1-2 brother trees or AVL trees revisited*

T. Ottmann† and D. Wood‡

1-2 brother trees are binary search trees which have similarities to both height balanced search trees and 2-3 trees. Firstly, $0(\log_2 n)$ insertion and deletion algorithms are demonstrated and their similarities with those for brother trees are noted. Secondly, it is proved that the space utilisation of (random) 1-2 brother trees is much better than that for (random) 2-3 trees. Thirdly, the close relationship between 1-2 brother trees and height balanced trees is demonstrated, and as this also holds for their right-sided counterparts it leads to $0(\log_2 n)$ insertion and deletion algorithms for right-sided height balanced trees. Finally, this demonstrates that the insertion and deletion algorithms for right-sided height balanced trees were already available, but hidden, in the corresponding algorithms for right brother trees.

(Received February 1978, revised January 1979)

1. Introduction

Recently there has been a flurry of activity in the area of binary search trees, for example Hirschberg (1976), Ottmann and Six (1976), Ottmann, Six and Wood (1978a; 1978b; 1979), Ottmann and Wood (1978), Räihä and Zweben (1979) and Zweben and McDonald (1978). Two trends are apparent. Firstly, as a result of Hirschberg (1976) and Knuth (1973), the pursuit of $0(\log_2 n)$ insertion and deletion algorithms for right-sided height balanced trees (RSHB trees) was initiated, that is, the right-sided variant of the AVL trees of Adelson-Velskij and Landis (1962). This has been successful very recently (see Räihä and Zweben (1979) for insertion and Ottmann and Wood (1978) and Zweben and McDonald (1978) for deletion). Secondly, brother trees (Ottmann and Six, 1976) and their right-sided variants, right brother trees (Ottmann, Six and Wood, 1978a), have been investigated, again demonstrating $0(\log_2 n)$ insertion and deletion algorithms in both cases.

Moreover in Ottmann, Six and Wood (1978b; 1979) the close correspondence between (right) brother trees and (right-sided) height balanced trees was demonstrated. However this correspondence assumes that the keys are stored at the leaves; while internal nodes contain queries as auxiliary information, the so called *leaf* search trees (brother and right brother search trees are by definition assumed to be leaf search trees). Further the correspondence enables $0(\log_2 n)$ insertion and deletion algorithms for (RS)HB leaf search trees to be derived directly from those available for (right) brother search trees.

Now the recent insertion algorithm (Räihä and Zweben, 1979) for the standard RSHB search trees (the keys stored at the internal nodes) is based on the principle contained in one for right brother search trees (Ottmann, Six and Wood, 1978a).

In the present paper we demonstrate that these distinct insertion algorithms are 'identical' in a strong sense. We first define 1-2 (right) brother search trees, which are brother trees in which the keys are stored at the internal nodes. The keys are stored analogously to the method used for 2-3 trees (Aho, Hopcroft and Ullman, 1974), namely, in 2-3 trees a node with two sons has one key and a node with three sons has two keys. For 1-2 brother trees a node with one son has no key and a node with two sons has one key. In Section 2, it is shown directly that $0(\log_2 n)$ insertion and deletion algorithms exist for 1-2 brother trees. However, it is then observed that these are the same algorithms as those for brother trees with appropriate modifications due to keys being stored at internal nodes. This leads immediately, in Section 4, to $0(\log_2 n)$ insertion and deletion algorithms for 1-2 right brother trees. Further, since the correspondence between 1-2 (right) brother trees and (RS)HB trees, respectively, is straightforward immediately the solution to Knuth's original problem (1973, p. 471) is immediate. In other words Ottmann, Six and Wood (1978a) contains the solution to the insertion and deletion problem for RSHB trees once the correct spectacles are worn!

We are convinced that the concepts of brother trees and 1-2 brother trees, although from one point of view a reinvention of AVL trees in a slightly different guise, have importance in their own right. Not only have they given insight into the solution of Knuth's problem for RSHB trees, but we maintain that conceptually they are simpler. Because of this, in Section 3, the space complexity of 1-2 brother trees is analysed and compared favourably with that for 2-3 trees by Yao (1978). This analysis is however a non-trivial extension of that of Yao, as Brown (1978) has also independently observed when treating AVL trees.

2. 1-2 brother trees

A tree in which each node may have one or two sons is called a *brother tree* (see Ottmann and Six, 1976) if 1 and 2 hold:

1. All leaves are of the same depth.
2. Each node with only one son has a brother with two sons.

In order to represent sets of keys we extend the method of storing keys which is usual for 2-3 trees, viz keys are stored at internal nodes only. An internal node with *two* sons has *one* key; an internal node with only *one* son has *no* key. If the keys stored at internal nodes are ordered according to 3 below, we call the resulting tree a *1-2 brother tree*.

3. For each internal node p with two sons, the keys in the left subtree of p are less than the key of p which in turn, is less than the keys in the right subtree of p.

The tree of **Fig. 1** shows an example of a 1-2 brother tree representing the set $\{2, 3, 5, 7, 8, 10, 13\}$. Let λp, ρp denote the left, resp. right son of p, if p has two sons, let σp denote the only son of p if p has only one son, and let ϕp denote the father of node p.

To search for a key x in a 1-2 brother tree with root p perform *search* (p, x).

*Work carried out partially under a Natural Sciences and Engineering Research Council of Canada Grant No. A-7700 and partially under a grant from the Deutsche Forschungsgemeinschaft (DFG).

†Institut für Angewandte Informatik und Formale Beschreibungsverfahren, Universität Karlsruhe, Postfach 6380, 7500 Karlsruhe, West Germany.
‡Unit for Computer Science, McMaster University, Hamilton, Ontario L85 4K1, Canada.

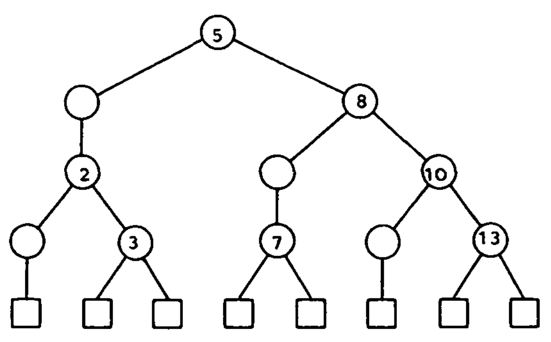


Fig. 1

Search (p, x)
Case 1 [p has two sons]

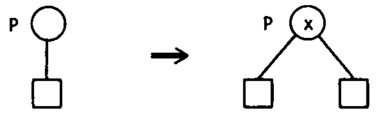
1.1 [key $(p) > x$]
    perform *search* $(\lambda p, x)$
1.2 [key $(p) = x$]
    then p is the desired node.
1.3 [key $(p) < x$]
    perform *search* $(\rho p, x)$.

Case 2 [p has only one son]
    perform *search* $(\sigma p, x)$

Case 3 [p is a leaf]
    There is no node with key x in the tree.

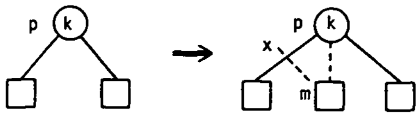
In order to *insert* a new key x in a 1-2 brother tree with at least one key and root r, firstly, perform *search* (r, x). Since the search will be unsuccessful we end up in a leaf; let p be the father of this leaf. We distinguish two cases.

Case 1: p has only one son
Then no key is stored in p. Hence, we can give p a second son, store x in p, and FINISH.



Case 2: p has two sons
Then p already has a key k. We create a new leaf m in between λp and ρp and call *up* (p, m, x).

Remark
We may assume without loss of generality that $x < k =$ key (p), since k and x can be interchanged, if necessary, before calling *up*.



Call $\underline{\text{up}}$ (p,m,x)

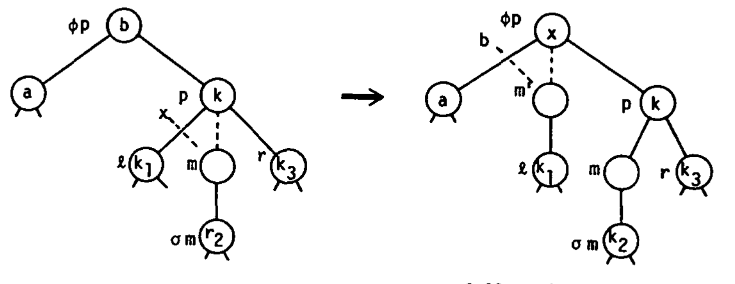
By T_q we denote the (sub)tree with root q and by *keys* (T_q) the set of keys stored in T_q. (Keys $(T_q) = \emptyset$, if q is a leaf or q is a semileaf with only one son.) We extend the relation '<' in the obvious way to a relation between sets of keys.

Then, the invariant condition maintained by the procedure *up* reads as follows:
Whenever *up* (p, m, x) is called then the following holds:

(a) p has two sons $l = \lambda p$ and $r = \rho p$, which are roots of 1-2 brother trees
(b) m is either a leaf or has only one son σm which is the root of a 1-2 brother tree, and height $(m) =$ height $(\lambda p) =$ height (ρp)
(c) either $x <$ key (p) if m is a leaf, or keys $(T_l) < x <$ keys $(T_{\sigma m}) <$ key $(p) <$ keys (T_r) otherwise.

procedure *up* (p, m, x);
    node p, node m, integer x;

Case 1 [p has a left brother with two sons]



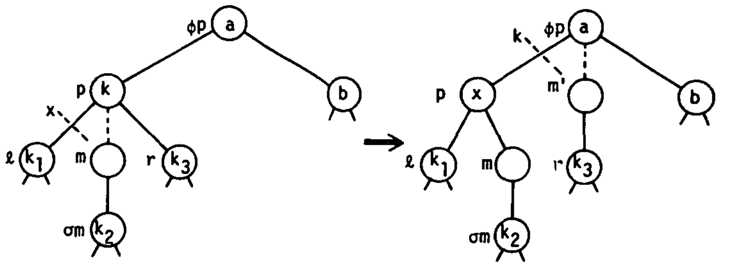
Call $\underline{\text{up}}$ (ϕp,m',b)

(Clearly, if l, m, r are leaves, then σm does not exist. Hence, in this case σm and the keys k_1, k_2, k_3 have to be removed from the above figure. Similar assumptions are also necessary in the following figures for treating the 'leaf case'.)

The restructuring to be performed in this case can be verbally described as follows: Create a new node m', make the left son l of p the only son of m', make m the new left son of p. Let $b =$ key (ϕp), key $(\phi p) = x$ and call *up* $(\phi p, m', b)$.

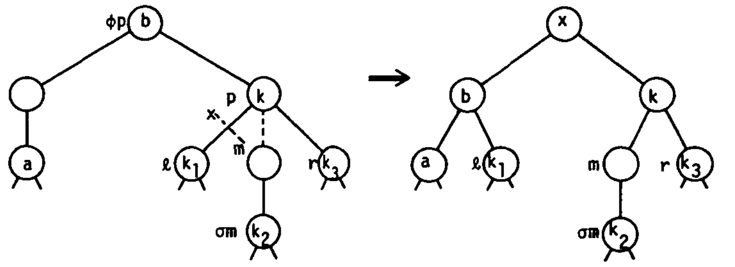
Observe that the invariant condition is maintained. In the following we only show the figure which describes the local restructuring to be performed.

Case 2 [p has a right brother with two sons]



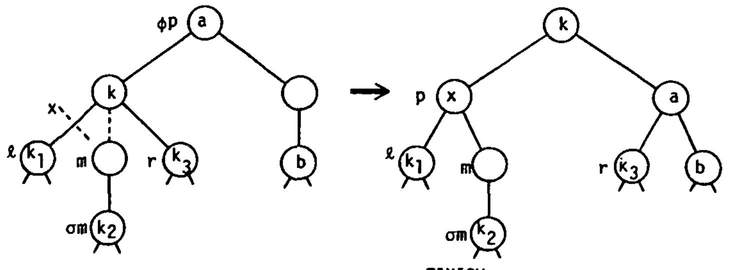
Call $\underline{\text{up}}$ (ϕp,m',k)

Case 3 [p has a left brother with only one son]



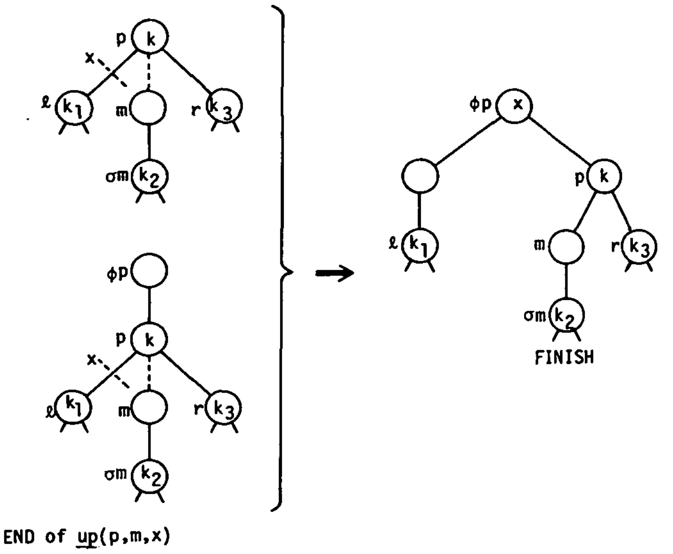
FINISH

Case 4 [p has a right brother with only one son]



FINISH

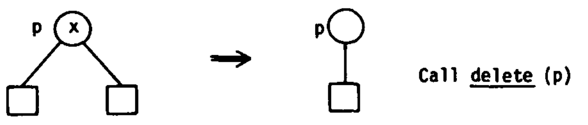
Case 5 [p has no brother]
(Then p is either the root or the only son of its father.)



In Ottmann and Six (1976) a similar upwards restructuring procedure for brother trees is given. These two procedures perform exactly the same restructuring, when key and query movements are ignored.

The same holds for the restructuring which will be necessary after the *deletion* of a key from a 1-2 brother tree: In order to *delete* a node with key x from a 1-2 brother tree we first search for the node p with key x. We may assume that p has two sons since each key in the tree appears at a node with two sons. We distinguish two cases:

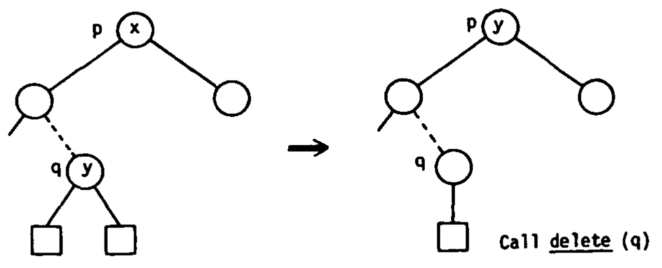
Case 1: The two sons of p are leaves
Then remove one of the leaves, remove the key x from p and call *delete* (p).



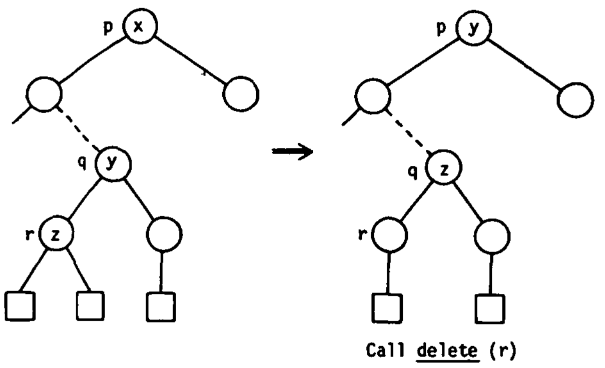
Case 2: The two sons of p are non-leaves
Now either λp is binary or ρp is binary, since we are dealing with a 1-2 brother tree. Consider the case λp is binary in the following, the case ρp is binary is similar.

Determine the rightmost binary node q in the left subtree of p. Let y be the key of q. One of the following two subcases must occur:

Case 2.1: The sons of q are leaves
Let y be the key of q. Replace the key x of p by y, remove the key of q and one of its leaves and call *delete* (q).



Case 2.2: q has a binary left brother r and a unary right brother



Replace the key x of p by the key y of q, make the key z of r the key of q, remove the key of r and one of its leaves and call *delete* (r).

Whenever *delete* (p) is called, then

(*a*) p has only one son which is either a leaf or the root of a 1-2 brother tree

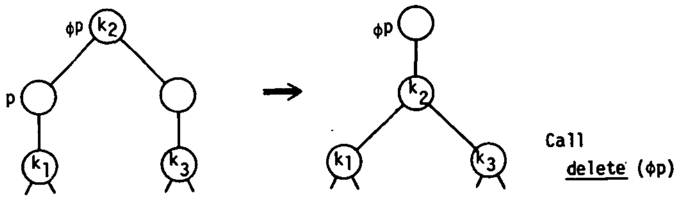
(*b*) p has lost its key

(*c*) For all nodes q in the tree apart from p and βp (if βp exists) if q is unary, then q has a binary brother.

procedure *delete* (p); node p;

Case 1 [p has a brother with two sons]
FINISH

Case 2 [p has a brother with only one son]

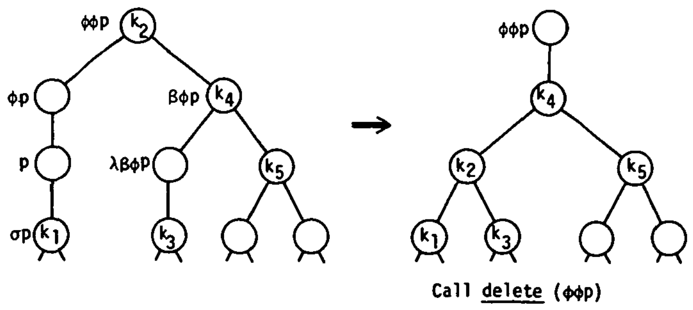


(The case that p is the left son of its father is treated similarly.)

Case 3 [p has no brother]
Case 3.1 [p is the root]
Then remove p, make the only son of p the new root, and FINISH.

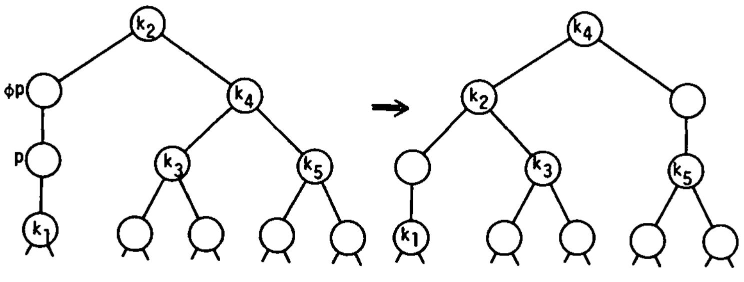
Case 3.2 [p is the only son of its father ϕp].
By the invariant condition, ϕp has a brother $\beta\phi p$ with two sons. We distinguish whether or not $\beta\phi p$ has three or four grandsons.

Case 3.2.1 [$\lambda\beta\phi p$ or $\rho\beta\phi p$ has only 1 son]
Assume that ϕp is the left son of its father and that $\lambda\beta\phi p$ has only one son.



We leave it to the reader to treat the symmetric cases which can be subsumed under Case 3.2.1.

Case 3.2.2 [*λβφp* and *ρβφp both have two sons*]
Assume that *φp* is the left son of its father



FINISH

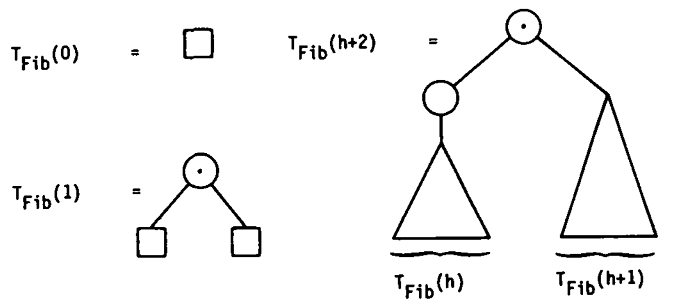
We leave it to the reader to treat the case where *φp* is the right son of its father.

End of *delete* (*p*).

Since an N-key brother tree is of height $0(\log_2(N + 1))$ by the results in (Ottmann, Six and Wood, 1979) and the opening remarks in Section 3, we have the following theorem.

Theorem 1
The algorithms to search for a key, to insert a new key, and to delete a key from an N-key 1-2 brother tree can be carried out in time at most $0(\log(N + 1))$.

3. Storage utilisation
The number of keys stored in a 1-2 brother tree T is equal to the number of internal nodes with two sons in T. This in turn equals the number of leaves of T minus 1. The Fibonacci tree $T_{\mathrm{Fib}}(h)$ of height h is defined by



Here, the dots represent stored keys. The Fibonacci trees are 1-2 brother trees of 'minimal denseness', since one easily shows (by induction):

1. A 1-2 brother tree with height h has at least $\mathrm{Fib}(h + 2)$ leaves. (Here $\mathrm{Fib}(n)$ denotes the nth Fibonacci number, i.e. $\mathrm{Fib}(1) = \mathrm{Fib}(2) = 1$, $\mathrm{Fib}(n + 2) = \mathrm{Fib}(n) + \mathrm{Fib}(n + 1)$, for $n > 0$).
2. $T_{\mathrm{Fib}}(h)$ has $\mathrm{Fib}(h + 3) - 2$ internal nodes.

For a Fibonacci tree of height h we have:

$$\text{storage utilisation} = \frac{\text{number of internal nodes}}{\text{number of stored keys}} = \frac{\mathrm{Fib}(h + 3) - 2}{\mathrm{Fib}(h + 2) - 1} \to \frac{1 + \sqrt{5}}{2}, \text{ as } h \to \infty = 1{\cdot}618$$

Hence, in order to store N keys in a 1-2 brother tree we need approximately 1·618 N internal nodes (= storage cells) for large N in the *worst case*.

We will now estimate the *average* storage requirement for random 1-2 brother trees by using a method which was developed for 2-3 trees by Yao (1978). The concept of a random insertion is central to the analysis.

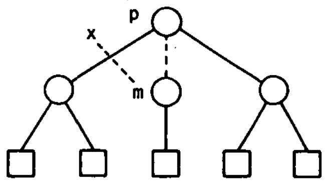
Definition of random insertion
Consider a 1-2 brother tree T containing $j - 1$ keys. These $j - 1$ keys divide all possible key values into j intervals. (These intervals are represented by the j leaves of T.) The insertion of the jth key k_j is said to be *random* if k_j has equal probability of being in any one of the j intervals, that is, of hitting any one of the j leaves.

We will estimate the average storage requirement for trees which are built by N successive random insertions. First of all, we recall Yao's notation and adapt it for 1-2 brother trees.

Let T be a 1-2 brother tree.

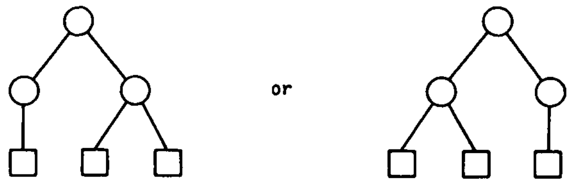
$n(T)$	denotes the number of internal nodes in T.
$f_N(T)$	denotes the probability of obtaining T after N random insertions (when starting with the empty tree).
$\bar{n}(N)$	denotes the average number of internal nodes in 1-2 brother trees which result after N random insertions (beginning with the empty tree).

To derive a bound for $\bar{n}(N)$ we estimate the number of internal nodes at the lowest two levels of random 1-2 brother trees. When analysing the average storage utilisation of random 2-3 trees Yao (1978) utilised the fact that one can completely determine the effect of a random insertion with respect to all subtrees of a given height h. Yao called this an analysis of order h. The corresponding observation does not hold in the case of 1-2 brother trees. If, for example, a key is inserted into the subtree of height 2 with 4 leaves, the upwards restructuring procedure *up* will be recursively called for the root p of the subtree (and some m, x):

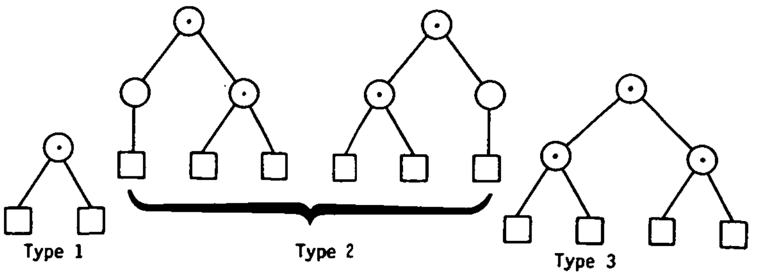


up (p, m, x)

As far as the resulting subtrees of height 2 are concerned we only know that one subtree of the form



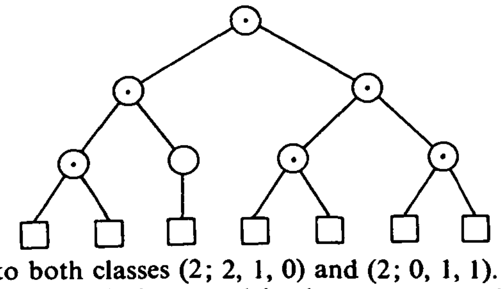
will be generated. But we do not know whether the root of the remaining subtree of height 1 with two leaves will be the only son of its father or will obtain a binary brother. This depends on whether or not p has a unary brother. Thus we cannot make a pure second order (in general, a pure h-order) analysis by considering only subtrees of height 2 (in general, of height h).* As a consequence of this fact we also drop Yao's requirement that a tree is represented uniquely by the number of small subtrees that occur. Consider the following three types of subtrees of heights 1 and 2 in a 1-2 brother tree. (Keys are represented by dots.)



*Instead of this we will consider a class of subtrees with height at most 2 (in general, at most h) and call this a second order (in general, an h-order) analysis.

The root of a type 1 subtree is assumed to be either the only son of its father or to have a binary brother. An arbitrary 1-2 brother tree T is said to be of class $(2; x_1, x_2, x_3)$ if T has x_i subtrees of type i, $i = 1, 2, 3$, (when counting each subtree only once). Observe that a tree T may belong to different classes. For example, the tree T

T =



belongs to both classes (2; 2, 1, 0) and (2; 0, 1, 1).

Since we are only interested in the average number of keys stored at the lowest two levels of internal nodes, the ambiguity of the classification of the trees will only affect the bounds of an estimation rather than making such an estimation impossible. The key observation is that for each tree of type i the result of a random insertion is uniquely determined. This fortuitous circumstance implies that Yao's analytic method can be adapted for 1-2 brother trees. Brown (1978) has recently made a similar observation for AVL trees in a different context.

Let T be a 1-2 brother tree of class $(2; x_1, x_2, x_3)$ with N keys.

Lemma 1

$$2x_1 + 3x_2 + 4x_3 = N + 1 .$$

Proof

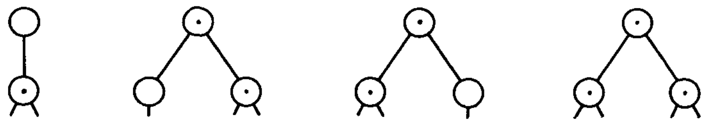
Both sides are equal to the number of leaves of T.

Lemma 2

$$N \leqslant n(T) \leqslant \frac{5}{3} N .$$

Proof

Consider the internal nodes which occur on two arbitrary adjacent levels in T. Whether or not a key is stored at an arbitrary node p, depends upon its neighbours. The following cases are possible:



(1) (2) (3)

The brother tree condition ensures that for each part of type (1) there must occur at least one part of type (2) or (3) on the same level. Hence, we obtain the following possible values for the ratio: (number of internal nodes in the considered two levels/number of keys in these levels):

Type	*ratio*
(2)	3/2
(3)	3/3
(1) and (2)	5/3 = 1·66 . . .
(1) and (3)	5/4

Hence, 5/3 is an upper bound for the ratio when considering two adjacent levels. Clearly, the same bound holds for the whole tree.

Remark

We could have improved the upper bound to $\frac{1 + \sqrt{5}}{2}$ for large N. But this would have only a minor influence on further estimates.

When considering four instead of two adjacent levels we obtain the following inequality with a similar elementary proof:

Lemma 2'

$$N \leqslant n(T) \leqslant \frac{18}{11} N = 1{\cdot}6363 \ldots N$$

Lemma 3

$$2x_1 + 4x_2 + 4x_3 - 1 \leqslant n(T)$$
$$n(T) \leqslant \frac{5}{3}(x_1 + x_2 + x_3 - 1) + 2x_1 + 3(x_2 + x_3)$$

Proof

Let us denote by N' (respectively $n'(T)$) the number of keys (respectively, the number of internal nodes) *above* the lowest two levels of internal nodes in the N-key 1-2 brother tree T of class $(2; x_1, x_2, x_3)$.

Lemma 2 yields

1. $N' \leqslant n'(T) \leqslant \frac{5}{3} N'$.

Clearly,

$n'(T) = n(T)$—number of internal nodes *at* the lowest two levels of internal nodes.

Each subtree of Type 2 or of Type 3 has three internal nodes at the lowest two levels of internal nodes. Two Type 1 subtrees of height 1 contribute at least three and at most four internal nodes at the lowest two levels of internal nodes. Thus, we can estimate $n'(T)$ as follows.

2. $n(T) - 2x_1 - 3(x_2 + x_3) \leqslant n'(T) \leqslant n(T) - \frac{3}{2}x_1 - 3(x_2 + x_3)$.

Furthermore $N' = N$ − number of keys *at* the lowest two levels of internal nodes. Each subtree of Type 2 has two and each subtree of Type 3 has three keys stored. For each two occurring subtrees of Type 1 there must be stored at least two and at most three keys at the lowest two levels of internal nodes. This yields

3. $N - \frac{3}{2}x_1 - 2x_2 - 3x_3 \leqslant N' \leqslant N - x_1 - 2x_2 - 3x_3$

1, 2, 3 and Lemma 1 yield the hypothesis.

Remark

When using Lemma 2′ instead of Lemma 2 we obtain:

Lemma 3'

$$2x_1 + 4x_2 + 4x_3 - 1 \leqslant n(T)$$
$$n(T) \leqslant \frac{18}{11}(x_1 + x_2 + x_3 - 1) + 2x_1 + 3(x_2 + x_3)$$

Notation (See also Yao (1978))

$F(x_1, x_2, x_3)$ = the set of trees of class $(2; x_1, x_2, x_3)$

$P_N(x_1, x_2, x_3)$ = the probability that a tree of class $(2; x_1, x_2, x_3)$ results after N random insertions.

$$= \sum_{T \in F(x_1, x_2, x_3)} f_N(T).$$

$A_i(N)$ = the average value of x_i for random N-key 1-2 brother trees.

$$= \sum_{x_1, x_2, x_3} x_i P_N(x_1, x_2, x_3).$$

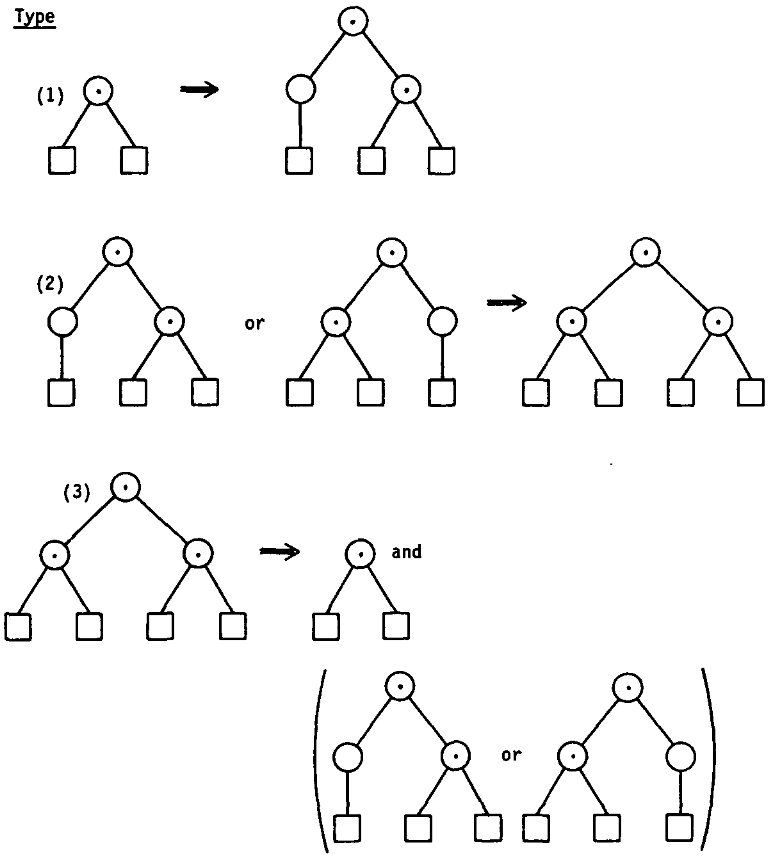
Lemma 4

$$2A_1(N) + 4A_2(N) + 4A_3(N) - 1 \leqslant \bar{n}(N)$$

$$\bar{n}(N) \leqslant \frac{5}{3}(A_1(N) + A_2(N) + A_3(N) - 1) + 2A_1(N) + 3(A_2(N) + A_3(N)).$$

Proof
By Lemma 3 and the definitions.

Consider any $(N - 1)$ key 1-2 brother tree T of class $(2; x_1, x_2, x_3)$. **Table 1** shows the possible transitions under a random insertion (of the Nth key). A tree of class $(2; x_1, x_2, x_3)$ becomes a tree of class $(2; x'_1, x'_2, x'_3)$ with a probability of occurrence shown in the last column. The probabilities follow from the definition of a random insertion and the fact that Type i subtrees have $i + 1$ leaves, $i = 1, 2, 3$. The values of x'_1, x'_2 and x'_3 follow from the insertion procedure of Section 2, which transforms the type i subtrees of heights 1 and 2 as follows (observe that the root of a type 1 subtree cannot have a unary brother, since it would then be a type 2 subtree).



From Table 1 we obtain the following recurrence relations for $A_i(N)$, $i = 1, 2, 3$:

$$A_1(N) = \sum_{x_1,x_2,x_3} P_{N-1}(x_1, x_2, x_3)\left[\frac{2x_1}{N}(x_1 - 1) + \frac{4x_3}{N}(x_1 + 1) + \left(1 - \frac{2x_1}{N} - \frac{4x_3}{N}\right)x_1\right]$$

$$= \sum_{x_1,x_2,x_3} P_{N-1}(x_1, x_2, x_3)\left[\left(1 - \frac{2}{N}\right)x_1 + \frac{4}{N}x_3\right]$$

$$= \left(1 - \frac{2}{N}\right)A_1(N - 1) + \frac{4}{N}A_3(N - 1).$$

$$A_2(N) = \sum_{x_1,x_2,x_3} P_{N-1}(x_1, x_2, x_3)\left[\left(1 - \frac{6}{N}\right)x_2 + 1\right]$$

$$= \left(1 - \frac{6}{N}\right)A_2(N - 1) + 1.$$

Table 1

x'_1	x'_2	x'_3	*Probability*
$x_1 - 1$	$x_2 + 1$	x_3	$2x_1/N$
x_1	$x_2 - 1$	$x_3 + 1$	$3x_2/N$
$x_1 + 1$	$x_2 + 1$	$x_3 - 1$	$4x_3/N$

Table 2

N	$A_1(N)$	$A_2(N)$	$A_3(N)$
1	1	0	0
2	0	1	0
3	0	0	1
4	1	1	0
5	3/5	4/5	3/5
6	4/5	1	3/5

$$A_3(N) = \sum_{x_1,x_2,x_3} P_{N-1}(x_1, x_2, x_3)\left[\frac{3}{N}x_2 + \left(1 - \frac{4}{N}\right)x_3\right]$$

$$= \frac{3}{N}A_2(N - 1) + \left(1 - \frac{4}{N}\right)A_3(N - 1).$$

For $N \leqslant 6$ we explicitly calculate the initial values of $A_i(N)$, $i = 1, 2, 3$. These values are contained in **Table 2.**

With these initial values and the above recurrence relations one easily shows:

$$A_1(N) = \frac{4}{7 \times 5}(N + 1), \quad \text{for } N \geqslant 6$$

$$A_2(N) = \frac{1}{7}(N + 1), \quad \text{for } N \geqslant 6$$

$$A_3(N) = \frac{3}{7 \times 5}(N + 1), \quad \text{for } N \geqslant 6$$

Thus Lemma 4 yields for $N \geqslant 6$:

$$\frac{2 \times 4}{7 \times 5}(N + 1) + \frac{4}{7}(N + 1) + \frac{4 \times 3}{7 \times 5}(N + 1) - 1 \leqslant \bar{n}(N)$$

and

$$\bar{n}(N) \leqslant \frac{5}{3}\left(\frac{4}{7 \times 5}(N + 1) + \frac{1}{7}(N + 1) + \frac{3}{7 \times 5}(N + 1) - 1\right) + \frac{2 \times 4}{7 \times 5}(N + 1) + 3\left(\frac{1}{7}(N + 1) + \frac{3}{7 \times 5}(N + 1)\right)$$

This leads to the following theorem.

Theorem 2

$$\frac{40}{35}N + \frac{5}{35} \leqslant \bar{n}(N) \leqslant \frac{52}{35}N - \frac{19}{105}, \quad \text{for } N \geqslant 6$$

Corollary

$$1{\cdot}143\,N < \bar{n}(N) < 1{\cdot}486\,N, \qquad \text{for } N \geqslant 6$$

(Using Lemma 3′ instead of Lemma 3 yields a merely minor improvement of the upper bound for $\bar{n}(N)$, namely:

$$\bar{n}(N) \leqslant \frac{568}{385}N - \frac{62}{385} = 1{\cdot}476\,N - 0{\cdot}161, \quad \text{for } N \geqslant 6.)$$

Thus we have estimated the average storage utilisation with an inaccuracy of about 0·343 N.

To compare our result with that of Yao's (1978) for 2-3 trees take into account the fact that each internal node in a 2-3 tree has storage capacity 2 while in 1-2 brother trees nodes have storage capacity 1. Yao proved that the average storage capacity of random N-key 2-3 trees lies between 1·40 N and 1·58 N.

Hence, we see that random 1-2 brother trees have a better storage utilisation than random 2-3 trees.

We can try to improve the bounds of Theorem 2 by a third order analysis. It is easy to find a set of eight different kinds of subtrees of height one, two, and three (which, in particular contains the trees of type 1, 2, 3 above) such that each 1-2 brother tree can be characterised (non-uniquely) by the number of occurring subtrees of these eight kinds. Further, recurrence formulas for the number of occurring subtrees of each respective kind can be derived. (Solving these recurrence formulas involves the manipulation of an 8×8 matrix). But it turns out that the number of subtrees of height 3 tends to 0 as the number N of random insertions goes to infinity. Thus, we ultimately obtain exactly the same information about the number of occurring 'small' subtrees as given by our second order analysis above. Hence, the question whether or not the bounds of Theorem 2 can be improved (by a higher order analysis) remains open.

In closing this section we mention that the result of Theorem 2 has been experimentally verified by Neuser (1977). He has obtained as the average number of internal nodes of an N-leaf (i.e. of an $(N-1)$-key) brother tree: $1{\cdot}308\ N \pm 0{\cdot}5837$ when using iterated random insertions. (If the trees obtained by iterated random insertions are completely destroyed by performing random deletions, the average number of internal nodes of an N-leaf intermediate tree is $1{\cdot}3236\ N \pm 0{\cdot}5883$). For each $N \leqslant 1{,}000$, 100 sample trees were built up by $N-1$ random insertions. Furthermore, each tree obtained was also completely destroyed by performing the respective number of random deletions.

4. Connection with height balanced trees

A binary search tree is called *height balanced* (or an *AVL-tree*), if for each node p the heights of the left subtree $T_{\lambda p}$ and the right subtree $T_{\rho p}$ of p differ by at most one, the balance factor of p. When using only internal nodes to store keys and using the leaves to represent key intervals for unsuccessful searches (see for example Knuth (1973)) the correspondence between height balanced trees and 1-2 brother trees is immediate: Let a 1-2 brother tree be given. *Contract* each node with only one son by replacing that node by its only son. The tree obtained is height balanced.

Conversely, let a height balanced tree be given. *Expand* each left (resp. right) son of a node with balance factor $+1$ (resp. -1) by inserting a new node which is the only son of its father and which has the same key as the expanded node, removing the key from the expanded node. The resulting tree will be a 1-2 brother tree. **Fig. 2** shows an example of a 1-2 brother tree and its corresponding height balanced tree.

The connection between height balanced *leaf* search trees and brother trees when considered as leaf search trees has already been established by Ottmann, Six and Wood (1979). This correspondence and also its counterpart for the respective subclasses of right-sided trees carries over to the (normal) AVL trees and 1-2 brother trees.

Before stating this fact, we recall the following definitions: An AVL tree is called *right-sided height balanced* if each node p has balance factor 0 or $+1$.

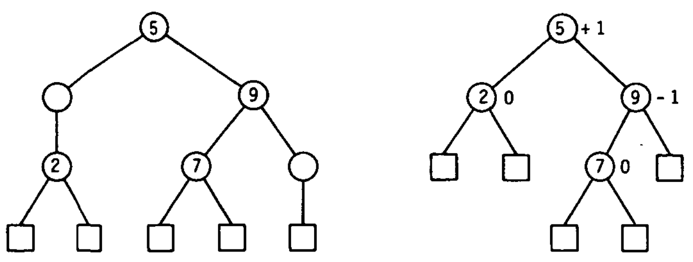


Fig. 2

A 1-2 brother tree is called a *1-2 right brother tree* if each node p with only one son has a *right* brother with two sons.

Theorem 3

1. A binary search tree is a (right-sided) height balanced tree if it results by contraction from a 1-2 (right) brother tree.
2. A binary search tree is a 1-2 (right) brother tree if it results by expansion from a (right-sided) height balanced tree.

Räihä and Zweben (1979) found a $O(\log N)$ insertion algorithm for right-sided height balanced search trees. Zweben and McDonald (1978) and Ottmann and Wood (1978) independently designed a $O(\log N)$ deletion procedure for these same trees.

Although Räihä and Zweben mentioned that their algorithm is based on the insertion algorithm for right brother trees developed in Ottmann, Six and Wood (1978a), the connection between these two algorithms was not made explicit. But it is now clear how insertion and deletion algorithms for (right-sided) height balanced trees can be obtained from the respective source algorithms for 1-2 (right) brother trees. Use *local* expansion and *local* contraction to obtain target algorithms which have run times of the same order as the source algorithms. An explanation of local expansion and contraction can be found in Ottmann, Six and Wood (1979), for the case of *leaf* search trees.

Although insertion and deletion algorithms for 1-2 right brother trees are not explicitly given here it is now straightforward to obtain them from those for right brother trees in Ottmann, Six and Wood (1978a).

We have already seen in Section 1 that the following is sufficient to carry over the algorithms which were originally developed for brother trees to 1-2 brother trees in which the internal nodes are used to store the keys. All one step transformations to be performed locally (and globally) maintain the number of internal nodes with two sons. Thus, a new assignment of key values can be carried out simultaneously with any local restructuring of the tree. This can always be restricted to the neighbourhood of the respective actual procedure parameter node and performed in such a way that the whole tree remains a search tree. When disregarding the keys the restructuring is the same as for brother trees.

The target algorithm, once obtained, can also be formulated without explicit reference to the source algorithm. Furthermore, it is often possible to clean up the target algorithm by cutting short the case analysis. However, we feel that the algorithms developed for (right) brother trees play a prominent role in the understanding and development of the involved algorithms for (one-sided) height balanced trees.

References

Adelson-Velskii, G. M. and Landis, Y. M. (1962). An Algorithm for the Organization of Information. *Doklady Akademiiy Nauk SSSR 146*, pp. 263-266. English translation in *Soviet Math. 3*, pp. 1259-1263.

Aho, A. V., Hopcroft, J. E. and Ullman, J. D. (1974). *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass.

Brown, M. (1978). A Partial Analysis of Random Height-Balanced Trees. *SIAM Journal of Computing*.

Hirschberg, D. S. (1976). An Insertion Technique for One-sided Height-Balanced Trees, *ACM*, Vol. 19 No. 8, pp. 471-473.

Knuth, D. E. (1973). *The Art of Computer Programming*, Vol. 3: Sorting and Searching, Addison-Wesley, Reading, Mass.

NEUSER, W. (1977). ExperimentelleUntersuchung ausgeglichener Binärbäumen, Master's Thesis, Karlsruhe, 1977.
OTTMANN, T. and SIX, H.-W. (1976). Eine neue Klasse von ausgeglichenen Binärbäumen, *Angewandte Informatik*, Vol. 18 No. 9, pp. 395-400.
OTTMANN, T., SIX, H.-W. and WOOD, D. (1978a). Right Brother Trees, *ACM*, Vol. 21, pp. 769-776.
OTTMANN, T., SIX, H.-W. and WOOD, D. (1978b). A Survey of New Results in Balanced Search Trees, *Datenstrukturen, Graphen, Algorithmen* (ed., J. Muhlbacher), Carl Hanser Verlag, Munchen, pp. 107-124.
OTTMANN, T., SIX, H.-W. and WOOD, D. (1979). On the Correspondence between AVL Trees and Brother Trees, *Computing*, to appear.
OTTMANN, T. and WOOD, D. (1978). Deletion in One-sided Height-Balanced Search Trees, *International Journal of Computer Mathematics*, Vol. 6, pp. 265-271.
RÄIHÄ, K. J. and ZWEBEN, S. H. (1979). An Optimal Insertion Algorithm for One-sided Height-balanced Binary Search Trees, *CACM*, Vol. 22, pp. 508-512.
YAO, A. C-C. (1978). On Random 2-3 Trees, *Acta Informatica*, Vol. 9, pp. 159-170.
ZWEBEN, S. H. and MCDONALD, M. A. (1978). An Optimal Method for Deletion in One-sided Height-balanced trees, *CACM*, Vol. 21, pp. 441-445.

Book reviews

Structured Programming, by R. C. Linger, H. D. Mills and B. I. Witt, 1979; 402 pages. (*Addison-Wesley*, £14·25)

The term 'structured programming' has long ago lost most of its meaning; for many potential readers it will mean nothing more than acceptance of a simple constraint on patterns of control flow (DO-WHILE and IF-THEN-ELSE), an anathema on GO TO and an adherence to the broad church of top down design. For Linger, Mills and Witt, 'structured programming' means much more than this: it means above all a detailed and careful consideration of control flow, and an insistence that program development should be 'rigorous' (i.e. provably correct) rather than 'heuristic' (i.e. proceeding by trial and error).

After a chapter briefly introducing some basic concepts such as sets, relations, digraphs, state machines, regular expressions and others, a program design language (PDL) is defined. PDL has an outer syntax providing various control flow constructs, jobs, procedures and modules, and some structuring of data. Attention is then concentrated, in the central portion of the book, on methodical analysis of control flow, including the rewriting of 'unstructured' into 'structured' programs by creating, where necessary, new variables to represent the value of the text pointer. Chapters on program reading and correctness proofs then follow, leading into the last part of the book. This last part is concerned with designing and writing structured programs and is based on a number of small examples, each illustrated by comparing a faulty, intuitive, heuristic design with the recommended form of rigorous stepwise refinement. The exposition in this part is not, perhaps, entirely successful. The main examples (long division, making change, tic tac toe) invite discussion of algorithmic aspects which lie far from most programmers' daily concerns. The classic IBM pollution reporting problem is a more typical task, but is handled rather less satisfactorily and with less confidence.

In some ways this is an old-fashioned book. The underlying idea of a program is that of a hierarchy of procedures; there is no mention of processes or of parallelism. The notion of a 'job' in the PDL is a very primitive, implementation based notion derived essentially from OS/360. But these defects are less than the book's virtues. Above all, it conveys clearly the idea that program development should be a rigorous activity relying on sound theoretical foundations: that is the book's chief message, and it deserves to be heard.

M. A. JACKSON (London)

Structured Systems Analysis: Tools and Techniques, by Chris Gane and Trish Sarson, 1979; 241 pages. (*Prentice-Hall*, £12·05)

The tools and techniques recommended and briefly described are these: data flow diagrams (DFDs), hierarchically decomposed; data dictionary; decision tables and decision trees; narrative procedural description in 'structured English' or 'tight English'; data base normalisation into third normal form; and program design according to Constantine's methods, as expounded by Myers, Yourdon and others. As a compendium of some well loved ideas the book passes muster. Most of the ideas are described very superficially, but there are references to more substantial works: Date's book on data base systems, Pollack or London on decision tables, Yourdon and Constantine on structured design. A reader interested in understanding and using these tools would need to supplement this book by studying many of the references given. This is no criticism: much of the value of the book lies in the conjunction of the various tools, which is claimed to provide at least the basis of a method.

The chief novelties are the 'structured' or 'tight' English, and the data flow diagrams. The former is no more than a slight relaxation of a typical pseudocode to make it more acceptable to a lay customer; 'structured' English adds a small dose of syntactic sugar to the pseudocode; 'tight' English adds a rather larger dose of syntactic salt which seems, to this reviewer at least, to spoil the dish. There is more substance in the data flow diagramming method. The general idea is that the function of the system is decomposed into lower level functions which communicate by sequential data flows and by updating globally accessible data stores. A function is an 'order to a dumb clerk', capable of 'being carried out in a simple clerical circumstance in 5-30 minutes'. Decomposition proceeds hierarchically, with some obvious rules (not always observed) relating the successive levels. Much attention is paid to the mechanics of the representation (such as the avoidance of crossing data flows). The claim is made that the DFD technique is 'logical' rather than 'physical', avoiding premature implementation decisions, but this claim is not well founded: the partitioning of the system established in the data flow diagrams is retained in the final implementation, and there is no discussion of how the system might be repartitioned during the implementation stage.

Too much of the book is taken up by vague generalisations, and there is little to exert the reader's intellect. But the book is not without content, and will certainly appeal to some. Enthusiasts could usefully compare it with de Marco's book on the same subject and from the same stable.* M. A. JACKSON (London)

Reference

TOM DE MARCO, (1979). *Structured Analysis and System Specification* Prentice-Hall, £16·25

Speech Communication with Computers, edited by Leonard Bolc, 1979; 206 pages. (*Carl Hanser Verlag* and *Macmillan*, £10·00)

The mechanical analysis of speech was recognised as a challenging problem long before the advent of computers. When computers became available the recognition of speech by computer quickly became an established problem, and its imminent solution is announced with great regularity. For instance it is reported that in 1965 it was announced that Dartmouth College would have a recogniser 'for any language within two years'.

The present volume is a collection of six articles describing the work in progress at laboratories in the United States and Europe in the mid-1970's. An article from Carnegie-Mellon University describes a comprehensive project, whereas others deal with more specialised topics, one specialised topic being the identification of a speaker, discussed by P. Jesorsky. This volume is intended to be the first in a survey series on natural communication with computers; it is a useful reference and worthy of a place in those libraries where student projects start.

J. J. FLORENTIN (London)