2. S. Berhovitz, J. Kowalchuk and B. Shanning, Implementing the public key scheme. *IEEE Communications Magazine* **17** (3), 2–3 (1979).
3. C. C. Chang and C. H. Lin, A reciprocal confluence tree unit and its applications. *BIT* **30**, 27–33 (1990).
4. D. E. Denning, *Cryptography and Data Security*. Addison-Wesley, Reading, Mass. (1982).
5. W. Diffie and M. Hellman, New directions in cryptography. *IEEE Transactions on Information Theory* **IT-22** (6), 644–654 (1976).
6. G. I. Davida, D. J. Linton, C. R. Szelag and D. L. Wells, Data base security. *IEEE Transactions on Software Engineering* **SE-4** (6) 531–533 (1978).
7. G. I. Davida and D. L. Wells, Microprocessors and data encryption. *Proceedings of IEEE COMPCON* Washington, D.C., 154–159 (1979).
8. G. I. Davida, D. L. Wells and J. B. Kam, A database encryption system with subkeys. *ACM Transactions on Database Systems* **6** (2) 312–328 (1981).
9. D. K. Hsiao, D. S. Kerr and S. E. Madnick, *Computer Security*. Academic Press Inc., ACM Monograph Series (1979).
10. J. Kam and J. D. Ullman, A model of statistical database and their security. *ACM Transactions on Database Systems* **2** (1), 1–10 (1979).
11. R. C. Merkle and M. E. Hellman, Hiding information and signatures in trapdoor knapsacks. *IEEE Transactions on Information Theory* **IT-24** (5), 525–530 (1978).
12. N. Minskey, Intentional resolution of privacy protection in database system. *Communications of the ACM* **19** (3), 148–159 (1976).
13. K. H. Nam, Cryptographic models for computer communications. *Ph.D Dissertation*, University of Southwestern Louisiana (1985).
14. I. Niven and H. Zuckerman, *Introduction to the theory of numbers*. Wiley, New York (1966).
15. R. L. Rivest, A. Shamir and L. Adleman, A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM* **21** (2), 120–126 (1978).

A Short Note on Perfectly Balanced Binary Search Trees

We present a perfect balancing method for a binary search tree. During the updates the algorithm allows the structure to grow gracefully and maintains the optimal shape without degeneration. The algorithm uses swapping as the basic operation. Since the tree produced by the algorithm is optimal it can favourably be compared with that produced by other balancing algorithms. In worst case situation, the algorithm takes $O(n)$ time, n being the total number of nodes in the tree. This is an added significance when it is compared with the static optimal binary search trees.

Received September 1989, revised November 1990

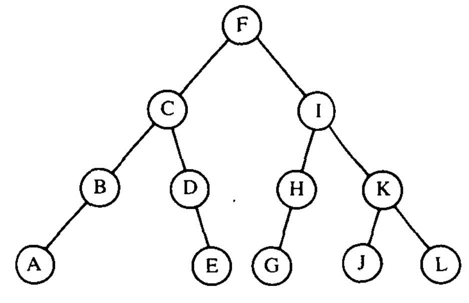
1. Introduction

Studies on the construction and maintenance of optimal binary search trees have received considerable importance in computer science literature. Knuth[9] and Wirth[13] describe algorithms producing static optimal binary trees. The complications involved in restoring the optimal shape of a binary search tree because of an update, lead to the formulations of 'permissive balance'. Such balancing strategies require only a little additional cost for the three reorganization process. Of the two balancing methods, global balancing algorithms[2, 3, 5, 10, 12] require either a traversal through the tree or a transformation of the input tree into some other structure, while the local balancing algorithms[1, 8, 11] keep the tree within the predefined balancing limit. A recent algorithm[7] produces a nearly optimal tree by displacing the data in an 'inorder' fashion until a vacant position is found in the lowest level of the tree. In the next section we present a strategy to create a perfectly balanced (optimal) binary search tree.

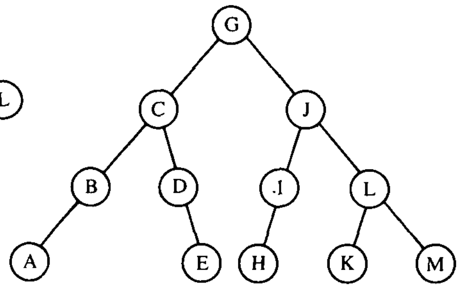
2. The insertion algorithm

Perfect balancing requires that at each node of the tree, the number of nodes in its left subtree nd right subtree differs at most by one. Hence perfectly balanced trees keep the optimal shape and they have the best worst case time in the tree operations. The standard tree data structure with an additional field for storing the number of nodes could be used to implement the tree.

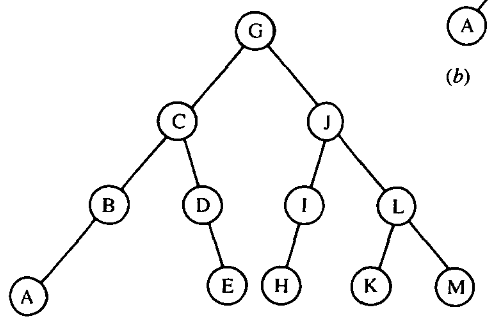
A subtree is defined as a heavy subtree if it has a brother subtree having less number of nodes. A perfectly balanced binary search tree



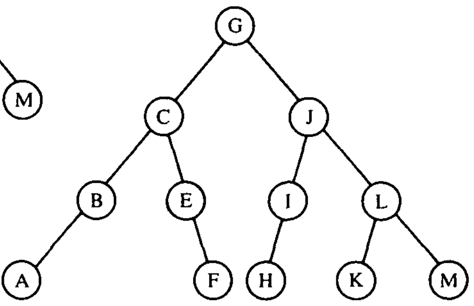
(*a*) Data to insert is say 'M'. Pivot node is 'C'. Number of nodes is 12.



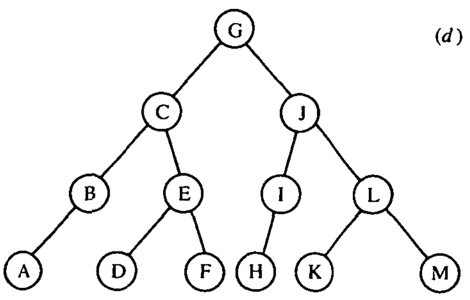
(*b*) After the first call (recursive). Data to insert is 'F'. Stepping down one level.



(c) After the second stage of the call. Data to insert is 'F'. Pivot is nil.



(*d*) After the third stage of call. Data to insert is 'D'.



(*e*) Final tree. A new node is inserted. Number of nodes is now 13.

Figure 1.

```
procedure insert (var t: tree; var newdata: typedata); {This procedure
inserts a new node into an optimal binary search tree. The function
Wt(t) returns t^·weight if t is not null else O. Previous is a variable.}

  begin
    if t = nil then {create a new node}
    begin
      new(t);
        begin
        t^·data := newdata; t^·left, t^·right := nil; t^·weight := 1
    end;
    end
    else
    begin
      previous := t^·data;
      if newdata < t^·data then
      begin
        if wt(t^·left)-wt(t^·right) > 0 then {left imbalance}
          begin
            pivot := t^·right;
            search1(t, newdata, previous);
            insert(t^·right, newdata);
          end
        else {No imbalancing}
        insert(t^·left, newdata);
      end
      else if newdata > t^·data then
      begin
        if wt(t^·left)-wt(t^·right) > = 0 then {no imbalancing}
          insert (t^·right, newdata)
        else {right imbalance}
          begin
            pivot := t^·left;
            search2(t, newdata, previous);
            insert (t^·left, newdata)
          end;
        end
        else write ('ERROR');
        {Update the weights}
        t^·weight := wt(t^·left)+wt(t^·right)+1;
    end
  end;
```

Figure 2. Insertion algorithm.

```
procedure search1 {var t: tree; var newdata: typedata; previous:
typedata);
        begin
        if t < > nil then
          begin
          if newdata < t^·data then
          begin
            previous := t^·data;
            search1(t^·left, newdata, previous);
          end
          else
          if (newdata > t^·data) and (newdata < previous) then
          begin
            previous := t^·data;
            search1 (t^·right, newdata previous)
          end;
        swap1(t, newdata)
      end;
end;

procedure swap1(var t:tree; var newdata: typedata);
var
        tempdata: typedata;
begin
      if (t < > nil) then if (t < > pivot) and (newdata < t^·data) then
      begin
        swap1(t^·left, newdata);
          begin
          if newdata < t^·data then
            begin
              tempdata := t^·data;
              t^·data := newdata;
              newdata := tempdata;
            end
          end;
        swap1(t^·right, newdata);
      end
end;
```

Figure 3. (*a*) The *search*1 algorithm. (*b*) *swap*1 algorithm. *swap*2 is the reverse operation of *swap*1.

Table 1

Algorithm	AVL	BB(α)	Gonnet	Gerasch	Ours
Balance	Height	Weight	*ipl*	Nearly optimal	Optimal
Extra space	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Run time (worst)	$O(\log_2 n)$	$O(\log_2 n)$	$O(n)$	$O(n)$	$O(n)$
Worst Ht:	$1.44H_n$	$2H_n$	$1.44H_n$	H_n	H_n

is out of balance if we insert a node in a heavy subtree. The insertion algorithm ensures that at each node of the tree the number of nodes in its left subtree and right subtree differs at most by one.

The algorithm invokes the balancing actions only when the datum to be inserted has a tendency to force the tree to be out of balance. In this situation the root node of the brother subtree of the heavy subtree is set as the pivot node. Then the *search* procedure is invoked which in turn activates the *swap* procedure for achieving the data displacement in an 'inorder' fashion. This cascading of data displacements terminate when the excess datum is pulled out from the heavy subtree. Then the *insert* procedure is called at the pivot node for inserting the pulled out datum from the heavy subtree and continue the process until a proper place is found to accommodate the newly displaced datum. Now a new node is created to store this latest pulled-out datum. The process is illustrated in Fig. 1.

The code for the insertion algorithm is given in Fig. 2. The counter 'weight' is the major factor to decide whether to invoke the balancing actions or to go down to the next level. The counter in each node of the path from the root to the newly inserted node is updated in each insertion. For convenience we use the function *wt*(*t*) which returns *O* if tree *t* is null otherwise *t*^. weight. The procedure *search*1 is given in Fig. 3 along with *swap*1 procedure. *search*2 differs from *search*1 by its initial checking and the usage of *swap*2. *swap* 2 permits a reverse operation of *swap*1.

3. Performance analysis

Since the algorithm maintains a counter at each node of the tree, the additional space required is $O(n)$. This space requirement can be reduced to $O(n)$ bits by using a bit flag showing the difference of the number of nodes in the left and right subtrees of the nodes. Thus the space requirement is comparable to that of Gerasch's[7] algorithm.

The running time of the algorithm is directly related to the number of calls of the procedure *insert* and the operation of swapping. The worst case time of the algorithm is linear with the number of nodes in the tree. The value of the new datum controls the swapping process. Thus the performance of the algorithm is determined by the position of the new datum

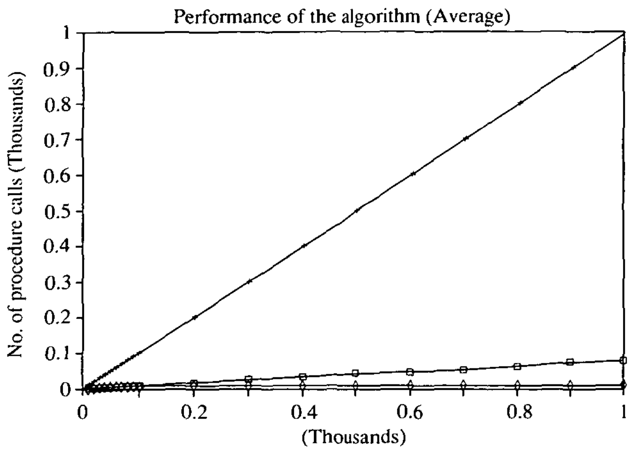


Figure 4. Number of nodes: □ swapping; + number of nodes; ◇ log (number of nodes).

in the tree. During the insertion process, the number of swapping required to insert a node at level $l, (0 \leqslant l \leqslant h)$, is $n-2^{h-(l-1)}-1$, $h = \lfloor \log_2 (n+1) \rfloor$. Gerasch's algorithm[7] produces a nearly optimal tree with swapping as the basic operation. In this algorithm[7] imbalancing occurs at a node P if a subtree of P has more number of complete levels than the other subtree of P. In this case, P becomes the pivot node for rebalancing and keys are displaced in an 'inorder' fashion to make room for the new key. The key that is displaced out of the subtree of P replaces the key in P that in turn becomes the key to insert into the other subtree of P in order to complete the appropriate incomplete level of the subtree. Therefore for trees with nodes just above 2^k, for any k, Gerasch's algorithm runs approximately in logarithmic time. In a similar situation the present algorithm runs in linear time as a consequence of the strict balancing condition for maintaining the optimal shape. In average situations, the algorithm runs in between logarithmic and linear with the number of nodes in the tree. Empirical studies of the present algorithm reveal that the performance of the algorithm in average situation is close to $\log_2(n)$ (Fig. 4). Since the optimal shape is maintained through a series of local rearrangements, in average situations the tree reorganisation is limited to local changes.

4. Conclusion

We have presented a strategy to create an optimal binary search tree. Using swapping as the basic operation, the algorithm dynamically keeps the tree optimal during the updates. In worst case situations, the number of swappings during a single insertion is at most n, where n is the number of nodes in the tree. Hence the algorithm is significant while considering the static optimal binary search trees.

Acknowledgement

The authors are thankful to the referee for his valuable suggestions which improved the earlier version of this paper. One of the authors(APK) is grateful to the University Grants Commission(INDIA) for providing a research fellowship to carry out this research work.

A. P. KORAH *and* M. R. KAIMAL
Department of Computer Science,
University of Kerala,
Trivandrum, India 695 034

References

1. G. M. Adel'son-vel'skii and E. M. Landis, An algorithm for the organization of information. *Dokl. Akad. Naak, USSR* **146** (2), 263–266 (1962).
2. J. L. Bently, Multidimensional binary search tree used for associative searching, *Communications of the ACM* **18** (9), 509-517 (1975).
3. H. Chang and S. S. Iyangar, Efficient algorithms to globally balance a binary search tree. *Communications of the ACM* **27** (7), 695–702 (1984).
4. J. Culberson and J. I. Munro, Explaining the behaviour of binary search trees under prolonged updates: A model and simulations. *The Computer Journal* **32** (1), 68–75 (1989).
5. A. C. Day, Balancing a binary tree. *The Computer Journal* **19** (4), 360–361 (1976).
6. J. L. Eppinger, An empirical study of insertion and deletion in binary trees. *Communications of the ACM* **26** (9) (1983).
7. T. E. Gerasch, An insertion algorithm for a minimal internal path length binary search tree. *Communications of the ACM* **31** (5), 579–585 (1988).
8. G. H. Gonnet, Balancing binary tree by internal path reduction. *Communications of the ACM* **26** (12), 1074–1081 (1983).
9. D. E. Knuth, *The Art of Computer programming, vol. 3 Sorting and Searching*, Addison-Wesley, Reading Massachusetts (1973).
10. W. A. Martin and D. N. Ness, Optimal binary trees grown with a sorting algorithm. *Communications of the ACM* **15** (2), 88–93 (1972).
11. J. Nievergelt and E. N. Reingold, Binary search trees of bounded balance. *SIAM Journal of Computing* **2** (1), 33–43 (1973).
12. Q. F. Stout and B. L. Warren, Tree rebalancing in optimal time and space. *Communications of the ACM* **29** (9), 902–908 (1988).
13. N. Wirth, *Algorithms + Data structures = Programs*. Prentice Hall of India, New Delhi (1988).

On Equivalent Systolic Designs of LU Decomposition and Its Algebraic Representation

Algorithms which are to be mapped onto interconnecting processing elements in order to design a systolic array are conventionally represented by graphs or networks. This paper introduces the concept of algebraic representation and uses a generating function to represent a systolic array.

Received May 1989, revised September 1989

1. Introduction

Conventional design of systolic arrays is based on the mapping of an algorithm onto an interconnection of processing elements. These algorithms are typically described by graphs or networks, where nodes represent processing elements or registers and edges represent interconnections. Although for many purposes these conventional representations are adequate for specifying the VLSI algorithm, the algebraic representation is more suitable for supporting formal manipulation on designs than the graphic or network models.

Space-time recursion equations of parallel algorithm can be naturally represented by an algebraic representation. In this paper we use a generating function to represent a systolic array. It adapts the power series notation to a more algebraic form to aid the specification and design of systolic array. It also provides a global view on the data-interacting activity of a systolic array. Using a generating function as an algebraic representation of a systolic array, properties of linear algebra, such as velocity addition, can be applied to derive different but equivalent designs of a systolic array.

2. Generating function and systolic flow

The proposed generating function[2,9,10] as an algebraic notation for representing a systolic design consists of a collection of data streams. A 'data stream' represents the moving path of a data item, including the relation between the space coordinates of the data item and time. It can be considered as a moving path of a particle in kinetics. A data stream B can be represented by the following generating function:

$$B = \sum_{t \geqslant 0} bX^{i(t)}Y^{j(t)}Z^{k(t)}\tau^t$$

where X, Y, Z are the space axes, τ is the time axis, and b is the data item name of the data stream B. Data item b locates at space $(i(t), j(t), k(t))$ at time t. Let $B(t)$ denote $bX^{i(t)}Y^{j(t)}Z^{k(t)}\tau^t$. Implicitly, data item b of $B(t)$ carries a value which depends on t, but what we are concerned with here is just the position vector $(i(t), j(t), k(t))$ of the moving data item at time t, not how its value is modified in each *PE*, *hence we only use the data item name b*, not its value, to represent the data stream in the following discussion.

If data item b moves with constant velocity $\mathbf{V} = \Delta i\mathbf{x} + \Delta j\mathbf{y} + \Delta k\mathbf{z}$ from the beginning position $(i(0), j(0), k(0))$, where $\mathbf{x}, \mathbf{y}, \mathbf{z}$ are unit