Environment and Planning B, 1980, volume 7, pages 463-484

The arithmetic of shapes

R Krishnamurti
Centre for Configurational Studies, The Open University, Milton Keynes, MK7 6AA, England
Received 28 November 1980, in revised form 15 December 1980

Abstract. Algorithms for the Boolean operations and relations on shapes and labelled shapes are presented.

This paper has two parts. The first part accomplishes two objectives. First, an efficient and uniform representation for shapes is presented, which is based on a linear order on the maximal lines of a shape. Second, simple and efficient algorithms for the Boolean operations on shapes (shape union, difference, and intersection) and the Boolean relations on shapes (subshape and shape equality) are presented. The second part deals with the computational aspects involved in performing the Boolean operations and relations on shapes and labelled shapes. An algorithm for the efficient decomposition of the Boolean operations and relations is presented. Last, the data structures required to implement the shape algorithms are described.

The relevant definitions and notations upon which this paper is based are given in Stiny (1980). Each shape is assumed to be initially described by a set of maximal lines; each labelled shape is given by a shape and an associated set of labelled points.

Rational shapes

A restriction is introduced to limit the class of shapes dealt with in this paper. This restriction—albeit a practical one—is necessitated by the fact that algorithms are defined with respect to some form of computing machine. In a random access machine with limited memory a real number is represented by a finite approximation which is determined by the word size of memory. This makes for inexact arithmetic. Moreover, it is usual for shapes to be drawn on some kind of graphics device such as a visual display unit or a digital plotter. On these devices only a limited number of points can be addressed, and the location of each point is given by an integral multiple or a pair of integral multiples of a unit of measurement. Since correct algorithms require exact arithmetic it is convenient to consider just those shapes which can be so described. Therefore, attention will be restricted to shapes which are, in the mathematical sense, rational. The following definition makes the notion of a 'rational shape' precise.

Definition: A point p is *rational* if and only if each of its coordinates $x_1(p), \ldots, x_d(p)$, $d \geqslant 2$, can be expressed as the ratio of two integers. A labelled point $p{:}A$ is *rational* if and only if p is rational. A line l, $l = \{p_1, p_2\}$, is *rational* if and only if its end points, p_1 and p_2, are rational. A shape s is *rational* if and only if each of its maximal lines is rational. A labelled shape σ, $\sigma = \langle s, p\rangle$, is *rational* if and only if s is rational and every labelled point in point set P is rational.

The ratio of two integers, r_n/r_d, may be expressed as the ordered pair, $\langle r_n, r_d\rangle$, which, in turn, may be described by its unique primitive form. A pair of integers, $\langle r_n, r_d\rangle$ is *primitive* if and only if the following conditions are satisfied:
(a) r_n and r_d are integers,

(b) $r_d \geqslant 0$,
(c) r_n and r_d are relatively prime—that is, there is no positive integer, say $k \neq 1$, such that k divides both r_n and r_d.
When $r_n < 0$, the primitive is said to be negative.

Examples of primitives are now given. An integer n is described by the primitive $\langle n, 1\rangle$; infinity ∞ is described by the primitive $\langle 1, 0\rangle$; zero 0 is described by the primitive $\langle 0, 1\rangle$. A rational number r, $r = r_n/r_d$, can always be reduced to its primitive form by applying Euclid's greatest common denominator (gcd) algorithm (see for instance, Aho et al, 1974, pages 300–302). The following procedure outlines the steps involved in determining the primitive form for the ratio of two integers, $r = r_n/r_d$, $r_d > 0$. Let $a = \gcd(|r_n|, |r_d|)$, where, for any integer q, $|q|$ denotes the absolute value of q. Let $b = |r_d|/r_d$. Then, r is described by the primitive $\langle (br_n)/a, |r_d|/a\rangle$.

Primitives allow one to compare two numbers for equality. Let $r^* = \langle r_n^*, r_d^*\rangle$ denote the primitive of the number r where r may be rational, integral, or infinity. Two numbers r_1 and r_2 are *equivalent* if and only if their primitives r_1^* and r_2^* are equal. That is, $r_{1,n}^* = r_{2,n}^*$ and $r_{1,d}^* = r_{2,d}^*$.

Arithmetic computation involving primitives can be conveniently described by the following algorithmic notation. The expression:

variable $\leftarrow_p$ (expression)

signifies that the 'variable' is assigned the primitive form of the number that results from the 'expression'. For example, let u, v, w be primitives. Then

$w \leftarrow_p (u+v)$

is equivalent to the steps:
set $w_n' = u_n v_d + u_d v_n$,
set $w_d' = u_d v_d$,
define w' to be the ratio of integers, w_n'/w_d',
assign to w the primitive form of w' by means of the procedure outlined above.

Part 1

A representation for shapes

A good shape algorithm requires a good internal representation for shapes. It is widely accepted that by a good algorithm is meant one which has computational *time-complexities* and *space-complexities* (Aho et al, 1974, pages 12–14) which are polynomials of its input size. For shape algorithms, the inputs are essentially the maximal lines and the labelled points of the labelled shapes to which they apply. How these labelled shapes are represented, in turn, depends upon the type and nature of the computational steps involved in performing the shape operations and relations. It is clear that shape algorithms must satisfy the following two computational requirements:
1. There is an effective mechanism or method for determining whether or not two lines are colinear.
2. There is an effective mechanism or method for determining whether or not a point is coincident with a line.
Once these requirements can be satisfied it is fairly straightforward to determine whether or not two colinear lines share a common line.

A simple representation for the maximal lines of a shape will completely satisfy these requirements. The representation for two-dimensional rational labelled shapes is demonstrated, and the extensions to d-dimensional, $d \geqslant 3$, rational labelled shapes will be apparent.

Line descriptors

Consider an infinite line drawn in a cartesian coordinate system. Its equation can be written as

$y = x(\text{slope}) + y\text{-intercept}$, if line is nonvertical,

$x = x\text{-intercept}$, if line is vertical.

Every maximal line may be viewed as a finite line segment on some infinite line whose equation takes on one of the above forms. Consequently, any two maximal lines which correspond to finite line segments on the same infinite line are *colinear.* Hence, every maximal line may be associated with a *line descriptor*, ψ, which is the ordered pair given by $\psi = \langle \mu, \nu \rangle$, where μ is the slope, and ν is the y-intercept if the line is nonvertical, and the x-intercept if the line is vertical. It is easy to show that for a rational line, its line descriptor may be represented by an ordered pair of primitives. Colinear lines have identical line descriptors and noncolinear lines do not.

Coincident points and shared lines

Let J and K be ordered pairs of primitives, where $J = \langle j_1, j_2 \rangle$, and $K = \langle k_1, k_2 \rangle$. Then, J is *less* than K, denoted by $J < K$, if and only if either $j_1 < k_1$ or $j_1 = k_1$ *and* $j_2 < k_2$. Here, $<$ is an order relation.

The ordering, $<$, on the points introduces an orientation on the maximal lines in the following manner. One may assume, without loss of generality, that $l = \{p_1, p_2\}$ is a line the end points of which satisfy $p_1 < p_2$. Then, l is represented as the ordered pair given by $l = \langle p_1, p_2 \rangle$. The endpoints p_1 and p_2 are, respectively, referred to as the *tail* and *head* of l. For any line l, tail of $l <$ head of l.

It follows, therefore, that a point which is coincident with a line must lie between the tail and head of the line. In other words, a point p is *coincident* with the line given by $l = \langle p_1, p_2 \rangle$ if and only if either $p_1 = p$, or $p_2 = p$, or p satisfies (1) the equation of the line; and (2) the inequality $p_1 < p < p_2$. Since, for the shape algorithms considered in this paper, only colinear lines are compared, condition (1) is always satisfied.

The ordering of the points allows one to compare two colinear lines for overlap. This comparison, as will be seen, is important to the effectiveness of the shape algorithms. Two colinear lines l_1 and l_2 are said to *share* a common line if and only if tail of $l_1 <$ head of l_2 *and* tail of $l_2 <$ head of l_1. Their common line $\hat{l}$, $\hat{l} = \langle \hat{p}_1, \hat{p}_2 \rangle$, is given by $\hat{p}_1$ which is the *maximum* of the tails of l_1 and l_2, and $\hat{p}_2$ which is the *minimum* of the heads of l_1 and l_2.

Figure 1 presents examples of pairs of colinear lines, l_1 and l_2, some of which overlap, in which case their common line is indicated by a thick line. The end points of l_1 and l_2 are, respectively, denoted by ○ and ●. End points common to l_1 and l_2 are denoted by ⊖. For each pair of colinear lines the conditions on their tails and heads are stated.

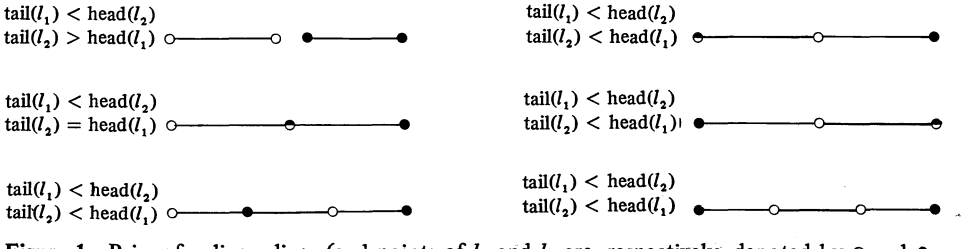


Figure 1. Pairs of colinear lines (end points of l_1 and l_2 are, respectively, denoted by ○ and ●; common end points are denoted by ⊖).

Ordering maximal lines

It is now possible to present the representation for shapes based on a linear ordering on the maximal lines of the shape. Let s be a shape described by a set of maximal lines each of which is associated with a line descriptor. The line descriptor induces a natural partition of s into disjoint subshapes $s_1, ..., s_n$, where each s_i, $1 \leqslant i \leqslant n$, consists of multiple colinear lines. Each maximal line in s_i, $1 \leqslant i \leqslant n$, has the same line descriptor, $\langle \mu_i, \nu_i \rangle$. Moreover, whenever $i \neq j$, $1 \leqslant i, j \leqslant n$, the shape intersection $s_i \cdot s_j$ is empty, and s is described by shape union: $s = s_1 + s_2 + ... + s_n$. Figure 2 presents a shape and its decomposition into disjoint subshapes each of which comprises colinear maximal lines.

The subshapes $s_1, ..., s_n$ of s are arranged so that their line descriptors form a linearly ordered list. That is, $\langle \mu_1, \nu_1 \rangle < ... < \langle \mu_n, \nu_n \rangle$. Each subshape, s_i, $1 \leqslant i \leqslant n$, in turn, is represented by a linearly ordered list of colinear maximal lines L_i, $L_i = \langle l_{i,1}, ..., l_{i,m} \rangle$ where each element in L_i has the same line descriptor $\langle \mu_i, \nu_i \rangle$. (Where no confusion can arise subscripts will be omitted.) The lines in L, $L = \langle l_1, ..., l_m \rangle$, are arranged so that whenever $1 \leqslant j < k \leqslant m$, head of $l_j <$ tail of l_k, which is denoted by $l_j < l_k$. Such a linear ordering of the maximal lines in s_i is always possible since any pair of colinear maximal lines in the *same* shape cannot overlap.

The labelled points for labelled shapes are likewise treated. The labelled points are arranged into lists, each list consisting of all the labelled points which have the same label. The lists are linearly ordered according to a lexicographical ordering on the labels. The labelled points in each list may be arranged according to the order relation $<$.

Thus, every labelled shape σ, $\sigma = \langle s, P \rangle$, is represented by the ordered pair σ given by $\sigma = \langle L, P \rangle$. L is a linearly ordered list of linearly ordered lists of maximal lines, $L = \langle L_1, ..., L_n \rangle$, where L_j, $1 \leqslant j \leqslant n$, contains all the maximal lines in s with the same line descriptor, $\langle \mu_j, \nu_j \rangle$ and $\langle \mu_1, \nu_1 \rangle < ... < \langle \mu_n, \nu_n \rangle$. P is a linearly ordered list of

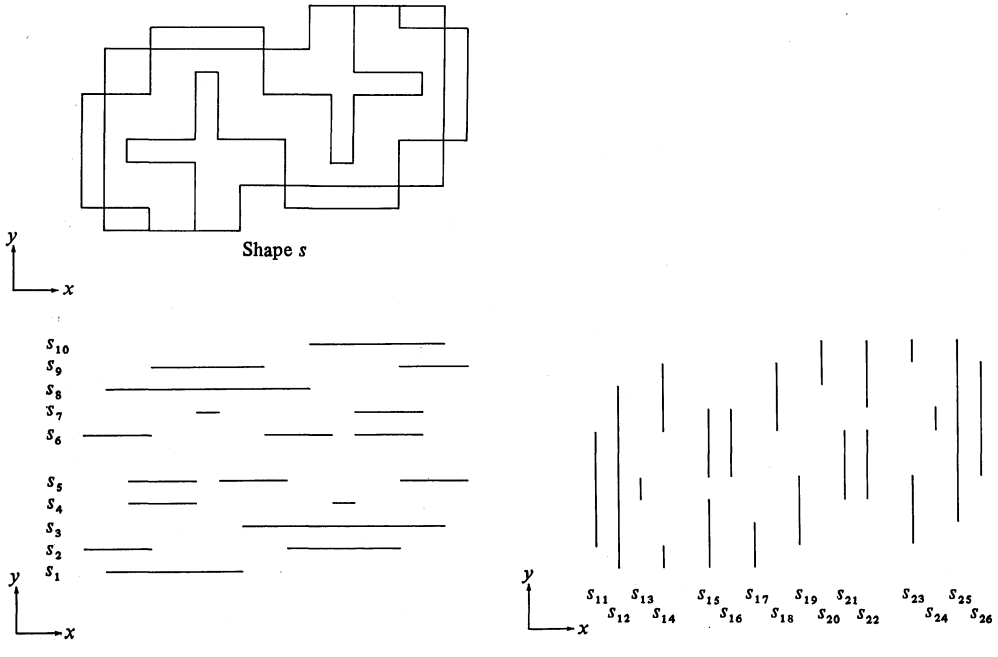


Figure 2. The decomposition of a shape s into disjoint subshapes, each of which comprises colinear maximal lines.

linearly ordered lists of labelled points, $P = \langle P_1, ..., P_m \rangle$ where P_k, $1 \leqslant k \leqslant m$, contains all the labelled points of P with the same label, A_k, and $A_1 < ... < A_m$.

List comparison properties

The linear-order based reorganisation of labelled shapes yields the following list comparison properties which are given for the maximal lines in the shape. Similar properties hold for the labelled points of the shape. For any list N, let $\psi(N)$ denote the line descriptor for the maximal lines in N. Then:

1. Let $L = \langle l_1, ..., l_{n_L} \rangle$ and $M = \langle m_1, ..., m_{n_M} \rangle$ be linear lists of colinear maximal lines such that $\psi(L) = \psi(M)$. Then, if l_j, $1 \leqslant j \leqslant n_L$, is the first line in L which shares a common line with m_k, $1 \leqslant k \leqslant n_M$, in M, then the lines $m_{k'}$, $k < k' \leqslant n_M$, do not share a common line with the lines $l_{j'}$, $1 \leqslant j' \leqslant j$. This observation enables one, as will be proved later, to compare two lists of colinear lines in a time *linear* in the number of maximal lines in both lists.
2. Let $L = \langle L_1, ..., L_{n_L} \rangle$ and $M = \langle M_1, ..., M_{n_M} \rangle$ be lists of lists of colinear maximal lines. Then, if L_j, $1 \leqslant j \leqslant n_L$, and M_k, $1 \leqslant k \leqslant n_M$ are lists such that $\psi(L_j) = \psi(M_k)$, the lists $L_{j'}$, $1 \leqslant j' < j$, and $M_{k'}$, $k < k' \leqslant n_M$ satisfy: $\psi(L_{j'}) < \psi(M_{k'})$.

Shape arithmetic

Let $\square$ be a variable over the Boolean operations and relations. Then, the shape operation or relation $\sigma_1 \square \sigma_2$ is equivalent to the ordered pair $\langle L_1 \square L_2, P_1 \square P_2 \rangle$, where $L_1 \square L_2$ represents the *shape* operation or relation, and $P_1 \square P_2$ represents the corresponding *set* operation or relation. For example, let $\square$ be the shape union operator. Then, $\sigma_1 + \sigma_2$ is the labelled shape given by the shape represented by the shape union: $L_1 + L_2$, and its associated set of labelled points represented by the set union: $P_1 + P_2$. Again, for example, let $\square$ be the subshape relation, $\leq$. Then, $\sigma_1 \leq \sigma_2$ if and only if L_1 represents a subshape of the shape represented by L_2, and P_1 represents a subset of the set of labelled points represented by P_2.

The shape operation or relation, $L_1 \square L_2$, can be decomposed into a list of shape operations or relations on pairs of shapes each of which consists of colinear maximal lines. Since the Boolean operations and relations on shapes are defined in terms of overlapping lines (see Stiny, 1980), it is sufficient to compare the maximal lines in lists, one each from L_1 and L_2, which share the same line descriptor.

For convenience, let the subscripts be omitted, and let L_1 and L_2 be referred to as L and M, $L = \langle L_1, ..., L_{n_L} \rangle$ and $M = \langle M_1, ..., M_{n_M} \rangle$. Let $\emptyset$ denote the *empty* list of maximal lines. Suppose $\square$ is an operator. Then, $L \square M$ is the list N, say, where $N = \langle N_1, ..., N_{n_N} \rangle$, and each N_i, $1 \leqslant i \leqslant n_N \leqslant n_L + n_M$, is a list of colinear maximal lines given by one of the following:

$$N_i = \begin{cases} L_j \square M_k & \text{if } L_j,\ 1 \leqslant j \leqslant n_L, \text{ and } M_k,\ 1 \leqslant k \leqslant n_M, \text{ are lists such that } \psi(L_j) = \psi(M_k) \text{ and } L_j \square M_k \neq \emptyset; \\ L_j & \text{if } \square \text{ and } \cdot \text{ are not the same operator, and } L_j,\ 1 \leqslant j \leqslant n_L, \text{ is a list such that for each } k,\ 1 \leqslant k \leqslant n_M,\ \psi(L_j) \neq \psi(M_k); \\ M_k & \text{if } \square \text{ is the operator } +, \text{ and } M_k,\ 1 \leqslant k \leqslant n_M, \text{ is a list such that for each } j,\ 1 \leqslant j \leqslant n_L,\ \psi(M_k) \neq \psi(L_j). \end{cases}$$

Clearly, N can be arranged so that $\psi(N_1) < ... < \psi(N_{n_N})$.

Suppose, on the other hand, $\square$ is a relation. For integers n and m, let $n \square m$ denote $n \leqslant m$ in the case when $\square$ is the subshape relation, and $n = m$ otherwise. Then, $L \square M$ if and only if $n_L \square n_M$ and for each list L_j, $1 \leqslant j \leqslant n_L$, there is a list M_k, $1 \leqslant k \leqslant n_M$, such that $\psi(L_j) = \psi(M_k)$ and $L_j \square M_k$. Therefore, it follows that all one requires are shape algorithms for pairs of shapes each consisting of colinear

maximal lines, and that the maximal lines in the two shapes have the same line descriptor.

In a similar fashion, $P_1 \square P_2$ can be decomposed into a list of set operations or relations on pairs of sets of labelled points having the same label. Again, all that is required are set algorithms for sets of identically labelled points.

Shape algorithms

Let $\square$ be a Boolean operator on shapes. Then, the *shape expression*

$$\sigma_3 \leftarrow \sigma_1 \square \sigma_2$$

signifies that σ_3 is the labelled shape resulting from the shape operation $\sigma_1 \square \sigma_2$. In the special case, where σ_3 is either σ_1 or σ_2, say σ_1, the expression

$$\sigma_1 \leftarrow \sigma_1 \square \sigma_2$$

signifies that σ_1 is *replaced* by the result of the shape operation $\sigma_1 \square \sigma_2$. The latter expression is particularly relevant to the shape grammar formalism. In any shape grammar there is a current shape denoted by γ. Suppose a shape rule, denoted by $\alpha \rightarrow \beta$, applies to γ. That is, there is a euclidean transformation τ such that $\tau(\alpha) \leq \gamma$. Then, shape rule application is described by the following two shape expressions in sequence:

$$\gamma \leftarrow \gamma - \tau(\alpha) , \qquad \gamma \leftarrow \gamma + \tau(\beta) .$$

Algorithms for shape rule application are taken up in greater detail in Krishnamurti (1981).

The following shape expressions are considered:

(a) $\sigma_1 \leftarrow \sigma_1 + \sigma_2$,
(b) $\sigma_1 \leftarrow \sigma_1 - \sigma_2$,
(c) $\sigma_3 \leftarrow \sigma_1 \cdot \sigma_2$,
(d) is $\sigma_2 \leq \sigma_1$?
(e) is $\sigma_1 = \sigma_2$?

The shape expressions (a) and (b) are chosen with the view to implementing the shape grammar formalism. Other shape expressions are possible. For instance, interested readers can devise their own algorithms for the shape expressions $\sigma_3 \leftarrow \sigma_1 + \sigma_2$. To that end, they may find the presentation in this paper for the shape expression (c) useful.

The inputs to the shape algorithms are ordered lists of colinear maximal lines, L and M, both lists sharing the same line descriptor. It is assumed that M is nonempty. The proofs for the correctness of the algorithms are provided in their descriptions, and where necessary illustrations are provided to facilitate explanation.

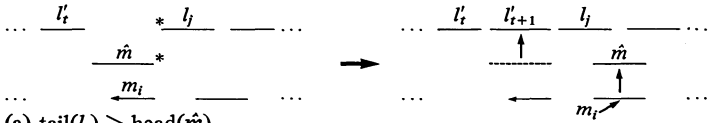
(a) *Shape union:* $L \leftarrow L+M$

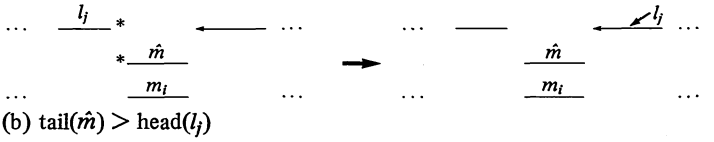
Step 0 (Is L empty?) If L is empty, go to *step 8*. Otherwise, set $i \leftarrow 1$, and select the first line, m_1, in M and copy it into $\hat{m}$ which is referred to as the 'working line'. Set $j \leftarrow 1$, and select the first line, l_1, in L.

Step 1 If tail of $l_j \leq$ head of $\hat{m}$, go to *step 3*. Otherwise, $\hat{m}$ shares no common line with any line $l_{k \geq j}$ in L, and $\hat{m} < l_j$. This condition is illustrated in figure 3(a). Insert $\hat{m}$ into L as the maximal line immediately preceding l_j in the list, and continue with *step 2*.

Step 2 If M is not exhausted, set $i \leftarrow i+1$, and select the next line, m_i, in M and copy it into $\hat{m}$. (Notice that at no stage in the execution of the algorithm is the head of $\hat{m}$ altered, and therefore, at all times $\hat{m} < m_{k>i}$.) Go to *step 1*. Otherwise, M is exhausted, and go to *step 9*.

Step 3 (Determine if $\hat{m}$ and l_j share a common line.) If tail of $\hat{m} \leq$ head of l_j, go to *step 5*. Otherwise, continue with *step 4*. [The working line $\hat{m}$ shares no common line with any line preceding and including l_j in L and $\hat{m} > l_j$. But $\hat{m}$ may share a common line with line(s), $l_{k>j}$, in L. This situation is illustrated in figure 3(b).]
Step 4 If L is not exhausted, set $j \leftarrow j+1$, and select the next line, l_j, in L and go to *step 1*. Otherwise, insert $\hat{m}$ as the last line in L and go to *step 8*. (The reader can verify that $\hat{m}$ is a maximal line.)
Step 5 As $\hat{m}$ and l_j either share a common line or share an end point, they can therefore be combined to form a single line. This is illustrated in figure 3(c). Set tail of $\hat{m}$ and l_j to the *minimum* of the tails of $\hat{m}$ and l_j.
Step 6 ($\hat{m}$ and l_j now have the same tail; one of these lines must contain the other, and the appropriate line is discarded from further consideration.) Go to *step 7* if head of $l_j \geq$ head of $\hat{m}$, otherwise, delete l_j from L and go to *step 4*. [At this stage $\hat{m}$ contains l_j. It will be assumed, for convenience, that the subscripts of the lines in L are not altered. That is, at any stage of the algorithm the line currently preceding l_j in the list L may not necessarily be the line originally subscripted l_{j-1}. This step and the next are illustrated in figures 3(d) and 3(e).]
Step 7 As l_j contains $\hat{m}$, then $\hat{m}$ is not maximal and can be ignored. Go to *step 2*.
Step 8 (All the lines, if any, in L have been examined. There may still be some unexamined lines in M.) Copy all the unexamined lines in M in their order, and attach them to the end of list L.
Step 9 (Finishing touch.) All the lines both in L and in M have been examined. L contains the maximal lines of the shape union in their sorted order.





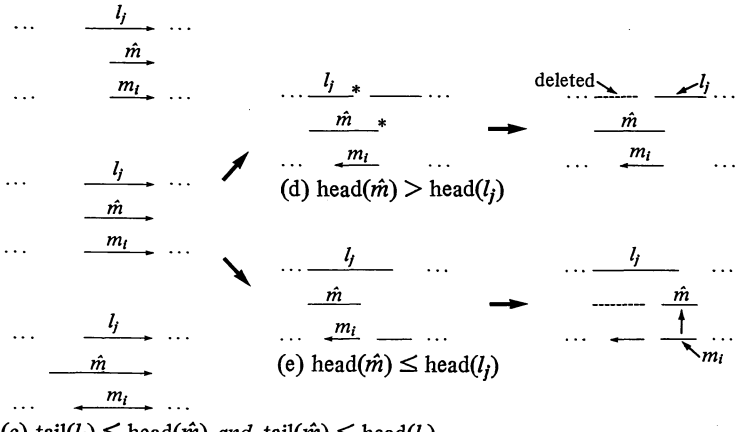
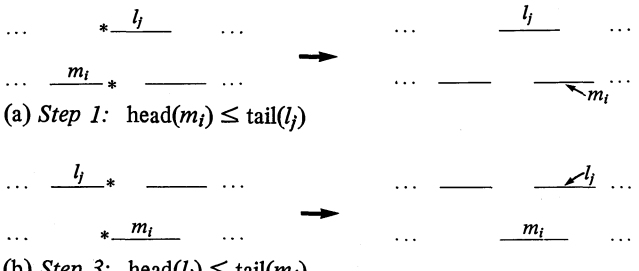


Figure 3. Some conditions that can arise in the algorithm for shape union (asterisks indicate the end points that are compared, and arrow heads indicate an unfixed end point).

The computational complexity of this algorithm can easily be determined as follows. Let there be n_L lines in L and n_M lines in M. In the worse case, each line, l, in L is compared *once* with the current working line, $\hat{m}$, except when (in *step 2*) tail of $l >$ head of $\hat{m}$ or when (in *step 7*) l contains $\hat{m}$, that is, when m is redefined (*step 2*). But there are n_M redefinitions of m once for each line in M. Therefore, the worst case time bound for shape union is $O(n_L + n_M)$.

(b) *Shape difference:* $L \leftarrow L - M$
Step 0 (It is assumed that L is nonempty; otherwise the shape difference is empty.) Set $i \leftarrow 1$, and select the first line, m_1, in M. Set $j \leftarrow 1$ and select the first line, l_1, in L.
Step 1 If head of $m_i >$ tail of l_j, go to *step 3*. [Otherwise, m_i shares no common line with l_j, through its successor line in M, m_{i+1}, may. See figure 4(a).]
Step 2 If M is not exhausted, set $i \leftarrow i+1$, and select the next line, m_i, in M and then go to *step 1*. Otherwise, the procedure is finished, and exit from algorithm.
Step 3 If head of $l_j >$ tail of m_i, go to *step 5*. [Otherwise, l_j shares no common line with m_i, though its successor line in L, l_{j+1}, may. See figure 4(b).]
Step 4 If L is not exhausted, set $j \leftarrow j+1$, and select the next line, l_j, in L and then go to *step 1*. Otherwise, the procedure is finished, and exit from algorithm.
Step 5 The lines l_j and m_i share a common line, and their corresponding tails and heads can now be compared. One of the following four cases must arise:
(1) $l_j - m_i = l_A$, $l_A = \langle$tail of l_j, tail of $m_i\rangle$, when tail of $m_i >$ tail of l_j, *and* head of $m_i \geq$ head of l_j;
(2) $l_j - m_i = l_B$, $l_B = \langle$head of m_i, head of $l_j\rangle$, when tail of $l_j \geq$ tail of m_i, *and* head of $l_j >$ head of m_i;
(3) $l_j - m_i = l_A + l_B$, when tail of $m_i >$ tail of l_j *and* head of $l_j >$ head of m_i;
(4) $l_j - m_i$ is the empty line, when tail of $l_j \geq$ tail of m_i *and* head of $m_i \geq$ head of l_j.
Compare the corresponding tails and heads of l_j and m_i. Depending upon the four above mentioned cases, one of the following is performed:
Case (1): replace the line l_j in L by l_A and go to *step 4*.
Case (2): replace the line l_j in L by l_B and go to *step 2*.
Case (3): replace the line l_j in L by l_B. Insert a copy of l_A to precede l_j in the list. That is, $l_A < l_B$ in L. Go to *step 2*.
Case (4): delete l_j from L. Go to *step 4*.
(Figure 5 pictorially illustrates the four cases mentioned above.)

To complete the proof of the correctness of the algorithm, the following observations are made:
1. If l_j and l_k are two successive lines in L which share a common line with m in M, then $l_j - m$ and $l_k - m$ are relatively maximal, and, moreover, the order $<$ between the lines in L is preserved.



(a) *Step 1:* head(m_i) $\leq$ tail(l_j)

(b) *Step 3:* head(l_j) $\leq$ tail(m_i)

Figure 4. Two conditions that can arise in the algorithm for shape difference (asterisks indicate the end points that are compared).

2. If m_i and m_k are two successive lines in M which share a common line with l in L, then $l - m_i$ and $l - m_k$ are relatively maximal, and, moreover, the order $<$ between the lines is preserved.

As in the case of shape union, the computational complexity for shape difference is $O(n_L + n_M)$ where n_L and n_M are, respectively, the numbers of maximal lines in L and M.

(c) *Shape intersection:* $N \leftarrow L \cdot M$

L may be assumed to be nonempty, otherwise the shape intersection is empty. Shape intersection is essentially the complement of the algorithm for shape difference. Only *steps 0* and *5* are modified. In addition to those statements, in *step 0* the following initialization statement must be included:

$$N \leftarrow \emptyset\,, \qquad k \leftarrow 0\,.$$

(That is, initialize the result list to be empty. k is an index to lines in N.)

Step 5 is modified to read as:

Step 5' Corresponding to each case in *step 5* for shape difference, the four cases hold:

(1) $l_j \cdot m_i = \langle$tail of m_i, head of $l_j\rangle$, when tail of $m_i >$ tail of l_j *and* head of $m_i \geq$ head of l_j;

(2) $l_j \cdot m_i = \langle$tail of l_j, head of $m_i\rangle$, when tail of $l_j \geq$ tail of m_i *and* head of $m_i <$ head of l_j;

(3) $l_j \cdot m_i = \langle$tail of m_i, head of $m_i\rangle$, when tail of $m_i >$ tail of l_j *and* head of $m_i <$ head of l_j;

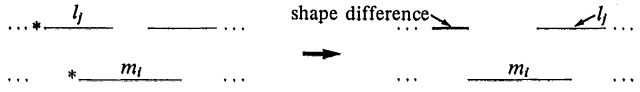
(4) $l_j \cdot m_i = \langle$tail of l_j, head of $l_j\rangle$, when tail of $l_j \geq$ tail of m_i *and* head of $m_i \geq$ head of l_j.

These cases are illustrated in figure 6. The following steps simulate the cases. Set $k \leftarrow k+1$, and let n_k denote the next line in N; n_k represents $l_j \cdot m_i$ which is not empty since l_j and m_i share a common line.

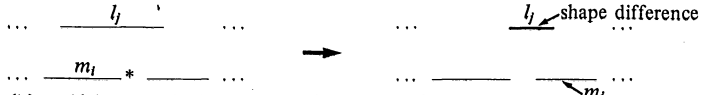
Step 5'.1 Set tail of n_k equal to the *maximum* of the tails of l_j and m_i.

Step 5'.2 If head of $l_j \geq$ head of m_i, set head of n_k equal to the head of m_i, and then go to *step 2.*

Step 5'.3 Otherwise, set head of n_k equal to head of l_j and then go to *step 4.*

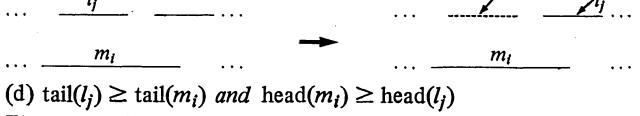


(a) tail(m_i) > tail(l_j) *and* head(m_i) ≥ head(l_j)



(b) tail(l_j) ≥ tail(m_i) *and* head(l_j) > head(m_i)

(c) tail(m_i) > tail(l_j) *and* head(l_j) > head(m_i)

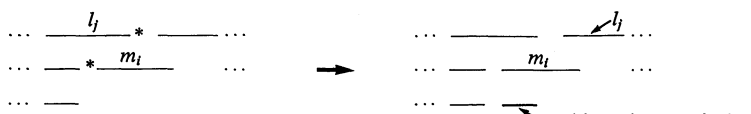


(d) tail(l_j) ≥ tail(m_i) *and* head(m_i) ≥ head(l_j)

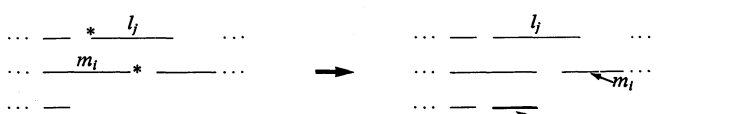
Figure 5. The four cases of *step 5* in the algorithm for shape difference (asterisks indicate the end points of the lines in the shape difference).

The proof of the correctness for the algorithm follows from the observation that the intersection of two overlapping maximal lines is a maximal line. The computational complexity for shape intersection is linear in the number of lines in L and M.

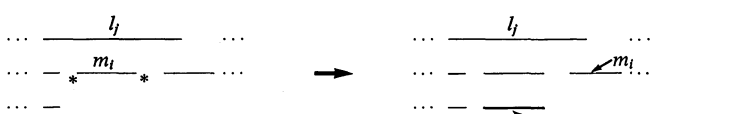
(d) *Subshape and equality:* $M \leq L$? and $M = L$?
In this case L is nonempty, otherwise the relationship does not hold (since M is assumed to be nonempty). The subshape algorithm will be considered here; the equality algorithm is equivalent to determining if the two lists are identical. A Boolean variable *flag* which returns a value **true** if $M \leq L$, and **false** otherwise.
Step 0 Set *flag* ← **true**. Set $j \leftarrow 1$, and select the first line, l_1, in L. Set $i \leftarrow 1$, and select the first line, m_1, in M.
Step 1 If tail of $m_i <$ head of l_j, go to *step 3*.
(Otherwise, l_j does not share a common line with m_i though its successor, l_{j+1}, in L may.)
Step 2 If L is not exhausted, set $j \leftarrow j+1$, and select the next line, l_j, in L and then go to *step 1*.
Otherwise, $\leq$ does not hold. Set *flag* to **false** and go to *step 6*.
Step 3 If tail of $l_j <$ head of m_i, go to *step 4*.
Otherwise, there is no line in L which contains m_i. Set *flag* to **false** and go to *step 6*.
Step 4 (l_j and m_i share a common line. Does l_j contain m_i?)
If tail of $m_i \geq$ tail of l_j and head of $m_i \leq$ head of l_j, go to *step 5*.
Otherwise, l_j does not contain m_i. Set *flag* to **false** and go to *step 6*.
Step 5 (l_j contains m_i. Choose next line in M.)
If M is not exhausted, set $i \leftarrow i+1$, and select next line, m_i, in L and then go to *step 1*.



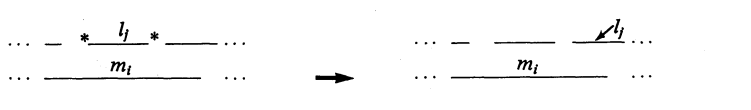
(a) tail(m_i) > tail(l_j) *and* head(m_i) ≥ head(l_j)



(b) tail(l_j) ≥ tail(m_i) *and* head(l_j) > head(m_i)



(c) tail(m_i) > tail(l_j) *and* head(l_j) > head(m_i)



(d) tail(l_j) ≥ tail(m_i) *and* head(m_i) ≥ head(l_j)

Figure 6. The four cases of *steps 5'* in the algorithm for shape intersection (asterisks indicate the end points of the common line).

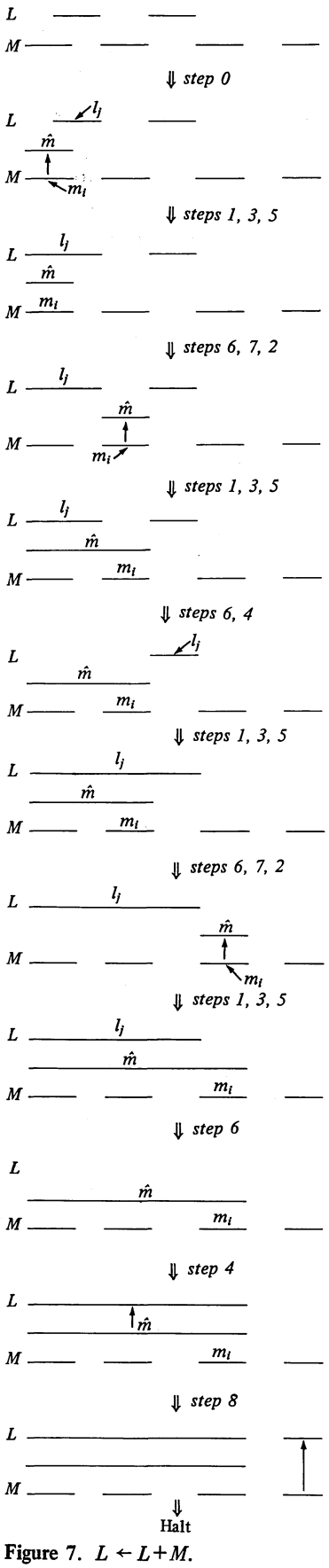


Figure 7. $L \leftarrow L+M$.

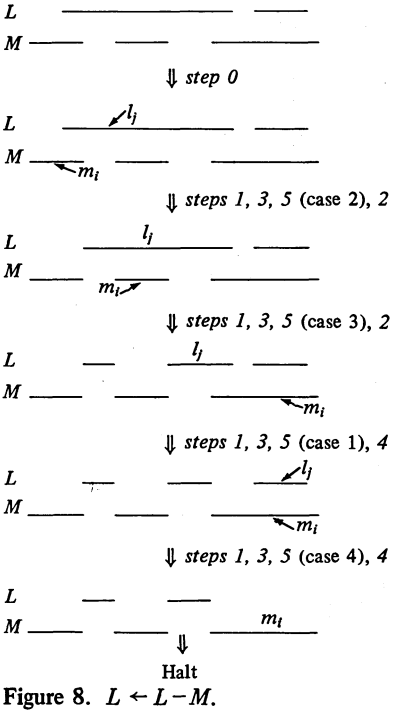


Figure 8. $L \leftarrow L-M$.

Step 6 (Finishing touch.) The procedure is finished, and *flag* contains the Boolean value representing whether or not the relation $M \leq L$ holds. Exit from algorithm with the value in *flag*.

Figures 7 through 10 present pictorial descriptions of the workings of the algorithms on sample shapes.

The corresponding operations and relations on sets of identically labelled points can be performed by means of the standard procedures for manipulating linear lists (see, for instance, Horowitz and Sahni, 1976, chapter 2).

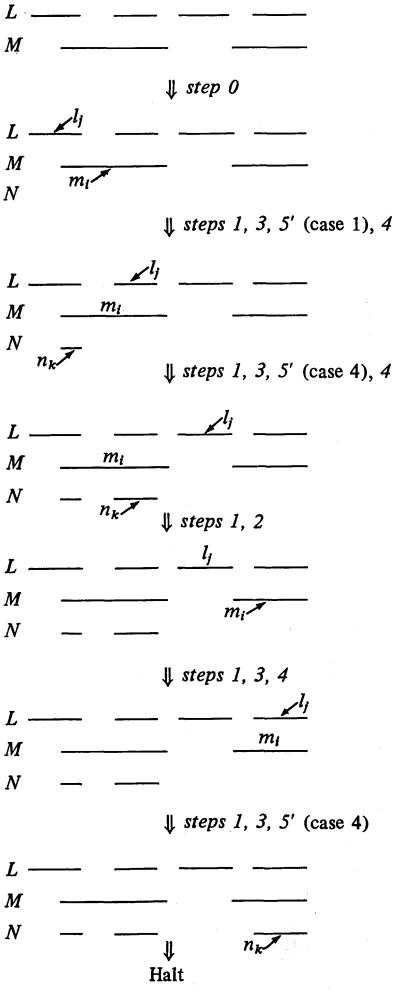


Figure 9. $N \leftarrow L \cdot M$.

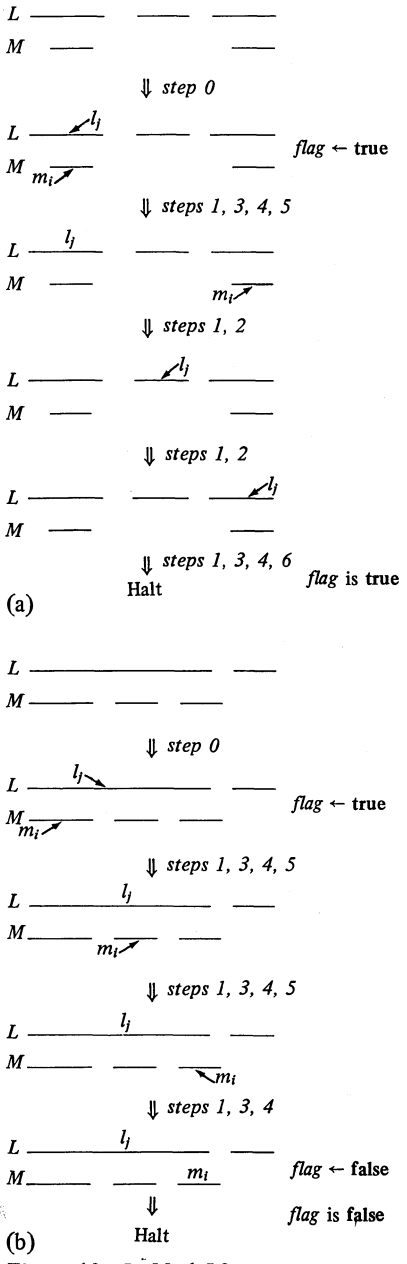


Figure 10. Is $M \leq L$?

Part 2

Efficient decomposition of the Boolean operations and relations

It has been shown that a Boolean operation or relation on labelled shapes can be decomposed into (a) a list each element of which is the shape operation or relation on linear lists of colinear maximal lines, and (b) a list each element of which is the set operation or relation on linear lists of labelled points having the same label. Now consider the following problem.

Suppose U and V are ordered lists of lists of elements, where $U = \langle U_1, ..., U_n \rangle$, $V = \langle V_1, ..., V_m \rangle$, and the U_ks and V_js are sorted according to their respective *keys* $\langle u_1, ..., u_n \rangle$ and $\langle v_1, ..., v_m \rangle$. That is, $u_1 < ... < u_n$, and $v_1 < ... < v_m$. Let $\square$ be an operator. The problem is that for each list V_j with key v_j in V to *search* the list U for a list U_k such that $u_k = v_j$. If the search is successful U_k is replaced by $U_k \square V_j$, which may lead to the *deletion* of U_k from U in the case when $\square$ is a difference or intersection operator. This occurs when $U_k \square V_j$ is empty. If the search is unsuccessful a copy of V_j is inserted into the list U in the case when $\square$ is a union operator, but at the same time preserving the linear ordering of the elements of U. Here U is a dynamic list and V is a static list. The analogy to the shape algorithms is obvious.

The crucial computation question related to the problem is: how *effectively* can this list searching, and/or list insertion/deletion be carried out? The answer to this depends to a large extent on the internal organization for the list. Two possible types of data structures are considered: (a) *linked lists*, and (b) *balanced binary trees.* Linked lists are useful when the lists are static and reasonably small in size, which is often the case with shape rules. Balanced binary trees are useful when the lists are dynamic and constantly increasing in size, which is generally the case with the current shape in the shape grammar formalism. Algorithms for searching, insertion, and deletion on linked lists are straightforward (see, for instance, Aho et al, 1974). In the remainder of this section a search algorithm is presented when the list U is organized as a balanced binary tree. This algorithm utilizes the Brown–Tarjan (1979) fast merge algorithm.

Organizing a linear list as a balanced binary tree

It will be assumed that the reader is familiar with binary tree terminology (see, for instance, Knuth, 1973b) with the exception that the term 'son' is now replaced by 'offspring'. A binary tree is *height-balanced* if the height of the left subtree of any node never differs by more than ±1 from the height of the right subtree of the node. The *height* of a tree is the length of the longest path from the root to a leaf node.

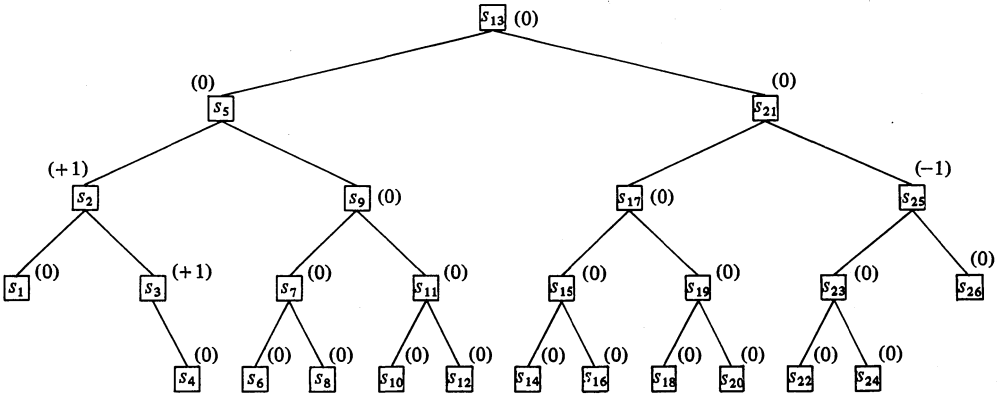


Figure 11. A balanced tree representation for shapes in figure 2 ([s] denotes the node representing shape s, and the numbers within the brackets are the balance factors).

A *leaf node* has no offsprings. The *left* and *right* subtrees of a node are, respectively, the trees rooted at the left and right offsprings of the node.

A linear list is *organized* as a balanced binary tree as follows. Each node in the tree represents a list element, and is referenced by its key. If k is the key presented by a node, then all nodes in its left subtree have keys $<k$, and all nodes in its right subtree have keys $>k$. The proper location of a node in the tree with a given key value can be determined by a standard binary tree search (Knuth, 1973a) in O(lb$n$) time, where n is the list size. The linear list can be reproduced in its original list order by an inorder traversal (Knuth, 1973a) of the tree in a time proportional to the list size. Figure 11 gives a balanced tree representation for the shape in figure 2.

A balanced tree search algorithm

The obvious method for comparing U and V is to take each key v_j in turn, and search the balanced tree for U, for a node associated with a key which equals v_j. If such a node exists, the search is successful. When $\square$ is a difference or intersection operator the algorithm may involve the deletion of the node from the tree. If no such node exists, the search is unsuccessful. When $\square$ is a union operator the algorithm involves inserting a new node with key value equal to v_j into the tree. To insert a node into or to delete a node from the balanced tree requires O(lb$n$) additional steps for rebalancing the tree. Rebalancing becomes necessary whenever the modified tree—as a result either of node insertion or of node deletion—fails the height balancing requirement. Rebalancing will be considered in greater detail later in this section. Knuth (1973b, algorithm 6.2.3A) provides a good description of node insertion into a tree. Deletion, however, is slightly more difficult. For a reasonable discussion of node deletion from a balanced tree the reader is referred to Crane (1972, pages 43–45, and chapter 4 for an ALGOLW description of the algorithm) and Wirth (1976, algorithm 4-64). The computational effort involved in comparing U and V is, in the worst case, O(m lb$m$).

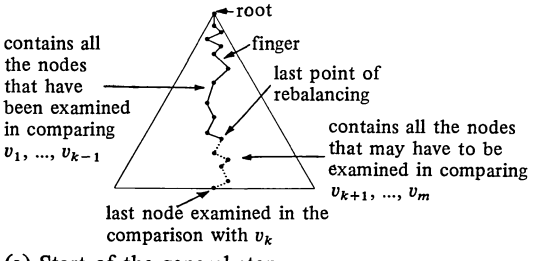
When $n \gg m$ the time bound can be improved. In the case of merging two disjoint lists into a single list Brown and Tarjan (1979) have demonstrated a computational time bound of O[m lb(n/m)]. I believe their contribution to balanced tree search can be used for other forms of list comparisons without materially changing the time complexity. The following material is adopted from their paper. As before, the first step is to search tree U for a node with its key equalling v_1. Once v_1 has been compared, some appropriate algorithmic action takes place which may result in a copy of v_1 together with the list element V_1 being inserted into the tree, or the node whose key equals v_1 being deleted from the tree, or the tree being left as it is. At the start of the general step, the keys $v_1, \ldots, v_k$ have been compared with the nodes in tree U, and a record is maintained of the nodes in the search path from the root to the last node compared with key v_k, which may, in the case of insertion, be the node whose key equals v_k. This path acts as a 'finger' into tree U moving from left to right as the keys of the lists in V are compared; the finger is useful since only nodes to the right of it have to be visited during later comparisons. This follows from list comparison property 2 (see page 467).

The general step comprises two parts. First, the finger is retracted towards the root, just far enough to the position that v_{k+1} 'lies' in the subtree rooted at the end of the finger. The key v_{k+1} is compared with the nodes in this subtree, and the finger is extended to follow the search path. After $m-1$ executions of the general step the algorithm is complete.

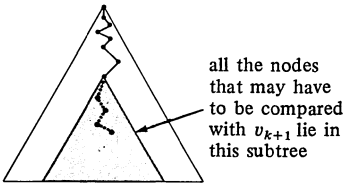
However, this scheme is complicated by the fact that both insertion and deletion into the tree may require rebalancing of the tree. When rebalancing takes place, it may remove a node from the finger path traced out by the search. It is possible to

update the finger to be consistent with the rearrangement, but Brown and Tarjan suggest that it is easier to just 'forget' about the part of the path which is corrupted. That is, to retract the finger to the last, and in the case of insertion the only, point of rebalancing. The algorithm takes the form shown in figure 12. At the start of the general step one now has a record on only part of the search path to the last node compared with v_k. The general step proceeds as before. Notice that the first step need not be treated separately. At the start of the algorithm, the finger is initialized to the root of tree U (which is always on the path to the first comparison), and the general step is executed m times.

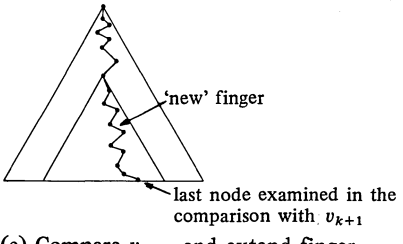
The algorithm can be speeded up by keeping a record of all nodes on the finger path where the finger turns left—that is, those nodes on the path whose left offspring is also in the path. It is easy to show that these are precisely the nodes on the path which have a larger key than the most recently compared item. Then, only these nodes have to be examined in the 'climbing up' phase of the general step.



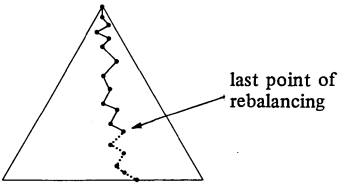
(a) Start of the general step



(b) Climbing up phase



(c) Compare v_{k+1} and extend finger



(d) Rebalancing the tree

Figure 12. The form of the balanced tree search algorithm.

Rebalancing transformations

Rebalancing is required whenever the balanced tree becomes unbalanced—that is, the tree fails to satisfy the height-balance requirement as a result of an insertion or a deletion. Each node is associated with a *balance factor* which is the difference between its right and left subtree heights, and can take on values from *lefttaller* (−1), *balanced* (0), and *righttaller* (+1) which have the obvious interpretations. Consequently, insertion of a node into the left (right) subtree of a node whose balance factor is originally *lefttaller* (*righttaller*), or deletion of a node from the left (right) subtree of a node whose balance factor is originally *righttaller* (*lefttaller*) results in the entire tree becoming unbalanced.

A leaf node is always inserted or deleted. In the case of deletion, if the node to be deleted is not a leaf node, the contents of a specific leaf node is copied into the node designated to be deleted, and the leaf node is deleted instead. Furthermore, the finger is extended to follow the search path to the node to be deleted. Rebalancing can be described as follows. Let x denote the inserted or deleted node. The successive ancestors of x (denoted by z) moving up towards the root along the search

path are examined; and, if necessary, the finger is retracted to z. During this climb one of the following steps is performed.

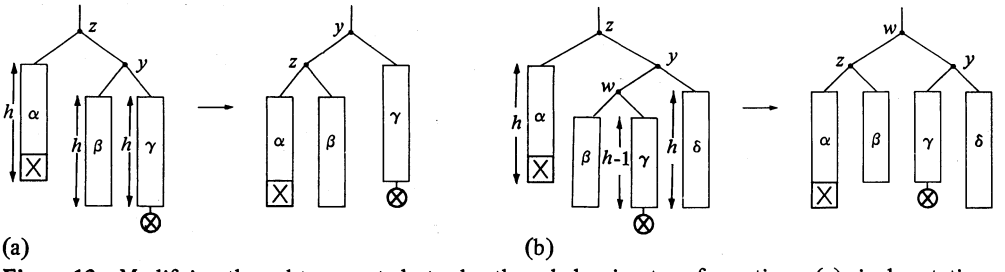
(1) *z is originally balanced:*

Change the balance factor of z to ± 1 as appropriate. In the case of insertion the climb is continued; in the case of deletion the tree remains height-balanced, and the climb is stopped.

(2) *z is originally unbalanced:*

z becomes either balanced or doubly unbalanced. In the latter case, the subtree rooted at z is locally modified via one of two types of transformations: single and double rotation. These are shown in figure 13 wherein the symbol ⊠ indicates the deleted node, and the symbol ⊗ indicates the inserted node. Notice that both in single and in double rotation the mirror-image transformation is possible. In fact there is an additional special case of double rotation which occurs when $h = 0$, and x is an inserted node. In that case, x equals w and the subtrees α, β, γ, and δ are empty. In the case of deletion one still has to continue the climb; for insertion, the rebalancing is completed.

If the root is reached during the climb rebalancing is complete; for insertion the balanced tree has increased in height, and for deletion decreased in height.



(a) (b)

Figure 13. Modifying the subtree rooted at z by the rebalancing transformation; (a) single rotation, and (b) double rotation.

Data structures

The following data structures serve to represent a labelled shape σ, which is described by the ordered pair $\langle L, P\rangle$, where $L = \langle L_1, ..., L_n\rangle$ and $P = \langle P_1, ..., P_m\rangle$. L is an ordered list of ordered lists of colinear maximal lines. P is an ordered list of ordered lists of labelled points having the same label. σ is assumed to be rational. Each data structure is described by the fields of the nodes in the data structure. The internal implementation of the data structures is given in terms of data classes. Each *data class* defines the fields of each data node in the class.

Each node in the data class *POINT* comprises two fields, and for rational shapes each field represents a pair of integers. *POINT* inplements the data structures for the coordinates of a point and the line descriptor for a list of multiple colinear lines. The data structures are:

coordinate node | x | y |,

line descriptor node | μ | ν |.

The fields in the data structures have the obvious interpretation.

The data class *LABEL* consists of one field, A, which stores the label for a list of labelled points sharing the same label.

Each node in the data class *LIST* consists of three fields one of which, referred to as *next*, is a pointer to a node in *LIST*. *LIST* serves to implement linked lists.

Linked lists are used to implement the lists L, P, L_j and P_k. The data structures are:

L_j-node or P_k-node | *key* | N | *next* |,

where *key* is a pointer either to a node in *POINT* which contains the line descriptor for a list of colinear maximal lines, or to a node in *LABEL* which stores the label for a list of labelled points sharing the same label; N is a pointer to a node in *LIST* which is the first node of the linked list representing either the list of colinear multiple lines or the list of labelled points having the same label; and *next* points to the next mode in the linked list in the sense that *key*[*node*] < *key*[*next*[node]].

line-node | *point* | *mark* | *next* |,

where *point* points to a node in *POINT* which stores the coordinates of a labelled point; *mark* is an integer which is not used in this paper (see Krishnamurti, 1981); and *next* satisfies the property *point*[*node*] < *point*[*next*[node]].

The last data class to be considered is *BTREE*. *BTREE* simulates height-balanced binary trees. Each node in *BTREE* consists of five fields.

tree-node | *B* | *left* | *right* | *key* | N |.

The fields have the following interpretation. B is the balance factor of a node in a balanced tree; *left* and *right* are pointers to nodes in *BTREE* and represent, respectively, the left and right offsprings of a node in a balanced tree; and *key* and N have the same interpretations as given earlier.

Each labelled shape σ is represented by the pair $\langle top_s, top_P \rangle$ where *top* is a pointer to a linked list or a pointer to a height-balanced tree. The subscript s refers to the data structure which maintains the maximal lines and the subscript P refers to the data structure which maintains the labelled points. A shape is described by the pair $\langle top_s,$ **null**$\rangle$ where **null** is a special pointer which points to an empty node. In a similar fashion, a labelled shape consisting of only labelled points is represented by the pair $\langle$**null**, $top_P\rangle$. Figure 14 illustrates the internal organization of a labelled shape in which its shape is housed in a balanced tree, and its set of labelled points is housed in a linked list. Notice that the data structures are arranged according to the data classes to which they belong with only sample links indicated.

Remark: It is assumed that these data structures can be implemented without violating or mixing data types. For instance, the data classes can be regarded as collections of integer arrays, and the fields may be equivalenced via some mechanism, say the Fortran EQUIVALENCE statement.

Algorithm

The data structures are incorporated into algorithms 1–4 which describe, in Algol-like notation, the algorithmic framework for decomposing the Boolean operations or relations on labelled shapes into (a) lists of shape operations or relations on lists of colinear maximal lines, or (b) lists of set operations or relations on lists of identically labelled points. The set of algorithms 1–4 is based on the modified Brown–Tarjan balanced tree search algorithm, and are called: SHAPE EXPRESSION, DELETE NODE, INSERT NODE, and ROTATE TREE. SHAPE EXPRESSION describes the algorithmic framework in which the algorithms for the shape expressions $\sigma_1 \leftarrow \sigma_1 \square \sigma_2$ in the case that $\square$ is an operator, and $\sigma_2 \square \sigma_1$ if $\square$ is otherwise, are invoked. The inputs are the labelled shapes σ_1 and σ_2; the shape operator or relation $\square$ which takes on values from $\{+, -, \cdot, \leq, =\}$; and a parameter c which takes on values from $\{s, P\}$. SHAPE EXPRESSION invokes algorithms DELETE NODE and INSERT NODE, respectively, wherever a node is deleted from or inserted into the balanced tree rooted at top_c (σ_1). DELETE NODE and INSERT NODE both invoke ROTATE TREE

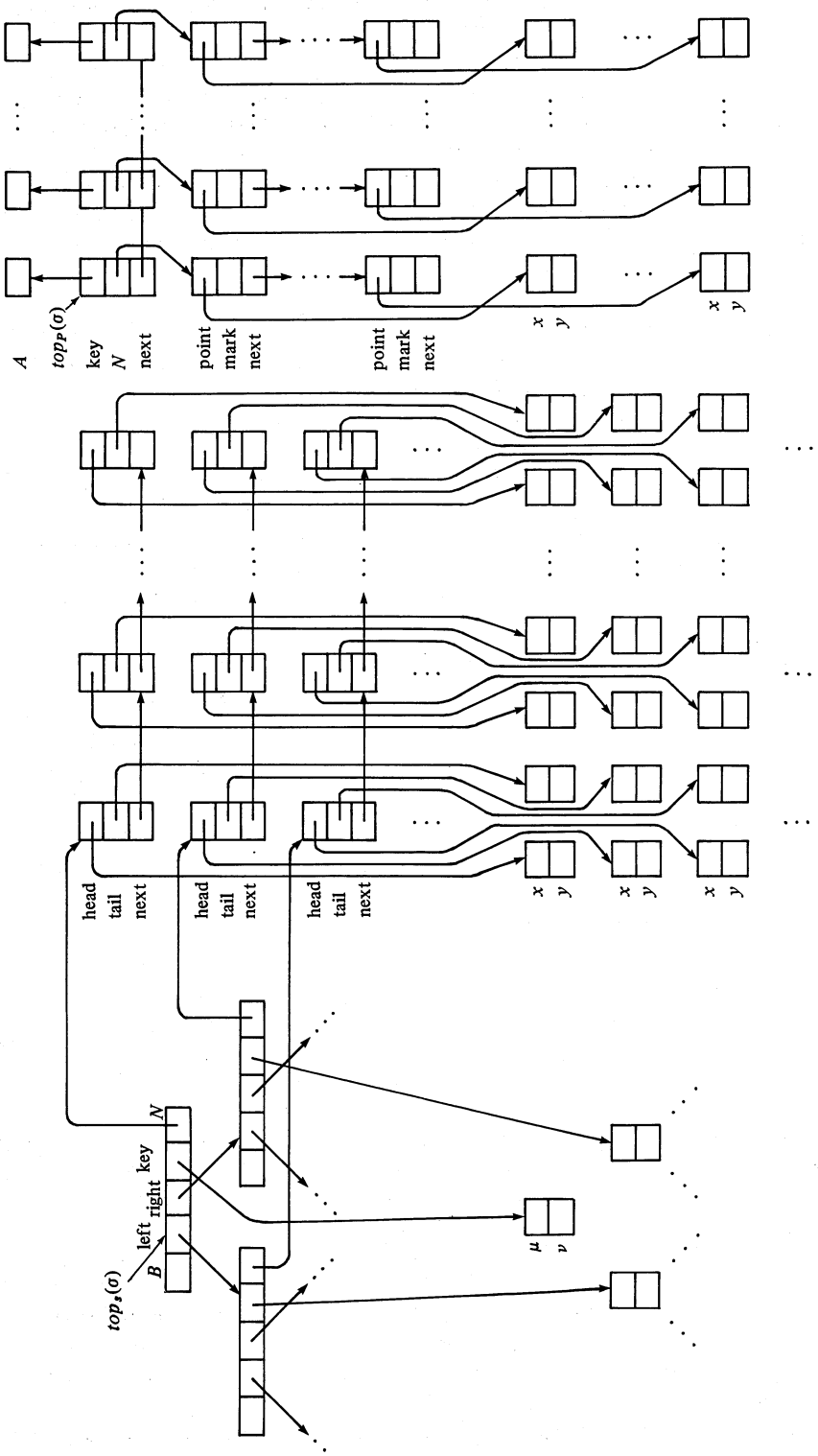


Figure 14. The internal organization of a labelled shape.

```
Algorithm SHAPE EXPRESSION (σ1, σ2, □, c)
¶ This routine describes the decomposition of the Boolean operations and relations based on the fast
  balanced tree search. The following shape expressions are considered as illustrative examples:
  σ1 ← σ1 □ σ2 for □ an operator, and σ2□σ1 for □ otherwise.
  σ1 is represented by the pair ⟨top_s(σ1), top_P(σ1)⟩, where top_c(σ1), c ∈ {s, P}, is the root of a
  balanced binary tree.
  The following 'conventions' are adopted:
  1. N_u denotes the list of elements represented by node u, the first element of which is pointed to
     by N[u].
  2. ψ(u) is the key associated with node u, and is given by: ⟨μ, ν⟩[key[u]] for c = s, and
     A[key[u]] for c otherwise.
  3. The expressions:
     node ⇐ DATA CLASS  fetches an unused (available) node in DATA CLASS, initializes all its
                        fields, and assigns it to node.
     DATA CLASS ⇐ node  releases the node pointed to by node, and makes it available for future
                        use ¶
¶ Initialization
  path:        an array containing all the nodes traversed from the root to the current node.
               For 1 ≤ i < pptr, path[i+1] is an offspring of path[i].
  successor:   an array containing all the nodes in path whose left offsprings are also in path.
               For 1 ≤ j ≤ sptr, path[successor[j]+1] = left[path[successor[j]]].
  pptr, sptr:  pointers to the last entry in path and successor respectively.
  flag:        a Boolean variable which signifies whether or not σ2 □ σ1 if □ is a relation, and
               σ1 □ σ2 = ∅ if □ is otherwise ¶
  path[pptr ← 1] ← top_c(σ1) ¶ σ1 is initially assumed nonempty ¶
sptr ← 0
flag ← true
¶ Balanced tree search algorithm
  GETNEXT(top_c(σ2)) is a global routine which fetches, on each invocation, the 'next' node, in
  order, in the data structure headed by top_c(σ2), c ∈ {s, P}.
  u, v: nodes representing the current lists under consideration in c(σ1) and c(σ2), c ∈ {s, P},
        respectively ¶
for v ← GETNEXT(top_c(σ2)) while (v ≠ null and flag)
do {
    ¶ Climb up phase. Retract the path until top of path, u, is either the root or satisfies
      ψ(u) ≤ ψ(v). It is only necessary to examine the nodes pointed to in successor ¶
    while (sptr > 0) and (ψ(v) > ψ[path[successor[sptr]]]) do { pptr ← successor[sptr]
                                                               { sptr − ← 1
    u ← path[pptr]
    ¶ Balanced tree search. Compare keys and extend the path ¶
    search ← true
    while search
    do {
        if ψ(u) = ψ(v)
        then {
            ¶ Matching keys found. Perform the appropriate action ¶
            search ← false
            case □ in
                (+, ·, −)  ¶ Shape or set operation ¶
                           N_u ← N_u □ N_v
                           if N_u = ∅ then { DELETE NODE
                                           { flag ← path[1] ≠ null
                (≤, =)     ¶ Shape or set relation ¶
                           flag ← N_v □ N_u
        }
        else {
            if ψ(v) < ψ(u)
            then {
                ¶ Branch left ¶
                successor[sptr + ← 1] ← pptr
                x ← left[u]
            }
            else {
                ¶ Branch right ¶
                x ← right[u]
            }
            if x = null
            then {
                ¶ There is no list N_u with the same key value as N_v ¶
                search ← false
                case □ in
                    (+)        ¶ Insert N_v into the tree ¶
                               x ⇐ BTREE
                               ψ(x) ← ψ(v)
                               N_x ← N_v
                               INSERT NODE
                    (·, −)     if successor[sptr] = pptr then sptr − ← 1
                    (≤, =)     flag ← false
            }
        }
    }
¶ Finishing touches ¶
case □ in
    (+, ·, −)  top_c(σ1) ← path[1]
               return
    (≤, =)     return(flag)
end SHAPE EXPRESSION
```

Algorithm 1.

```
Algorithm DELETE NODE
¶ This routine deletes node u from the tree and rebalances the tree ¶
if right[u] = null
then x ← left[u]
else { if left[u] = null
       then x ← right[u]
       else { ¶ u has nonempty subtrees. Delete, instead, a node from the higher subtree of u
              after copying its content into u so that the resulting tree still represents the altered
              list in its original list order ¶
              if B[u] = righttaller
              then { ¶ Let y be the immediate successor of u in the list ¶
                     path[pptr+ ← 1] ← y ← right[u]
                     while left[y] ≠ null do { successor[sptr+ ← 1] ← pptr
                                               path[pptr+ ← 1] ← y ← left[y]
                     x ← right[y] }
              else { ¶ Let y be the immediate predecessor of u in the list ¶
                     successor[sptr+ ← 1] ← pptr
                     path[pptr+ ← 1] ← y ← left[u]
                     while right[y] ≠ null do path[pptr+ ← 1] ← y ← right[y]
                     x ← left[y] }
       ¶ Copy contents of y into u and rename y ¶
       ψ[u] ← ψ[y]
       N[u] ← N[y]
       u ← y }
if c = s then POINT ⇐ key[u] else LABEL ⇐ key[u]
BTREE ⇐ u
pptr− ← 1
¶ x is either the root or becomes an offspring of the former parent of u ¶
if pptr = 0
then { path[pptr ← 1] ← x ¶ x equals null signifies an empty tree ¶
       sptr ← 0 }
else { ¶ Deleting a node from the right (left) subtree effectively makes the left(right) subtree
       taller ¶
       z ← path[pptr]
       if u = right[z]
       then { right[z] ← x
              a ← lefttaller }
       else { left[z] ← x
              a ← righttaller }
       ¶ Save the path pointer so as not to destroy path. Trace back along the path towards the
       root either adjusting the balance factors or rebalancing the tree ¶
       saveptr ← pptr
       while (saveptr > 0) and (B[z] ≠ balanced)
       do { if B[z] ≠ a
            then { ¶ Subtree rooted at z has become balanced ¶
                   B[t ← z] ← balanced
                   if (saveptr− ← 1) > 0 then a ← (if t = right[z ← path[saveptr]]
                                                   then lefttaller
                                                   else righttaller) }
            else { ¶ z has become doubly unbalanced. That is, the difference in the heights of the
                   right and left subtrees of z equals ±2. Rotate the subtree rooted at z and let
                   w be the new root ¶
                   y ← (if a = righttaller then right[z] else left[z])
                   ROTATE TREE
                   t ← z
                   path[pptr ← saveptr] ← w
                   if (saveptr− ← 1) > 0
                   then { if t = right[z ← path[saveptr]]
                          then { right[z] ← w
                                 a ← lefttaller }
                          else { left[z] ← w
                                 a ← righttaller } } }
       if saveptr > 0 then B[z] ← a
       ¶ Rebalancing may have corrupted the path. Delete the invalidated section ¶
       while (sptr > 0) and (successor[sptr] ≥ pptr) do sptr− ← 1
end DELETE NODE
```

Algorithm 2.

```
Algorithm INSERT NODE
¶ This routine inserts a new mode x as an offspring of u, and rebalances the tree ¶
if ψ(x) < ψ(u) then left[u] ← x else right[u] ← x
path[pptr + ← 1] ← x
¶ Save the path pointer so as not to destroy path. Trace back along the path towards the root
  adjusting the balance factors until the point of rebalancing is reached ¶
z ← path[saveptr ← pptr − 1]
y ← x
while (saveptr > 1) and (B[z] = balanced)
do { B[z] ← (if y = right[z] then righttaller else lefttaller)
   { y ← z
   { z ← path[saveptr − ← 1]
¶ z is either the root or the point of rebalancing. Determine in which subtree of z is x inserted ¶
a ← (if y = right[z] then righttaller else lefttaller)
¶ The following segment is a translation of steps 7-10 of algorithm 6.2.3A in Knuth (1973b) ¶
if B[z] ≠ a
then { ¶ The tree remains height-balanced ¶
     { B[z] + ← a
     { ¶ z is doubly unbalanced. That is, the difference in the heights of the right and left subtrees
       of z equals ±2. Rotate the subtree rooted at z, and let w be the root of the rotated
       subtree ¶
       ROTATE TREE
else { path[pptr ← saveptr] ← w
       if saveptr > 1 then { t ← path[saveptr − 1]
                           { if z = right[t] then right[t] ← w else left[t] ← w
       ¶ Rebalancing may have corrupted the path; delete the invalidated section ¶
       while (sptr > 0) and (successor[sptr] ⩾ pptr) do sptr − ← 1
end INSERT NODE
```

Algorithm 3.

```
Algorithm ROTATE TREE
¶ This routine rotates the subtree rooted at z. w contains the new root of the subtree. a, w, y,
  and z are global variables ¶
if B[y] = −a
     { ¶ Double rotation [figure 13(b)] ¶
       if a = righttaller
            { w ← left[y]
            { left[y] ← right[w]
       then { right[w] ← y
            { right[z] ← left[w]
            { left[w] ← z
then   { w ← right[y]
       { right[y] ← left[w]
       else { left[w] ← y
       { left[z] ← right[w]
       { right[w] ← z
       B[z] ← (if B[w] = a then −a else balanced)
       B[y] ← (if B[w] = −a then a else balanced)
       B[w] ← balanced
     { ¶ Single rotation [figure 13(a)] ¶
       w ← y
       if a = righttaller
            { right[z] ← left[y]
       then { left[y] ← z
else   { left[z] ← right[y]
       else { right[y] ← z
       B[y] ← (if B[w] = balanced then −(B[z] ← a) else B[z] ← balanced)
end ROTATE TREE
```

Algorithm 4.

which describes the two types of rebalancing transformation: single and double rotation which are shown in figure 13. GETNEXT which is not given here is a global routine for selecting the nodes in the data structure rooted at top_c (σ_2) in order.

Acknowledgements. I am indebted to George Stiny for his support and whose suggestions considerably improved the presentation of this paper. The research reported in this paper was supported in part by a grant from the Science Research Council.

References

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

Brown M R, Tarjan R E, 1979 "A fast merging algorithm" *Journal of the Association for Computing Machinery* **26**(2) 211-226

Crane C A, 1972 *Linear Lists and Priority Queues as Balanced Binary Trees* PhD thesis, technical report, STAN-CS-72-259, Computer Science Department, Stanford University, Stanford, Calif.

Horowitz E, Sahni S, 1976 *Fundamentals of Data Structure* (Computer Science Press, Woodland Hills, Calif.)

Knuth D E, 1973a *The Art of Computer Programming, Volume 1: Fundamental Algorithms* (Addison-Wesley, Reading, Mass)

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

Krishnamurti R, 1981 "The construction of shapes" *Environment and Planning B* **8** (forthcoming)

Stiny G, 1980 "Introduction to shapes and shape grammars" *Environment and Planning B* **7** 343-351

Wirth N, 1976 *Algorithms + Data Structure = Programs* (Prentice-Hall, Englewood Cliffs, NJ)