ and level L is either empty or is the last non-empty level of the tree.

The root of a tree is defined to be at level zero. Clearly, binary trees that are nearly complete to level L are those trees with minimal internal and external path lengths. The insertion algorithm that maintains the minimal internal path length is the inorder shift algorithm described in Reference 6. Obviously, the minimal insertion cost is $O(\log_2 n)$ when no reorganization is required. However, in an extreme case $O(n)$ cost may be required if there is only one free entry in level L of the tree and the inorder shifting involves all the nodes. If an insertion of a node is beyond level L, and there are free locations at level L on both sides of the insertion position, then the inorder shift could be done in either direction. In this case it is important to choose the direction which will displace the fewest nodes of the tree. Gerasch's algorithm does not always choose the optimal direction. An improvement to Gerasch's algorithm that chooses the optimal shift direction is presented in Reference 7.

In our previous research,[8] we developed several results related to the expected number of various types of fringe nodes for trees described by Definition 1. Furthermore, we refer to the tree defined in Definition 1 as an *ISA* tree since the tree maintains its balance by displacing keys, when necessary, using an inorder shift algorithm (ISA). We defined the shift cost of the ISA algorithm as the minimum number of nodes that change values during the inorder shift. In most cases an inorder shift can be performed either to the right or to the left of the insertion point, and in all cases the minimum cost shift was used.

Figure 1 shows the shift cost of inserting nodes on a given level. As expected, the shift cost is zero for the first node on each level of an ISA tree. Notice that the shift cost increases very slowly until a certain point, then increases very sharply as the level begins to fill up. The shift cost of inserting the last few nodes on a given level is as much or more than inserting all of the other nodes on the level. It is interesting to note the cost midpoint (crossover point) on each level. The crossover point, x, on a given level occurs x nodes from the end of the level. That is, at what point on a given level is the sum of the cost of inserting previous nodes on the level equal to the sum of the cost of inserting the remaining x nodes of the level? We inserted 8195 random nodes into an ISA tree and noted the shift cost for each node. We repeated the experiment for 10 random trees and recorded the average shift cost. The crossover points are shown in Figure 2. For example consider level 10 (1024 nodes), where the crossover point is 36. That is, the cost crossover

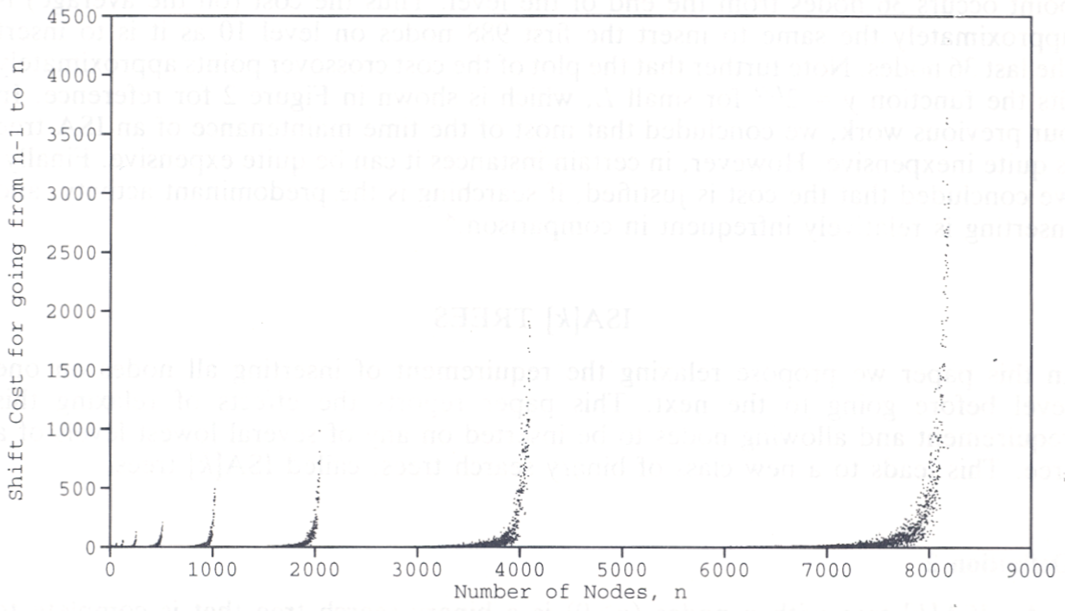


Figure 1. Shift cost for a random ISA[1] tree

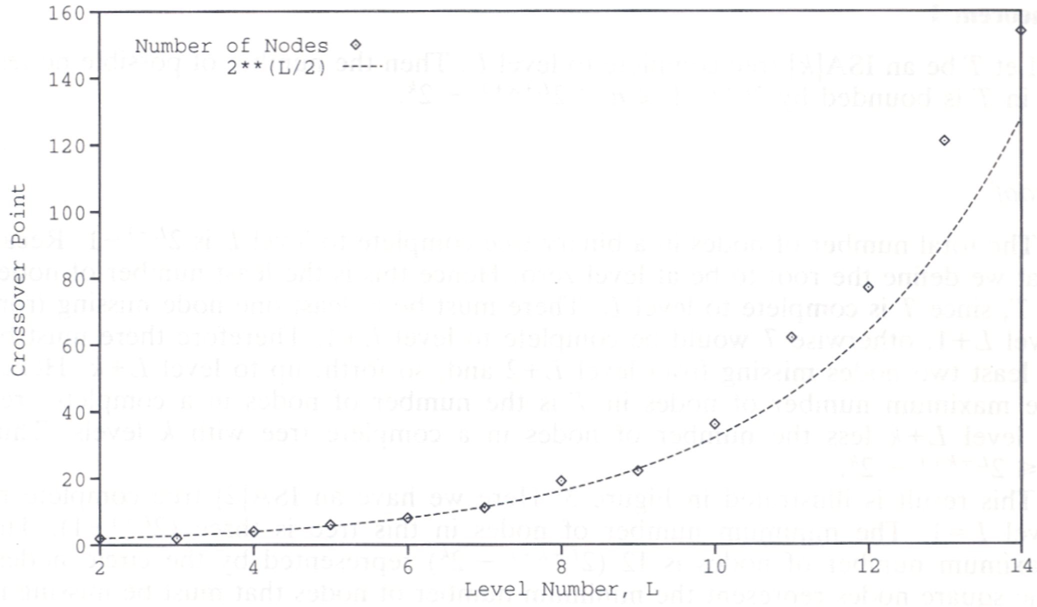


Figure 2. Shift cost crossover point for a random ISA[1] tree

point occurs 36 nodes from the end of the level. Thus the cost (on the average) is approximately the same to insert the first 988 nodes on level 10 as it is to insert the last 36 nodes. Note further that the plot of the cost crossover points approximately fits the function $y = 2^{L/2}$ for small L, which is shown in Figure 2 for reference. In our previous work, we concluded that most of the time maintenance of an ISA tree is quite inexpensive. However, in certain instances it can be quite expensive. Finally, we concluded that the cost is justified, if searching is the predominant activity, and inserting is relatively infrequent in comparison.[8]

ISA[k] TREES

In this paper we propose relaxing the requirement of inserting all nodes on one level before going to the next. This paper reports the effects of relaxing this requirement and allowing nodes to be inserted on any of several lowest levels of a tree. This leads to a new class of binary search trees, called ISA[k] trees.

Definition 2

An ISA[k] tree with n nodes ($n>0$) is a binary search tree that is complete to level L, and levels $L+1$ to $L+k$ ($k>0$) are all either empty or are the last non-empty levels of the tree.

Notice that all trees satisfying Definition 1 are ISA[1] trees. Figure 4(a) illustrates an example ISA[2] tree. The tree is complete to level $L=1$, and levels $L+1$ and $L+2$ are not complete.

Theorem 1

Let T be an ISA[k] tree complete to level L. Then the number of possible nodes, n, in T is bounded by $2^{L+1}-1 \leqslant n \leqslant 2^{L+k+1} - 2^k$.

Proof

The total number of nodes in a binary tree complete to level L is $2^{L+1}-1$. Recall that we define the root to be at level zero. Hence this is the least number of nodes in T, since T is complete to level L. There must be at least one node missing from level $L+1$, otherwise T would be complete to level $L+1$. Therefore there must be at least two nodes missing from level $L+2$ and, so forth, up to level $L+k$. Hence the maximum number of nodes in T is the number of nodes in a complete tree to level $L+k$ less the number of nodes in a complete tree with k levels. Thus $n \leqslant 2^{L+k+1} - 2^k$.

This result is illustrated in Figure 3. Here we have an ISA[2] tree complete to level $L=1$. The minimum number of nodes in this tree is three ($2^{L+1}-1$). The maximum number of nodes is 12 ($2^{L+k+1} - 2^k$) represented by the circle nodes. The square nodes represent the minimum number of nodes that must be missing in the tree and still be complete to level L.

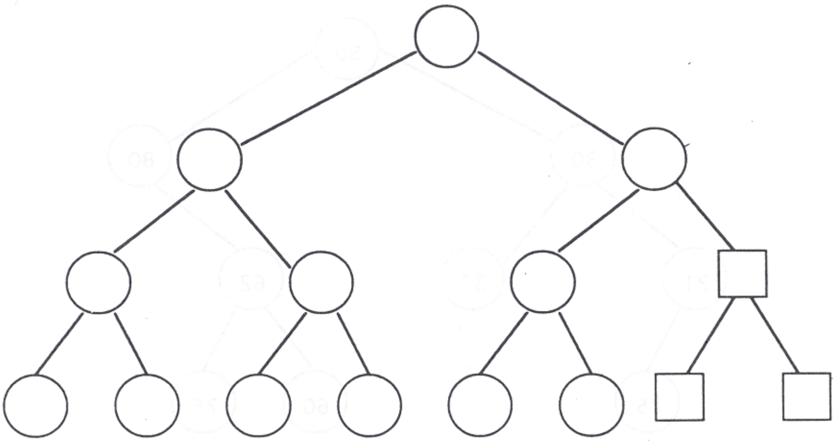


Figure 3. The maximum number of nodes in an ISA[2] tree, with L=1

Theorem 2

Let T be an ISA[k] tree with n nodes. Then the bounds for the highest numbered level L are given by $\text{ceiling}(\log_2(n/2^k+1)-1) \leqslant L \leqslant \text{trunc}(\log_2(n+1)-1)$.

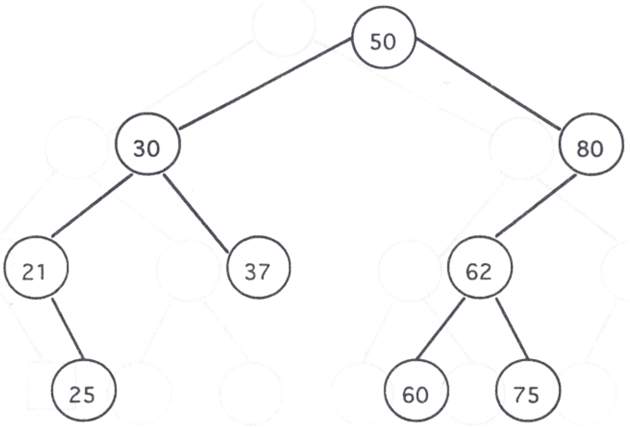
Proof

The proof follows immediately from Theorem 1, solving for L.

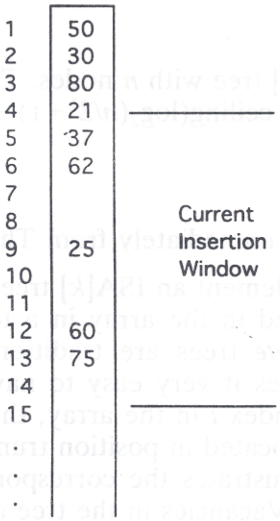
We elected to implement an ISA[k] tree using an array data structure. The nodes of the tree are placed in the array in a top-down, left to right order. Heaps and other *almost complete* trees are traditionally implemented in this manner. This implementation makes it very easy to navigate through the tree: given a node in the tree located at index i in the array, the children are located in positions $2i$ and $2i+1$; the parent is located in position trunc($i/2$), provided of course that the values exist. Figure 4(b) illustrates the corresponding array data stucture for the ISA[2] tree in Figure 4(a). Vacancies in the tree correspond to empty array positions. The algorithm to insert a key into an ISA[k] tree is given below.

The insertion algorithm for an ISA[*k*] tree

1. Insert the key using the traditional binary search tree insertion algorithm.
2. If the location of the inserted key violates the ISA[k] restriction (Definition 2) then
   (a) determine the 'best' direction to perform the inorder shift. This is accomplished by repeatedly calling the Successor routine and repeatedly calling the Predecessor routine from the violated position in the tree attempting to locate a valid empty location to place the last shifted element. The direction which requires the minimum number of shifts is selected.
   (b) perform the inorder shift, in the direction determined in part (a). The



(a)



(b)

Figure 4. (a) An example ISA[2] tree; (b) corresponding array data structure

Predecessor and Successor routines are used to relocate (shift) the keys in an inorder fashion.

The number of times the Predecessor and Successor functions are called in Step 2(a) in the above algorithm is dependent on the structure of the tree at the time and how full each level is. The cost of Step 2(a) ranges from $O(1)$ to $O(n)$. However, in our data collection experiments, most of the time Step 2 was not performed; see Figure 1.

In each ISA[k] tree, a valid insertion window is maintained. The window represents valid insertion levels where the inserted key can be placed without violating Definition 2. The levels of an ISA[k] tree, which constitute the insertion window, are manipulated by two indices in the array implementation of the ISA[k] tree. The two indices indicate the range of valid insert positions. Since $k=2$ and $L=1$ in Figure 4(a), the valid insertion window is levels two and three. This is shown in Figure 4(b) where positions four to 15 are the current valid insert locations. Maintaining the current insertion window is relatively simple. After each insertion, level $L+1$ is checked to see if it is complete. If so, the insertion window is shifted down by one level. For example, suppose node 90 is inserted into Figure 4(a). The ISA[2] tree becomes complete to level two. This causes the current insertion window of levels two and three (as shown in Figure 4(b)) to slide down to become levels three and four. This would translate to positions 8 to 31 in the array in Figure 4(b).

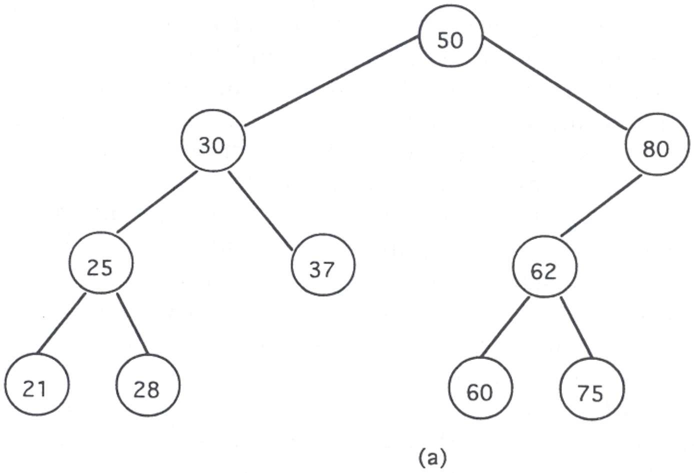
The predecessor (successor) of a node in an ISA[k] tree is the node with the next smaller (larger) key value. This is the standard definition for predecessor and successor. These functions look for an open position in the tree within the valid insertion window in what would be predecessor or successor positions. For example, consider inserting key 28 into the ISA[2] tree in Figure 4(a). This should be inserted as the right child of node 25. This translates to position 19 in the array data structure in Figure 4(b). However, the insertion is outside the valid window. The Predecessor function will first look at node 25, noting that the left child position of node 25 is outside the valid insertion window. It will then proceed to node 21 noting that its left child position is within the valid insertion window. Therefore adjusting the ISA[2] tree via the Predecessor function involves moving nodes 21, 25 and 28 at a cost of three keys involved. This is illustrated in Figure 5(a). Now consider the Successor function after inserting key 28 into Figure 4(a). The successor to 28 is node 30 which has both its children positions occupied. Thus the next successor position is considered, which is key 37. The left child of key 37 is checked. If it is vacant and within the insertion window, the left child of key 37 would represent the next valid successor position. Since it is an open and valid position, the Successor function is finished. Hence, adjusting the ISA[2] tree via the Successor function involves moving key 28 to 30's position, and moving key 30 to the left child location of key 37. This is illustrated in Figure 5(b). This involves the movement of only two keys compared to three keys for the Predecessor function. Thus in this case the Successor function is the chosen direction for shifting.

The Predecessor and Successor functions for an ISA[k] tree are shown in Figures 6 and 7 in abstract form. Note that the functions Left_child, Right_child, and Parent are the same as the traditional functions for traversing binary trees. We found all of these functions very simple to implement when an ISA[k] tree is represented as an array.

PERFORMANCE ANALYSIS FOR ISA[k] TREES

Construction costs

ISA[k] trees are a class of trees that offer minimum or near minimum search times. We analyzed numerous ISA[k] trees to determine the cost required for construction and maintenance. We constructed ISA[k] trees of size n where $n = (2^i-1) + [(j-1)\ 2^{(i-2)}]$, such that $3 \leq i \leq 12$ and $1 \leq j \leq 4$. That is, we tested 40 tree sizes representing powers of two (less one) and three equal distant values between each power of two, namely 7, 9, 11, 13, 15, 19, 23, 27, 31, . . ., 2047,



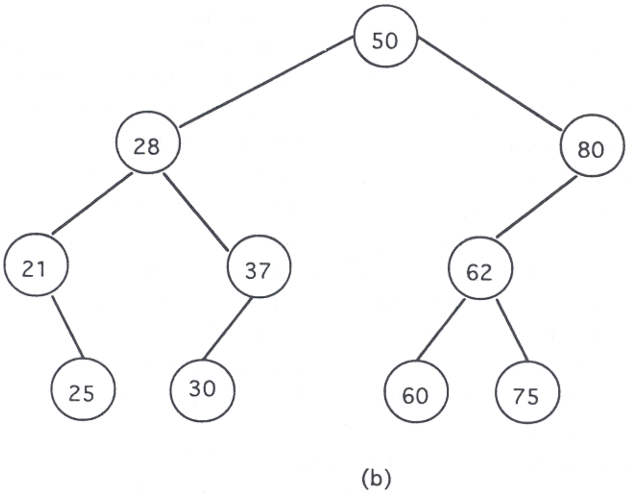


Figure 5. Insert Node 28 into Figure 4(a): (a) using the Predecessor *function; (b) using the* Successor *function*

2559, 3071, 3583, 4095, 5119, 6143 and 7167. In addition, for each of the 40 tree sizes we constructed ISA[k] trees for $1 \leq k \leq 5$. Furthermore, for each ISA[k] tree of size n and for each k, ten different random trees were constructed and the average performance of the ten trees was recorded. This represents a total of 2000 trees.

Definition 3

The locate cost for inserting a key in an ISA[k] tree is the same as an unsuccessful search cost. That is, the height of the key after insertion into the tree.

```
Function Predecessor(NODE)
/*  This function assumes an array implementation of a tree    */
/*  as described in Figure 4.  A single call to this function  */
/*  returns the location in the array of the predecessor of    */
/*  NODE.  Note this location may be empty (vacant) of may     */
/*  have a value in it.  Furthermore, if the location is vacant */
/*  this function guarantees the location is within the        */
/*  insertion window.                                          */

{
  if NODE is a leaf then
  {
    if NODE the first node on that level then
    {
      if Left_child(NODE) is within insertion window then
        return Left_child(NODE)
      else
        return NO_PREDECESSOR_FOUND
    }
    if Left_child(NODE) is within insertion window then
      return Left_child(NODE)
    while NODE is a left child
      NODE = Parent(NODE)
    return Parent(NODE)
  }
  else /* not a leaf node */
  {
    NODE = Left_child(NODE)
    if NODE is an empty location then return(NODE)
    while NODE is not a leaf
      NODE = Right_child(NODE)
    if NODE is not an empty location then
    {
     if Right_child(NODE) is within insertion window then
       return Right_child(NODE)
     else
       return NODE
    }
    else
     return NODE
  }
}
```

Figure 6. Predecessor *function*

Definition 4

The shift cost for inserting a key is the cost of maintaining the ISA[k] tree (beyond the locate cost of the key). It is the number of predecessor or successor calls required to complete the inorder shift. The average shift cost for inserting n keys into a tree is the sum of all n shift costs divided by n.

```
Function Successor(NODE)
/*  This function assumes an array implementation of a tree     */
/*  as described in Figure 4.  A single call to this function   */
/*  returns the location in the array of the successor of       */
/*  NODE.  Note this location may be empty (vacant) of may      */
/*  have a value in it.  Furthermore, if the location is vacant */
/*  this function guarantees the location is within the         */
/*  insertion window.                                           */

{
  if NODE is a leaf then
  {
    if NODE the last node on that level then
    {
      if Right_child(NODE) is within insertion window then
         return Right_child(NODE)
      else
         return NO_SUCCESSOR_FOUND
    }
    if Right_child(NODE) is within insertion window then
       return Right_child(NODE)
    while NODE is a right child
      NODE = Parent(NODE)
    return Parent(NODE)
  }
  else /* not a leaf node */
  {
    NODE = Right_child(NODE)
    if NODE is an empty location then return(NODE)
    while NODE is not a leaf
       NODE = Left_child(NODE)

    if NODE is not an empty location then
    {
     if Left_child(NODE) is within insertion window then
       return Left_child(NODE)
     else
       return NODE
    }
    else
      return NODE
  }
}
```

Figure 7. Successor *function*

Note that if a node is inserted into a valid window location in the tree, there will be no shift cost, only a locate cost.

Definition 5

The insertion cost of an ISA[*k*] tree of size *n* is the cost of inserting each key in the tree. That is, insertion cost = Σ (locate cost(*i*) + shift cost(*i*)), $1 \leq i \leq n$.

Locate cost(*i*) and shift cost(*i*) are dependent on the particular list of keys preceding the *i*th key. That is, the cost is dependent on the current shape of the tree when the *i*th node is inserted. These functions do not represent average locate and shift costs for the *i*th node over all possible tree shapes. We use this definition to determine insertion cost empirically as we construct random ISA[*k*] trees of various sizes.

Definition 6

The total successful search cost of an ISA[*k*] tree of size *n* is Σ [(number of nodes at level *i*) × *i*), $0 \leq i \leq$ last level. This is the same as the traditional definition for the internal path length of a tree. Thus the average successful search cost for a node in an ISA[*k*] tree of size *n* is the total successful search cost/*n*.

During construction of each ISA[*k*] tree the following statistics were collected: (1) the locate cost for each insertion, (2) the shift cost for each insertion, (3) the total insertion cost of the tree, (4) the number of insertions requiring an inorder shift and (5) the average successful search cost of the tree after construction was completed.

In Figures 8–13, for each tree size *n* and each value of *k*, ten random trees were generated and the average performance of the ten trees is depicted. Figure 8 depicts the average locate cost for all of the trees tested (the sum of the locate cost to insert all *n* nodes/*n*). Notice that the average locate cost is $O(\log_2 n)$ as expected, and appears to be independent of *k*, when $k<6$.

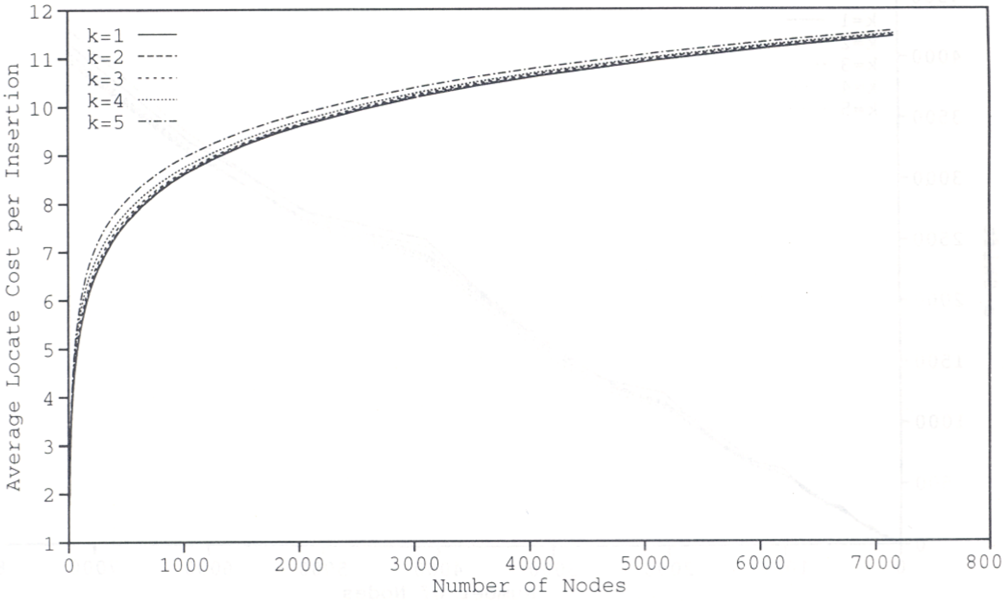


Figure 8. Average locate cost for a random ISA[k] tree

Figure 9 shows the number of insertions requiring a shift. Results indicate that the value of k has little impact on the amount of rebalancing required. Incrementally there is little change in the amount of rebalancing between an ISA[k] and ISA[$k+1$] tree for values of k under six. This is an interesting result. For example, for tree size 7167 the difference between $k=2$ and $k=5$ is about 3 per cent. Furthermore, Figure 9 shows that the number of insertions requiring rebalancing ranged from 55 to 58 per cent and appears to depend very little on the size of the tree or k. Given an ISA[k] tree with n nodes, the probability of inserting a node outside of the insertion window is $2x/(n+1)$, where x is the number of nodes on the lowest level. Our results show that inserting the same n nodes into an ISA[$k+1$] tree yields approximately (within a few per cent) the same number of nodes on the lowest level. This result applies to random trees and may not apply to other types of trees. Note that the amount of rebalancing is not totally indepenent of k, however, since rebalancing goes to zero as k goes to infinity.

Figure 10 shows the average shift cost per node for all tree sizes and k values tested. Figure 11 shows an enlargement of Figure 10 for tree sizes $\leqslant 1000$. The value of k has a tremendous effect on the shift cost. Results show that for small size trees (around 160 or less, Figure 11) larger k values tend to reduce the shift cost. However, for larger tree sizes (over 160 or so) the larger k values become much more expensive than smaller k values. The number of insertions requiring a shift varies little with k (Figure 9). Hence the contributing factor for shift cost for any tree is the shift cost of each individual insertion. This implies, for larger k, that individual shift costs are more expensive. We believe that the reason for this is the clustering of nodes within the insertion window. Small clusters tend to breed larger clusters, since any node inserted into a cluster will increase the cluster size by one.

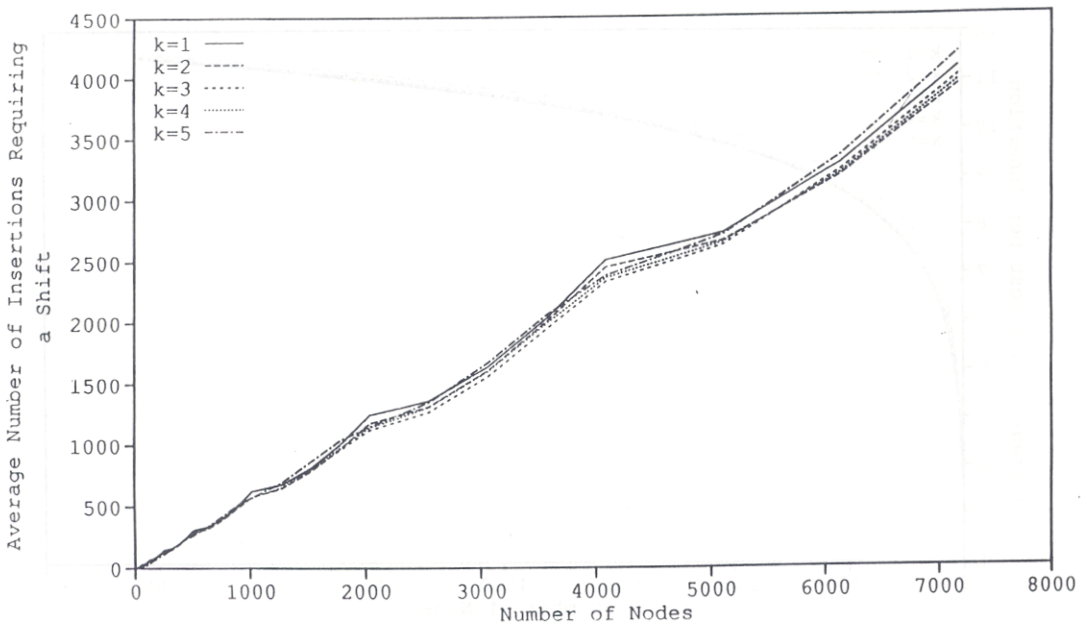


Figure 9. Average number of insertions requiring a shift for a random ISA[k] tree

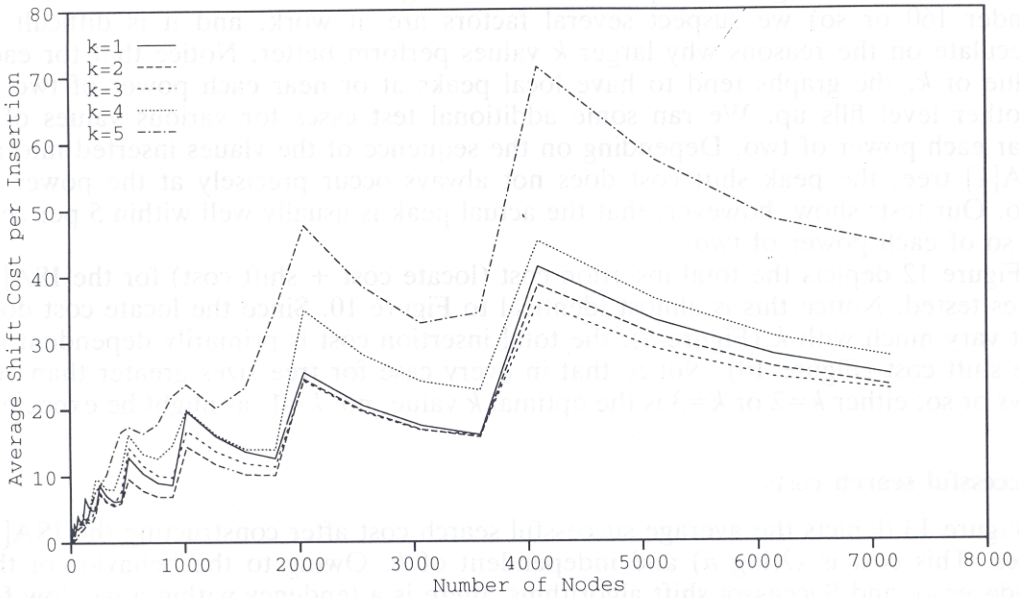


*Figure 10. Average shift cost for a random ISA[*k*] tree*

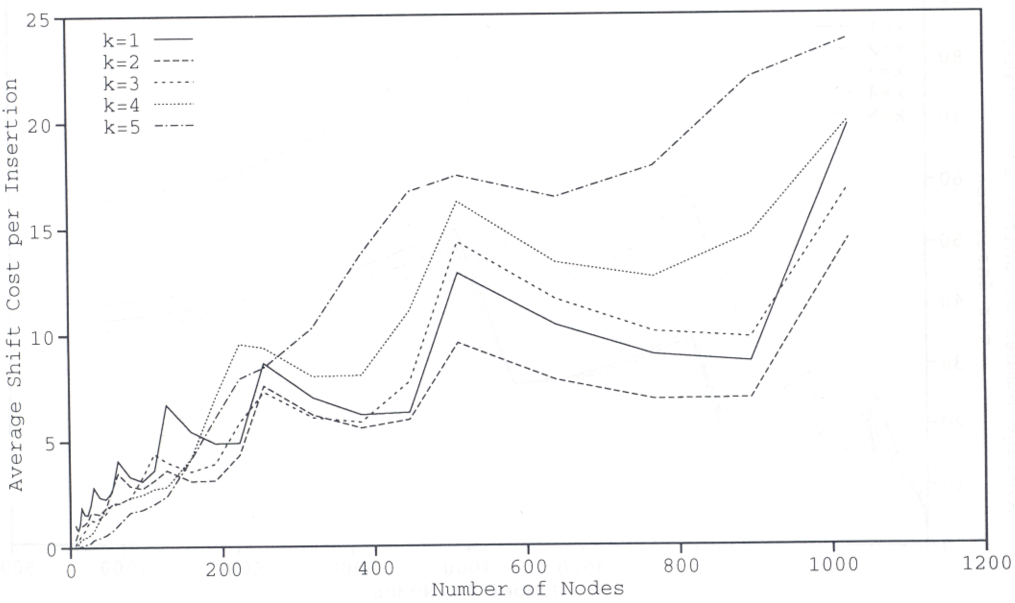


*Figure 11. Average shift cost for a random ISA[*k*] tree (first 1000 keys)*

The cost of clustering is more pronounced for larger tree sizes. In smaller tree sizes (under 160 or so) we suspect several factors are at work, and it is difficult to speculate on the reasons why larger *k* values perform better. Notice that for each value of *k*, the graphs tend to have local peaks at or near each power of two as another level fills up. We ran some additional test cases for various values of *n* near each power of two. Depending on the sequence of the vlaues inserted into an ISA[*k*] tree, the peak shift cost does not always occur precisely at the power of two. Our tests show, however, that the actual peak is usually well within 5 per cent or so of each power of two.

Figure 12 depicts the total insertion cost (locate cost + shift cost) for the ISA[*k*] trees tested. Notice this is almost identical to Figure 10. Since the locate cost does not vary much with *k* (Figure 8), the total insertion cost is primarily dependent on the shift cost (Figure 10). Notice that in every case for tree sizes greater than 160 keys or so, either *k*=2 or *k*=3 is the optimal *k* value, not *k*=1, as might be expected.

Successful search costs

Figure 13 depicts the average successful search cost after constructing the ISA[*k*] trees. This cost is *O*(log *n*) and independent of *k*. Owing to the behavior of the Predecessor and Successor shift algorithms, there is a tendency within a window for the higher levels to fill more rapidly than the lower levels. This appears to be independent of *k* as shown in Figure 13. Notice in Figure 13 that the best case is the same as *k*=1. For comparison the worst case was determined (for *k*=5) and plotted. The interesting result is that the plots for 1⩽*k*⩽5 are all grouped together very close to the best case and well below the worst case.

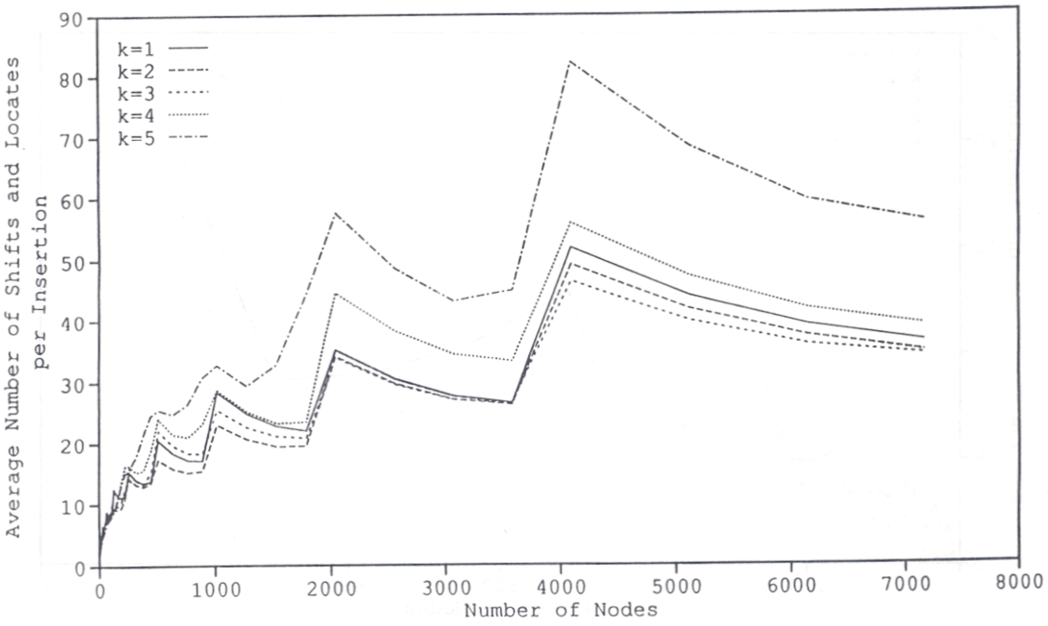


Figure 12. Average construction cost for a random ISA[k] tree

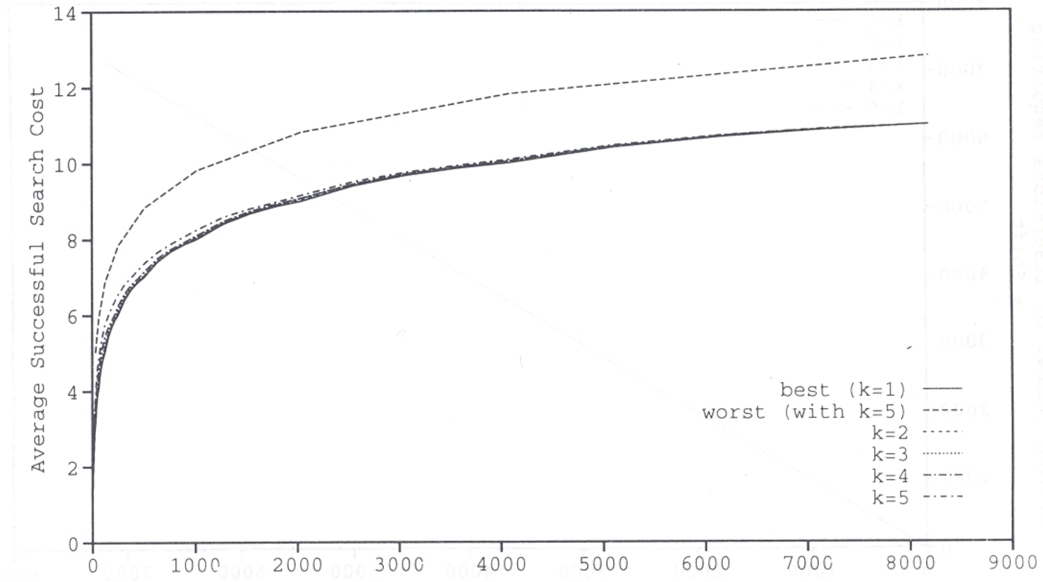


*Figure 13. Average successful search cost for a random ISA[*k*] tree*

Inserting sequential keys into an ISA[*k*] tree

We conducted a second set of experiments on ISA[*k*] trees. How do ISA[*k*] trees behave when constructed from an increasing sequence of keys instead of random? Figure 14 shows the average number of insertions requiring a shift. As expected for a sorted sequence of keys this was nearly 100 per cent. The plots for $1 \leqslant k \leqslant 5$ are indistinguishable and are independent of *k*. Furthermore, Figure 15 shows the average shift cost for a sorted sequence of keys. Results are indistinguishable for the various *k* values. The shift cost peaks at powers of two as expected. However, the shift cost is independent of *k*, which is not true for random keys.

RESULTS AND CONCLUSIONS

Throughout this paper we have assumed the equal likelihood of inserting each key. Thus binary search trees with minimal internal path length offer optimal search times. We proposed relaxing the requirement of inserting all nodes on one level before going to the next level, leading to a new class of binary search trees, ISA[*k*] trees. The results of allowing nodes to be inserted on any of several lowest levels of a tree has proved very interesting. When constructing an ISA[*k*] tree we noticed that the average locate cost is $O(\log_2 n)$ as expected, and is independent of *k*. Results indicate that the value of *k* has little impact on the amount of rebalancing required. That is, the number of insertions requiring a shift is also independent of *k*. However, the average shift cost per node is dependent on *k*. For small size trees (under 160 nodes or so), larger *k* values tend to have lower shift costs. Conversely, for larger trees (over 160 nodes or so) the higher the *k* value, the higher the shift costs.

The important issue for an ISA[*k*] tree is the total insertion cost. Since the locate

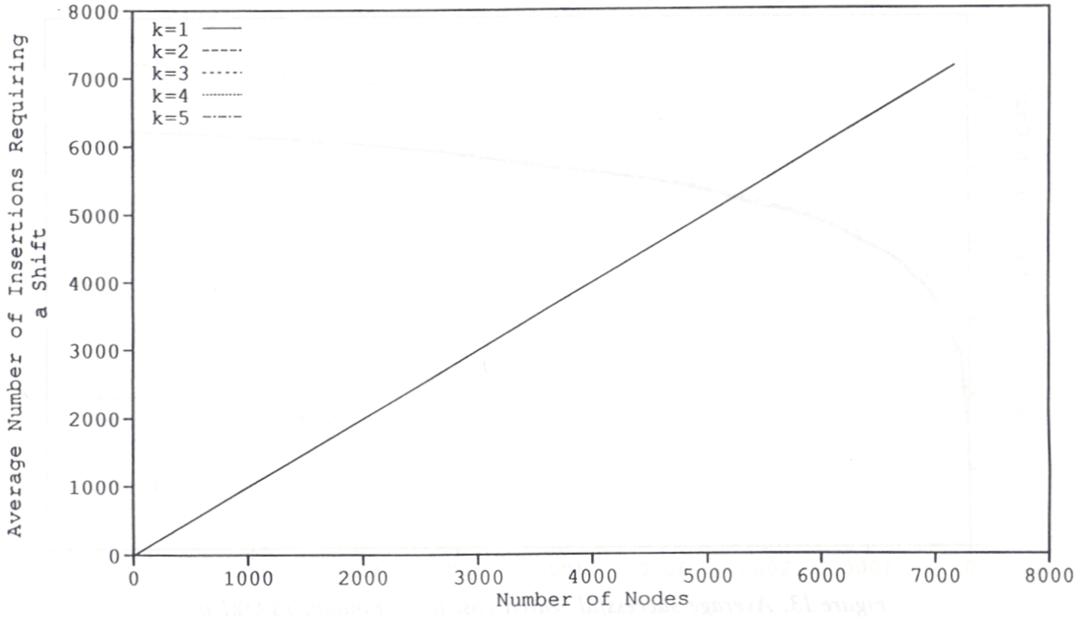


Figure 14. Average number of insertions requiring a shift for a sorted ISA[k] tree

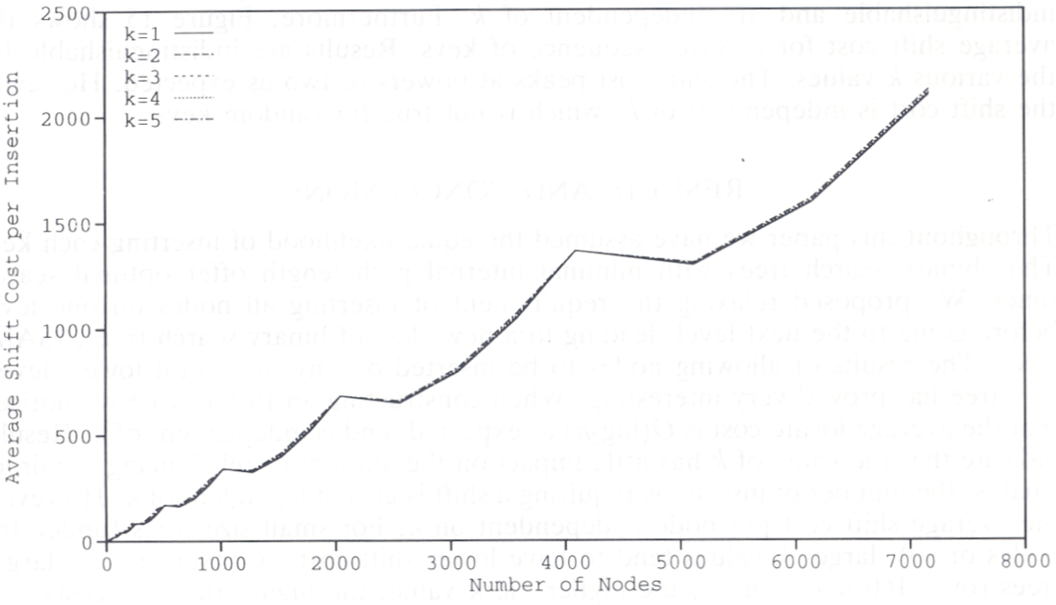


Figure 15. Average shift cost for a sorted ISA [k] tree

cost is independent of k, the total insertion cost is dependent primarily on the shift cost. The important results presented in this paper are (a) the shift cost is the major issue in deciding which k to select, (b) for tree sizes greater than 160 keys or so, either $k=2$ or $k=3$ is the optimal k choice, not $k=1$ as might be expected, and (c) the average successful search costs are relatively independent of k.

Even though the maintenance of an ISA[k] tree can be quite expensive in certain instances, the cost can be justified if searching is the predominant activity, and inserting is relative infrequent in comparison. Therefore for large binary search trees (over 160 nodes) we suggest that the user consider using an ISA[2] or ISA[3] tree.

For future research we are looking at the inorder shift algorithm to improve the selection of the 'best' direction to perform the shift. It seems intuitive to select empty locations in 'unsaturated' regions of the tree. We are investigating the use of a weighted average based on factors such as 'saturation' of the region to the left and to the right of an empty location, rather than the minimum cost predecessor or successor shift.

ACKNOWLEDGEMENTS

This research has been partially supported by OCAST Grant ARO-038. The authors also wish to acknowledge the support of Sun Microsystems Inc. The authors would like to thank the referees for their thorough reading of the manuscript, which resulted in numerous improvements and clarifications.

REFERENCES

1. D. E. Knuth, *The Art of Computer Programming, vol. 3, Sorting and Searching*, Addison-Wesley, 1973.
2. J. Nievergelt and E. M. Reingold, 'Binary search trees of bounded balance', *Siam J. Comput,* **2**, (1), 33–43 (1973).
3. G. H. Gonnet, 'Balancing binary trees by internal path reduction', *Commun. ACM,* **26**, (12), 1074–1081 (1983).
4. H. Chang and S. S. Iyengar, 'Efficient algorithms to globally balance a binary search tree', *Commun. ACM,* **27**, (7), 695–702 (1984).
5. Q. F. Stout and B. L. Warren, 'Tree rebalancing in optimal time and space', *Commun. ACM,* **29**, (9), 902–908 (1986).
6. T. E. Gerasch, 'An insertion algorithm for a minimal internal path length binary search tree', *Commun. ACM,* **31**, (5), 579–585 (1988).
7. I. Richard, 'Technical correspondence on Gerasch's insertion algorithm', *Commun. ACM,* **34**, (2), 79–80 (1991).
8. F. N. Abuali and R. L. Wainwright, 'Fringe analysis of binary search trees with minimal internal path length', *Proceedings of the 1991 Computer Science Conference*, San Antonio, Texas, 5–7 March, 1991, pp. 61–70.