1979

Trees as data structures

Alfs T. Berztiss
University of Pittsburgh, uow@berztiss.edu.au

PREFACE

In these notes for a series of lectures given at the University of Wollongong in the Winter of 1979 emphasis is on the structural aspects of trees, i.e., there is little concern with properties related to the information content of trees. Hence nothing will be found here on, for example, optimization of weighted trees, B-trees, or decision trees as used in artificial intelligence work.

Some of the material is an adaptation of the auhtor's research results. This research was partly supported by the National Science Foundation of the United States under Grants GJ-41683 and MCS 77-01462.

A. T. Berztiss
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260, U.S.A.

TREES AS DATA STRUCTURES

1. TREES AND THEIR REPRESENTATION

1.1 Trees and binary trees

A connected acyclic graph $G = \langle V,E \rangle$, where V is its set of nodes and E its set of edges is a tree. Designate some $r \in V$ as the root of the tree. Then G becomes a rooted tree. If a node in a rooted tree is not the root of the tree and has just one node adjacent to it, then it is a leaf or terminal node; else it is an internal node. Every edge of a rooted tree lies on some path from the root to a leaf. If a direction is assigned to every edge, the tree becomes a directed tree. An edge with direction is an arc. The usual convention (and the one followed here) is to require that every arc in a directed tree be directed away from the root. In drawings of digraphs arcs are represented as arrows. This is rarely done in drawings of directed trees. Instead the direction is implicitly indicated by placing the tail of an arc above its head. Suppose several arcs have the same node as their tail. When the left-to-right order of the heads of these arcs is important we speak of an ordered directed tree. These latter structures have greater significance to computer science than the more general trees, and in what follows the term tree will be used as shorthand for ordered directed tree.

Consider node n in a tree. Nodes reachable from n are descendants of n, and nodes that are heads of arcs whose tail is n are successors of n. All nodes that have n as descendant are ancestors of n, and the ancestor nearest to n is the parent of n. Consider the nodes that have the same parent in their left-to-right order. The nearest node to the left (right) of node n is its left (right) neighbour.

A tree in which no more than k arcs originate from a node is a k-nary tree (binary when k=2, ternary when k=3). In a binary tree, whenever two arcs originate from a node, one of the arcs goes to the left, and the other to the right. However,

when there is just one arc originating from a node, it is still possible to impose a left or right orientation on this arc, and, if this is done, the binary tree becomes oriented. It is common practice to reserve the term binary tree for structures that would be more precisely called oriented binary trees.

The level of a node in a tree is the length of the path (i.e., the number of arcs on the path) from the root to this node. The level of the root is thus zero. The height of a tree is the greatest level assignment given to a node in the tree, i.e., it is the length of the longest path from the root to a terminal node.

Level i of a binary tree is full if there are exactly 2^i nodes at this level. A binary tree of height k is full if level k in this binary tree is full -- level k being full implie of course, that levels k-1,...,1,0 are all full. A binary tree of height k is balanced if level k-1 in this binary tree is full Consider node n in a binary tree. The binary subtree rooted at the left successor of n is the left subtree of n, and that rooted at the right successor is the right subtree. A binary tree is height balanced (or an AVL-tree, after Adelson-Velskii and Landis) if, at every node in the tree, its left and right subtrees differ by no more than 1 in height (for the purposes of this definition a tree that does not exist has height -1 -- for example, if a node has no left successor, and its right successor is a leaf, the heights of its left and right subtrees are -1 and 0, respectively).

Every tree can be transformed into a binary tree by means of the following procedure, which has become known as the Knuth transformation:

> Denote the arcs originating at internal node x in a tree by $\langle x,y_1\rangle$, $\langle x,y_2\rangle$,..., $\langle x,y_t\rangle$. Assign left orientation to arc $\langle x,y_1\rangle$, and replace the remaining arcs by arcs $\langle y_1,y_2\rangle$, $\langle y_2,y_3\rangle$,..., $\langle y_{t-1},y_t\rangle$, all with right orientation. After this has been done for all internal nodes of the tree the result is a binary tree that contains the same number of arcs as the original tree.

Note that the transformation is reversible, i.e., that a tree

an be reconstructed from its Knuth transform. Figures 1 and 2 how a tree and its Knuth transform, respectively.

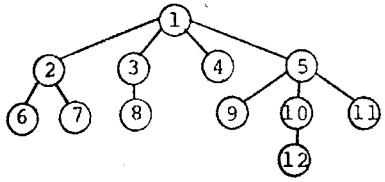

Figure 1

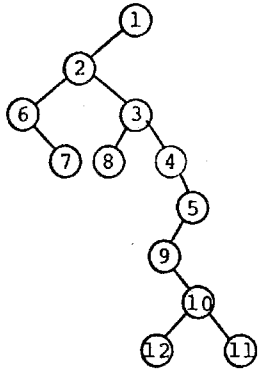

Figure 2

.2 Traversals of trees

The three classical disciplines of traversal of binary rees -- preorder, inorder (or symmetric order), postorder -- re well known and well understood. For the binary tree of ig. 2 they generate the following sequences:

pre	1	2	6	7	3	8	4	5	9	10	12	11
in	6	7	2	8	3	4	9	12	10	11	5	1
post	7	6	8	12	11	10	9	5	4	3	2	1

(1)

Now extend the definitions of preorder and postorder raversals to arbitrary trees. Preorder traversal: At each node n the tree, process the node and then proceed to the successors f the node in left-to-right order. Postorder traversal: At each ode, proceed to the successors of the node in left-to-right rder, and then process the node. For the tree of Fig. 1 we have

T-pre	1	2	6	7	3	8	4	5	9	10	12	11
T-post	6	7	2	8	3	4	9	12	10	11	5	1

(2)

omparison of (2) with (1) shows that preorder traversal is

when there is just one arc originating from a node, it is still possible to impose a left or right orientation on this arc, and, if this is done, the binary tree becomes oriented. It is common practice to reserve the term binary tree for structures that would be more precisely called oriented binary trees.

The level of a node in a tree is the length of the path (i.e., the number of arcs on the path) from the root to this node. The level of the root is thus zero. The height of a tree is the greatest level assignment given to a node in the tree, i.e., it is the length of the longest path from the root to a terminal node.

Level i of a binary tree is full if there are exactly 2^i nodes at this level. A binary tree of height k is full if level k in this binary tree is full -- level k being full implies, of course, that levels k-1,...,1,0 are all full. A binary tree of height k is balanced if level k-1 in this binary tree is full. Consider node n in a binary tree. The binary subtree rooted at the left successor of n is the left subtree of n, and that rooted at the right successor is the right subtree. A binary tree is height balanced (or an AVL-tree, after Adelson-Velskii and Landis) if, at every node in the tree, its left and right subtrees differ by no more than 1 in height (for the purposes of this definition a tree that does not exist has height -1 -- for example, if a node has no left successor, and its right successor is a leaf, the heights of its left and right subtrees are -1 and 0, respectively).

Every tree can be transformed into a binary tree by means of the following procedure, which has become known as the Knuth transformation:

> Denote the arcs originating at internal node x in a tree by $\langle x,y_1\rangle$, $\langle x,y_2\rangle$,..., $\langle x,y_t\rangle$. Assign left orientation to arc $\langle x,y_1\rangle$, and replace the remaining arcs by arcs $\langle y_1,y_2\rangle$,$\langle y_2,y_3\rangle$,...,$\langle y_{t-1},y_t\rangle$, all with right orientation. After this has been done for all internal nodes of the tree the result is a binary tree that contains the same number of arcs as the original tree.

Note that the transformation is reversible, i.e., that a tree

can be reconstructed from its Knuth transform. Figures 1 and 2 show a tree and its Knuth transform, respectively.

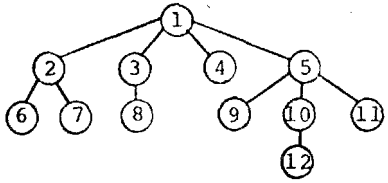

Figure 1

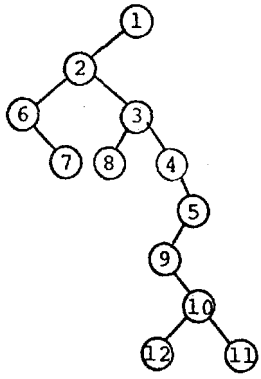

Figure 2

1.2 Traversals of trees

The three classical disciplines of traversal of binary trees -- preorder, inorder (or symmetric order), postorder -- are well known and well understood. For the binary tree of Fig. 2 they generate the following sequences:

pre	1	2	6	7	3	8	4	5	9	10	12	11
in	6	7	2	8	3	4	9	12	10	11	5	1
post	7	6	8	12	11	10	9	5	4	3	2	1

(1)

Now extend the definitions of preorder and postorder traversals to arbitrary trees. Preorder traversal: At each node in the tree, process the node and then proceed to the successors of the node in left-to-right order. Postorder traversal: At each node, proceed to the successors of the node in left-to-right order, and then process the node. For the tree of Fig. 1 we have

T-pre	1	2	6	7	3	8	4	5	9	10	12	11
T-post	6	7	2	8	3	4	9	12	10	11	5	1

(2)

Comparison of (2) with (1) shows that preorder traversal is

invariant under the Knuth transformation, but that postorder traversal of a general tree is equivalent to inorder traversal of its transform. This observation permits us to advance an alternative definition of preorder and postorder traversals of trees. Preorder (postorder) traversal: Process nodes in the order they would be processed in the Knuth transform of the tree under preorder (inorder) traversal.

Preorder traversal is also known as depth-first traversal. If we wished to explore a tree in a breadth-first manner, we would be employing breadth-first or levelorder traversals. A traversal whose direction is from left to right will be designated to be of type LR, and one whose direction is right to left to be of type RL. We then have, for the binary tree of Fig. 2,

LR-level	1	2	6	3	7	8	4	5	9	10	12	11	(3)
RL-level	1	2	3	6	4	8	7	5	9	10	11	12	

and for the general tree of Fig. 1,

T-LR-level	1	2	3	4	5	6	7	8	9	10	11	12	(4)
T-RL-level	1	5	4	3	2	11	10	9	8	7	6	12	

It is of interest to determine what levelorder traversals (4) become in the transform. In the equivalent to LR-levelorder traversal of a general tree we are moving down diagonally from upper left to lower right (LR-down). In the equivalent to RL-levelorder we move from lower right to upper left (RL-up). So, for the binary tree of Fig. 2 we obtain two new traversal sequences:

LR-down	1	2	3	4	5	6	7	8	9	10	11	12	(5)
RL-up	1	5	4	3	2	11	10	9	8	7	6	12	

Let us now introduce two operators on traversals, R (for reverse) and C (for converse). Under R(t) nodes of a binary tree are processed in an order that is the exact opposite of the order in which they are processed under traversal t, and C(t) is

equivalent to traversal t in a binary tree that has been flipped over, thus interchanging the meaning of left and right. For the binary tree of Fig. 2 we now have

R(pre)	11	12	10	9	5	4	8	3	7	6	2	1
R(in)	1	5	11	10	12	9	4	3	8	2	7	6
R(post)	1	2	3	4	5	9	10	11	12	8	6	7

(6)

C(pre)	1	2	3	4	5	9	10	11	12	8	6	7
C(in)	1	5	11	10	12	9	4	3	8	2	7	6
C(post)	11	12	10	9	5	4	8	3	7	6	2	1

(7)

Note that

$$\begin{aligned} R(pre) &= C(post) \\ R(in) &= C(in) \\ R(post) &= C(pre) \end{aligned} \tag{8}$$

Indeed, we can relate the three classical traversal disciplines and their converses by the diagrams of Fig. 3.

Now, if we apply the C and R operators to LR-level, LR-down, RL-up, we obtain in each instance three new traversals, as shown in Fig. 4. These nine new traversals are distinct from the nine traversals defined earlier, as can be seen by comparison of (1), (7), (3), and (5) with (9) below, where we give the sequences obtained when the nine traversals are applied to the binary tree of Fig. 2.

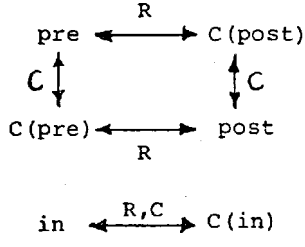

Figure 3

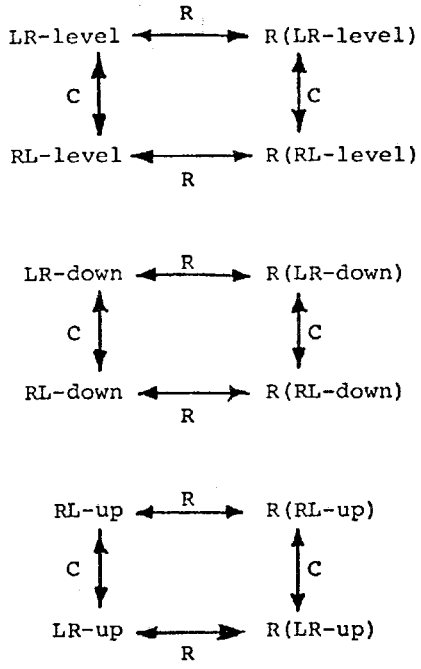

Figure 4

RL-level	1	2	3	6	4	8	7	5	9	10	11	12	
R(LR-level)	11	12	10	9	5	4	8	7	3	6	2	1	
R(RL-level)	12	11	10	9	5	7	8	4	6	3	2	1	
RL-down	1	2	6	3	8	7	4	5	9	10	12	11	
R(LR-down)	12	11	10	9	8	7	6	5	4	3	2	1	(9)
R(RL-down)	11	12	10	9	5	4	7	8	3	6	2	1	
LR-up	6	2	1	7	8	3	4	9	5	12	10	11	
R(RL-up)	12	6	7	8	9	10	11	2	3	4	5	1	
R(LR-up)	11	10	12	5	9	4	3	8	7	1	2	6	

1.3 Algorithms for binary tree traversals

The classical traversals and their reverses (or converses) correspond to the six permutations of statements X,Y,Z in the schema

```
procedure traverse (root);
begin {X,Y,Z} end;
```
(10)

where

```
X:  process node;
Y:  if left exists then traverse ( left );
Z:  if right exists then traverse (right);
```

The ordering of X,Y,Z in (10) defines the traversals as follows:

pre	XYZ	R(pre)	ZYX	C(pre)	XZY
in	YXZ	R(in)	ZXY	C(in)	ZXY
post	YZX	R(post)	XZY	C(post)	ZYX

Note that the order of the statements for R(t) is exactly the reverse of that for traversal t, but that C(t) is obtained from t by interchanging Y and Z alone.

Let us now remove recursion from schema (10) by the introduction of an explicit pushdown store.

```
procedure traverse (binary tree);
begin PD1 := ∅ ;
      PD1 ← root;
      while PD1 ≠ ∅ do
      begin node ← PD1;
            if node marked then process node
            else begin {A,B,C} end
      end
end;
```

where

```
A:  mark node; PD1 ← node;
B:  if left exists then PD1 ← left;
C:  if right exists then PD1 ← right;
```

Now, pushdown store PD1 may be either a stack or a queue. If it is a stack then the permutations of A,B,C still define the three classical traversals and their converses (or reverses), as follows:

pre	CBA	C(pre)	BCA
in	CAB	C(in)	BAC
post	ACB	C(post)	ABC

If PD1 is a queue then the nodes of the binary tree are processed in RL-levelorder irrespective of whether the order of A,B,C is CBA, CAB, or ACB. Similarly, LR-levelorder is obtained with permutations BCA, BAC, ABC. In other words, the three classical orders all map to RL-levelorder, and their converses (or reverses) to LR-levelorder.

Under schema (11) each node of the binary tree is pushed down twice. It is possible to refine (11) specifically into a procedure for preorder traversal in which each node is pushed down only once.

```
procedure preorder1  (bitree);
begin  PD1 := ∅ ;
       PD1 ← root;
       repeat node ← PD1;
              process node;
              if right exists then PD1 ← right;
              if left exists then PD1 ← left
       until PD1 = ∅
end;
```

Procedure preorder1 still processes the nodes of a binary tree in RL-levelorder when PD1 is a queue.

Let us refine the procedure still further. We note that when "PD1 ← left" is executed its execution is followed immediately by "node ← PD1", and we take advantage of this.

```
procedure preorder2 (bitree);
begin PD1: = ∅;
      PD1 ← root;
      repeat node ← PD1;
             noleft: = false;
             repeat process node;
```

```
                    if right exists then PD1 ← right;
                    if left exists then  node: = left
                    else noleft: = true
                until noleft
            until PD1 = ∅
        end;
```

Substitution of queue for stack in preorder2 produces a procedure for RL-downorder.

In the next modification, instead of processing nodes directly as they are taken off PD1 or reached in tracing left successor chains in the inner repeat-until loop, we save them up in a second pushdown store. This modification cannot be called a refinement because there is now a substantial increase in cost over that of preorder2. In addition to the activities associated with PD1, which is utilized in exactly the same manner as in preorder2, now every node gets pushed down exactly once on the second pushdown store.

```
procedure preorder3 (bitree);
begin  PD1: = ∅;  PD2: = ∅;
       PD1 ← root;
       repeat node ← PD1;
              noleft: = false;
              repeat  PD2 ← node;
                      if right exists then PD1 ← right;
                      if left exists then node: = left
                      else noleft: = true
              until   noleft;
              repeat  node ← PD2;
                      process node
              until   PD2 = ∅
       until PD1 = ∅
end;
```

The behaviour of preorder3 is expressed in the following table:

PD1	PD2	Traversal
stack	queue	preorder
stack	stack	LR-uporder
queue	queue	RL-downorder
queue	stack	(nonstandard)

Let us summarize our results on traversals of binary trees. We have defined 18 distinct and meaningful traversal disciplines, and shown that they can all be derived from five basic disciplines by means of operators R and C. It should be noted that the two operators are fundamentally different: C operates on an algorithm (it interchanges the meaning of left and right); R operates on the node sequence produced by an algorithm. Only in the special case of the schemas for the classical traversals, e.g. (11), does R too function as an operator on an algorithm. We shall designate the following five traversals as our set of basic traversals:

preorder
inorder
RL-levelorder
RL-downorder
LR-uporder

Fig. 5 shows how the procedures for generating these traversals are related.

1.4 Storage representations of trees

A tree can be represented by a set of arc lists, one for each internal node. For example, the tree of Fig.1 has the following arc list representation:

```
 1:  2,3,4,5
 2:  6,7
 3:  8                                   (12)
 5:  9,10,11
10:  12
```

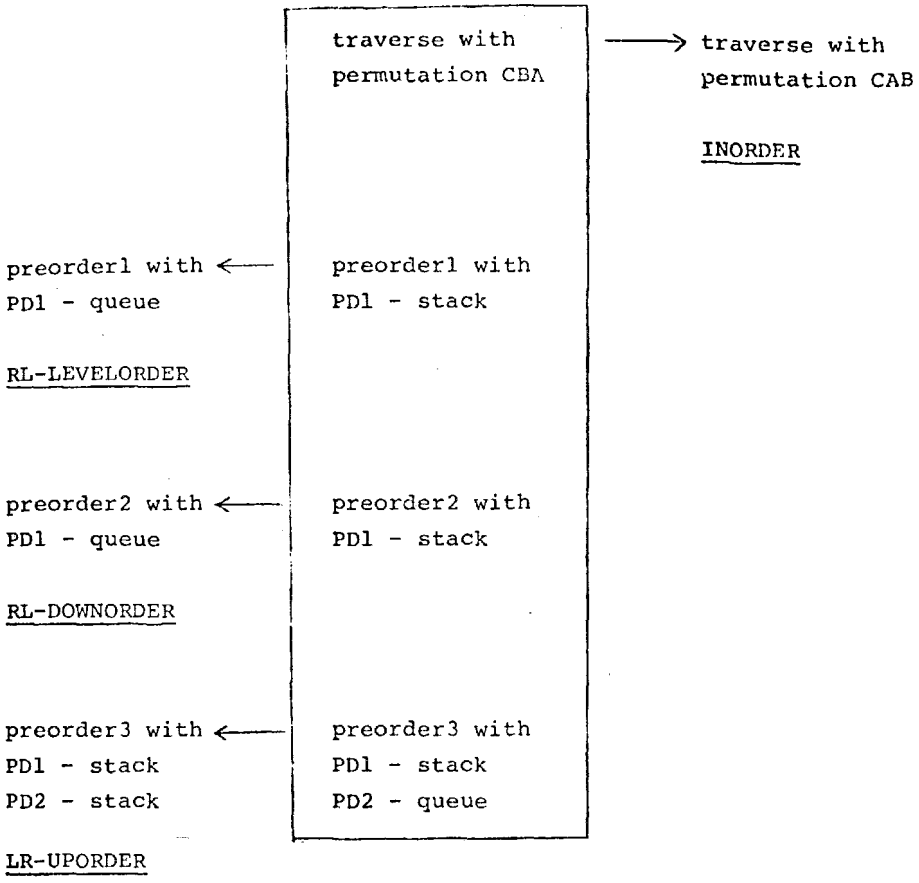
traverse with
permutation CBA
traverse with
permutation CAB
INORDER
preorder1 with
PD1 - queue
RL-LEVELORDER
preorder1 with
PD1 - stack
preorder2 with
PD1 - queue
RL-DOWNORDER
preorder2 with
PD1 - stack
preorder3 with
PD1 - stack
PD2 - stack
LR-UPORDER
preorder3 with
PD1 - stack
PD2 - queue
PREORDER

Figure 5

The unequal lengths of the arc lists create some problems in the design of a space-efficient storage structure. Also, in many applications one has to provide for addition and deletion of arcs, and the storage structure should permit this to be done reasonably efficiently.

For a binary tree each arc list contains at most two entries. If each arc list is made to have exactly two elements, then it is possible to indicate orientation of an arc: the head of an arc with left orientation is stored in the first element, the head of an arc with right orientation in the second element. This uniformity in the representation of binary trees is one reason for mapping arbitrary trees to binary trees by means of the Knuth transformation. The only disadvantage in operating with the transform is that path lengths in the transform are greater than in the original tree. For example, the length of the path from node 1 to node 11 is only 2 in the tree of Fig. 1, but 7 in the transform (Fig.2).

We have, for the binary tree of Fig.2:

```
 1:  2, -
 2:  6, 3
 3:  8, 4
 4:  -, 5
 5:  9, -
 6:  -, 7
 9:  -, 10
10: 12,11
```

(13)

It is customary to provide arc lists for terminal nodes as well, although both elements in such lists are empty. Thus, if the nodes are numbered 1,2,...,n, the set of arc lists can be stored as an array of n rows and two columns. Names of arc lists are now provided implicity by the row subscripts.

This approach has the further advantage that the empty locations can be used to hold threads, which can be very useful in the implementation of some iterative traversals of the binary tree that do not then require a pushdown store. For example, if the second linkage element in the row for node n is empty, i.e., if n has no right successor, then this element can be made to hold the node that follows node n under

inorder traversal, i.e., it is then a thread from n to this node, called the right thread. Similarly, if the first linkage element is empty, it can be made to hold a left thread from n to the node that precedes it in the inorder sense. A binary tree with right threads is a right threaded binary tree, one with both right and left threads is a fully threaded binary tree. Fig. 6 shows the fully threaded representation of the binary tree of Fig. 2, where the threads are distinguished from successor links by flags (*). Since the inorder sequence begins at 6 and ends at 1, there is no left thread for node 6 and no right thread for node 1.

```
 1:  2   -
 2:  6   3
 3:  8   4
 4:  3*  5
 5:  9   1*
 6:  -   7
 7:  6*  2*
 8:  2*  3*
 9:  4* 10
10: 12  11
11: 10*  5*
12:  9* 10*
```

Figure 6

An algorithm for inorder traversal of a right threaded binary tree:

```
procedure inorder (bitree);
begin node: = root;
      alldone: = false;
      repeat if left exists∧node not reached by thread then
                  node: = left
             else begin process node;
                        if right exists then node: = right
```

```
                    else if thread exists then node: =
                         else alldone: = true
                end
        until alldone
    end;
```

A similar algorithm based on right threads alone can be defined for preorder traversal. The algorithm for postorder traversal is more complicated and requires full threading. Again if there exists an algorithm for traversal t, the algorithm for C(t) is obtained by simply interchanging left and right in the algorithm t. This means, of course, that left threads have to be provided before the modified algorithm for inorder traversal can be applied to a binary tree for C(inorder).

By means of right threads one can easily determine in the Knuth transform of tree T the parent of node n with respect to

```
node: = n;
while right exists do node: = right;
parent: = thread;
```

On occasion, in case the tree is stored on disk, it may be convenient to determine first whether or not a node has a particular successor rather than try to retrieve the (possibly nonexistant) successor. This is achieved by means of bit vectors where for the binary tree of Fig. 2 we have:

1	2	3	4	5	6	7	8	9	10	11	12
1	1	1	0	1	0	0	0	0	1	0	0
0	1	1	1	0	1	0	0	1	1	0	0

(14)

A very dense binary tree, i.e., one close to being full, can be represented by a vector tree. Consider the binary tree of Fig.7. Its height is 3. Take a vector of $15(=2^{3+1}-1)$ elements. Store the root in tree[1]. Then for each node already stored in the array, if the node is stored at tree[k], store its left and right successors at tree[2*k] and tree[2*k+1] respectively.

The result here is

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
9	8	6	4	-	7	3	2	1	-	-	-	5	-	-

(15)

Additions and deletions can be handled quite easily, provided they do not change the height of the binary tree upward.

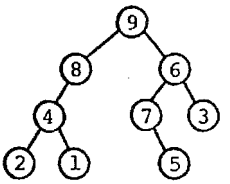

Figure 7

2. SEARCHING, SORTING, AND TREES

2.1 Heaps and binary search trees

Trees are often used to express relationships between records in a file. The records are regarded as occupying the nodes of a tree, and the relative location of two nodes in the tree implies some relationship between the keys of the records associated with these nodes. Consider node numbers 1,2,...,9 in the binary tree of Fig. 7 as keys. Here the tree has the heap property, which means that for any node the key at this node is larger than the keys at its successor nodes. Consequent the largest key in the entire file is at the root of the tree, and the second largest is at one of the successors of the root. The significance of this is that a binary tree with the heap property functions as a priority queue, i.e., irrespective of the changes that it may undergo, it provides immediate access to the largest of the keys stored in it at any one time.

If a binary tree that has the heap property is balanced, then it can be stored very compactly, and it is easy to provide O (log n) algorithms for addition and deletion of nodes that maintain both the heap property and balance. Using the storage concept expressed by (15), the following array (to which we give the name heap) corresponds to the binary tree of Fig. 8.

i	1	2	3	4	5	6	7	8	9
heap[i]	9	8	4	7	5	1	2	3	6

(16)

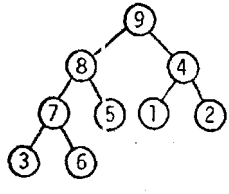

Figure 8

Suppose exactly one node k in a binary tree of n nodes is responsible for the heap property not holding. This node may either have to be bubbled up until the heap property is restored, or it may have to be shifted down in the tree. The following procedures perform these two tasks.

```
procedure bubbleup (heap, k);
begin  ordered: = false;
       repeat parent: = k div 2;
              if parent = 0 then ordered: = true
              else if heap[k] < heap[parent] then ordered:= true
                   else begin swap heap[k] and heap[parent];
                              k: = parent
                        end
       until ordered
end;
```

The procedure for shifting down is slightly more complicated because now it has to be first established which of the two possible successors of the node is the greater, because, in case a swap is required, it is with this successor. The complications arise out of the need to deal with cases in which the node has no successors or just one successor.

```
procedure siftdown (heap, k, n);
begin ordered: = false;
      repeat next: = 2*k;
             if next > n then ordered: = true
             else
             begin if next = n then hi: = next
                   else if heap[next] > heap[next+1] then hi:=next
                        else hi: = next+1;
                   if heap[k] > heap[hi] then ordered:=true
                   else swap heap[k] and heap[hi];
                   k: = hi
             end
      until ordered
end;
```

With these procedures the basic heap operations become very easy to implement.

(i) Building a heap of n elements.

```
for i: = 1 to n do
begin read (datum);
      heap[i]: = datum;
      bubbleup (heap,i)
end;
```

(ii) Deletion of an element. Suppose element heap[j] is to be deleted. This is effected by moving heap[n] into the vacated location, and then restoring the heap property to the remaining n-1 elements.

```
temp: = heap[j];
heap[j]: = heap[n];
if heap[j]> temp then bubbleup (heap,j)
                 else siftdown (heap,j,n-1);
```

The creation of the heap is the first phase of an O(n log n) sorting procedure known as heapsort or treesort. The second phase is extremely simple:

```
for i: = n downto 2 do
begin swap heap[1] and heap[i];
      siftdown (heap,1,i-1)
end;
```

At the end of the k-th iteration the k largest keys are in their final positions, and, since the tree of the remaining n-k keys possesses the heap property, the largest of these keys is in heap[1], ready to be shifted into heap[n-k]at the start of the next iteration.

The search tree property of binary trees is similar to the tree property: for each node, the key at the left successor is smaller and the key at the right successor larger than the key at the node. Keys 1,2,...,9 have to be assigned to the binary

tree of Fig. 8 in the manner of Fig. 9 for this tree to have the search tree property.

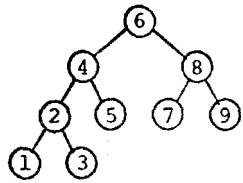

Figure 9

For a given tree shape and a given set of keys there is just one search tree, but there may be more than one heap. However, for a given set of n keys one can construct search trees of $\binom{2n}{n}/(n+1)$ distinct shapes. The numbers $\binom{2n}{n}/(n+1)$ are known as Catalan numbers or Segner numbers. When any one of these search trees is traversed under inorder, the result is a sorted sequence of the keys -- here the interpretation of "process node" is "append the key associated with the node to the sequence of keys being generated".

The height of a balanced binary tree is O(log n). Hence the cost of retrieving an element or inserting a new element in a balanced binary search tree is at most O(log n). However, insertion may require subsequent reorganization of the tree. Note now that the height of an AVL tree is also O(log n), and for binary search trees that have the AVL property it is relatively easy to show that a new node can be inserted or an existing node deleted, and the tree reorganized so that the AVL property is preserved, in time bounded by O(log n). This means that an AVL search tree of n nodes can be grown in time O(n log n), and that consequently an O(n log n) sorting procedure can be based on the search tree as well.

Further in this chapter the retrieval properties of balanced, height balanced, and arbitrary binary search trees will be compared. This analysis will be based on the solution of recurrence equations, and the next three sections are devoted to this topic, which is of major importance in the analysis of algorithms.

2.2 Recurrence relations with constant coefficients

The expression

$$c_1 a_n + c_2 a_{n-1} + \cdots + c_k a_{n-k+1} = f(n), \qquad (17)$$

where all c_i are constants is a k-term linear recurrence relation with constant coefficients. Suppose (17) is used to compute a sequence $[a_1, a_2, a_3, \ldots]$. Since it computes a_n from its k-1 predecessors, the initial k-1 members of the sequence must be known before any computing can begin. These values are called the boundary values. Here we need boundary values $a_1, a_2, \ldots, a_{k-1}$ to compute a_k. We shall be interested in solving recurrence relations rather than in using them to generate sequences, but for a fully determined solution of a k-term relation we shall still need k-1 boundary values. Recurrence relations are often called just recurrences, or they are called difference equations.

Examples:

1. The number of moves required to solve the Towers of Hanoi problem is defined by the two-term recurrence

$$a_n - 2a_{n-1} = 1, \quad \text{with} \quad a_1 = 1. \qquad (18)$$

2. The Fibonacci sequence $[1, 1, 2, 3, 5, 8, 13, 21, \ldots]$ is defined by the 3-term recurrence relation

$$a_n - a_{n-1} - a_{n-2} = 0, \quad \text{with} \quad a_1 = a_2 = 1. \qquad (19)$$

3. Let a_n be the max. number of pieces into which a pancake can be sliced with n cuts. These pancake numbers are defined by

$$a_n - a_{n-1} = n \quad (n \geq 1), \quad \text{with} \quad a_0 = 1 \qquad (20)$$

Let us try to solve from first principles the Fibonacci recurrence (19), which we rewrite as

$$a_{n+2} - a_{n+1} - a_n = 0. \qquad (19a)$$

Try different forms for a_n.

a. If we set $a_n = cn+d$, where c and d are constants, then we obtain $a_{n+1} = c(n+1)+d$ and $a_{n+2} = c(n+2)+d$, and, on substitution into (19a) and simplification, we obtain

$$-cn + c - d = 0,$$

which leads to the contradiction $n = (c-d)/c$.

b. Similar contradictions arise with any polynomial form for a_n.

c. Try $a_n = d^n$. Then

$$\begin{aligned} 0 &= d^{n+2} - d^{n+1} - d^n \\ &= d^n(d^2 - d - 1). \end{aligned}$$

Hence $d^2 - d - 1 = 0$, giving $d = (1 \pm \sqrt{5})/2$. In the most general form the solution is

$$a_n = c_1 \left(\frac{1+\sqrt{5}}{2}\right)^n + c_2\left(\frac{1-\sqrt{5}}{2}\right)^n \qquad (21)$$

Now make use of boundary values $a_1 = a_2 = 1$. Solving the simultaneous equations

$$c_1\left(\frac{1+\sqrt{5}}{2}\right) + c_2\left(\frac{1-\sqrt{5}}{2}\right) = 1,$$

$$c_1\left(\frac{1+\sqrt{5}}{2}\right)^2 + c_2\left(\frac{1-\sqrt{5}}{2}\right)^2 = 1,$$

gives $c_1 = 1/\sqrt{5}$, $c_2 = -1/\sqrt{5}$. Thus

$$a_n = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right] \qquad (22)$$

Specification of (21) did already require a certain measure of sophistication. Let us now express it as a systematic procedure for solving linear recurrence relations with constant coefficients. Equation (17) with the right hand side set to zero is known as a homogeneous equation, and we start by finding the solution to this equation. This solution is called the

homogeneous solution $a_n^{(h)}$, and in the theory of recurrence equations it is established that a homogeneous linear recurrence relation has a solution of the form

$$a_n^{(h)} = A\alpha^n. \tag{23}$$

Taking (17) with the right hand side set to zero, and making substitution (23), we obtain

$$c_1 A\alpha^n + c_2 A\alpha^{n-1} + \ldots + c_k A\alpha^{n-k+1} = 0,$$

which simplifies to

$$c_1\alpha^{k-1} + c_2\alpha^{k-2} + \ldots + c_k = 0. \tag{24}$$

This is the characteristic equation of the recurrence relation. The roots of (24) are called characteristic roots, and (24) has k-1 such roots, which need not all be real or distinct. Our discussion will be restricted to the case of distinct real roots. Let the characteristic roots be denoted $\alpha_1,\alpha_2,\ldots,\alpha_{k-1}$. Then $a_n = A_1\alpha_1^n$ satisfies the homogeneous equation with any A_1, and so does $a_n = A_2\alpha_2^n$ with any A_2, etc. A deterministic homogeneous solution must contain all the characteristic roots. It is

$$a_n^{(h)} = A_1\alpha_1^n + A_2\alpha_2^n + \ldots + A_{k-1}\alpha_{k-1}^n, \tag{25}$$

in which the constants are determined by means of the boundary values.

Example: The Fibonacci recurrence is homogeneous, and the characteristic equation is $\alpha^2 - \alpha - 1 = 0$. It has the two distinct roots $\alpha_1 = (1+\sqrt{5})/2$, $\alpha_2 = (1-\sqrt{5})/2$. Hence

$$a_n = a_n^{(h)} = A_1\left(\frac{1+\sqrt{5}}{2}\right)^n + A_2\left(\frac{1-\sqrt{5}}{2}\right)^n \tag{26}$$

and the constants are found as before.

When the right hand side of (17) is not zero, then solution (25) is incomplete in that it reduces the left hand side to zero, which then does not correspond to the nonzero right hand side. The homogeneous equation has to augmented by a <u>particular solution</u> $a_n^{(p)}$, and the total solution is now

$$a_n = a_n^{(h)} + a_n^{(p)} \quad . \tag{27}$$

We expect $a_n^{(p)}$ to have the same form as the right hand side of (17).

Examples:

1. The homogeneous solution of (18) is $A2^n$. For the particular solution try the constant B. The substitution $B-2B = 1$ gives $B = -1$. Hence the total solution is $A2^n-1$, and initial value $a_1 = 1$ establishes $A = 1$. WARNING: Constants A_i are <u>not</u> to be evaluated before the total solution has been derived. Here, if $a_1=1$ had been used in $a_n^{(h)} = A2^n$, we would have obtained $A=\frac{1}{2}$.

2. The most skewed AVL tree of height n has the two subtrees of every internal node differing in height by exactly 1. The number of nodes in such a tree obeys the recurrence

$$a_{n+2} - a_{n+1} - a_n = 1 \ , \tag{28}$$

with boundary values $a_o=1$, $a_1=2$. We already have $a_n^{(h)}$ -- it is given by (26). Try $a_n^{(p)} = k$. Substitution into (28) gives $a_n^{(p)} = -1$, and we get

$$a_n = A_1\left(\frac{1+\sqrt{5}}{2}\right)^n + A_2\left(\frac{1-\sqrt{5}}{2}\right)^n - 1. \tag{29}$$

Here we obtain $A_1 = 1 + 2/\sqrt{5}$, $A_2 = 1-2/\sqrt{5}$.

3. The characteristic equation of $a_n + a_{n-1} - 2a_{n-2} = 2^n$ is $\alpha^2 + \alpha - 2 = 0$, with solutions $\alpha_1 = -2$, $\alpha_2 = 1$. Hence

$a_n^{(h)} = A_1 (-2)^n + A_2$. For the particular solution try $B2^n$. Then $B2^n + B2^{n-1} - 2B2^{n-2} = 2^n$, or $2B + B - B = 2$, giving $B=1$. Hence $a_n = (-2)^n A_1 + A_2 + 2^n$, where A_1 and A_2 are determined from boundary values.

4. The characteristic equation of the pancake recurrence $a_n - a_{n-1} = n$ is $\alpha - 1 = 0$, and hence $a_n^{(h)} = A$. However, when we try $a_n^{(p)} = Bn$, we get $Bn - B(n-1) = n$, which results in the contradiction $B=n$. Hence we try next $a_n^{(p)} = Bn^2 + Cn$. Substitution of this into $a_n - a_{n-1} = n$ and $a_{n-1} - a_{n-2} = n-1$ gives two simultaneous equations in B and C, which have the solution $B = C = \frac{1}{2}$. The boundary value $a_o=1$ next establishes that $A=1$. Hence $a_n = 1 + \frac{1}{2}n(n+1)$.

2.3 Generating functions in the solution of recurrences

We just saw in the case of the pancake recurrence that the particular solution does not always have the form of the right hand side of (17). The use of generating functions simplifies the solution of (17) in that it permits us to find the total solution without us having to consider its two components separat

Generating functions are used extensively in combinatorics because they make sequences of numbers amenable to mathematical analysis. Suppose we have the sequence $[a_o, a_1, a_2, \ldots]$. Instead of manipulating the sequence we operate instead on

$$F(x) = a_o f_o(x) + a_1 f_1(x) + a_2 f_2(x) + \ldots, \qquad (30)$$

the generating function of sequence $[a_o, a_1, a_2, \ldots]$. The $f_o(x)$, $f_1(x)$, $f_2(x)$, ... are indicator functions. Any set of linearly independent functions could serve as the set of indicator functions, but for ease of manipulation the most commonly used indicator functions are the monomials $f_i(x) = x^i$ $(i=0,1,2,\ldots)$. We shall use these functions exclusively. Then

$$F(x) = \sum_{i=0}^{\infty} a_i x^i. \qquad (31)$$

The benefits that derive from the use of generating functior are a consequence of the operations that can be performed on them Some examples follow:

a. Change of formal mark. Let $F(x)$ generate $[a_0, a_1, a_2, \ldots]$. Then $F(cx)$ generates $[a_0, a_1c, a_2c^2, \ldots]$. Example: $\frac{1}{2}[F(x) + F(-x)]$ and $\frac{1}{2}[F(x) - F(-x)]$ generate $[a_0, a_2, a_4, \ldots]$ and $[a_1, a_3, a_5, \ldots]$, respectively.

b. Addition. If $F(x)$ generates $[a_0, a_1, a_2, \ldots]$ and $G(x)$ generates $[b_0, b_1, b_2, \ldots]$, then $F(x) + G(x)$ generates $[a_0+b_0, a_1+b_1, a_2+b_2, \ldots]$.

c. Multiplication. If $F(x)$ generates $[a_0, a_1, a_2, \ldots]$ and $G(x)$ generates $[b_0, b_1, b_2, \ldots]$, then $F(x)G(x)$ is the generating function of $[c_0, c_1, c_2, \ldots]$, where

$$c_k = \sum_{i=0}^{k} a_i b_{k-i}.$$

d. Differentiation. If $F(x)$ is the generating function of $[a_0, a_1, a_2, \ldots]$, then

$$F'(x) = a_1 + 2a_2x + 3a_3x^2 + \ldots = \sum_{k=0}^{\infty} (k+1)a_{k+1}x^k.$$

Hence $xF'(x)$ generates $[0 \times a_0, 1 \times a_1, 2 \times a_2, \ldots]$.

Perhaps the most important generating function is the one that generates $[1, 1, \ldots]$. Let us write down two mathematical facts needed for its derivation.

a. The binomial theorem for exponent n a negative integer:

$$(x+a)^n = x^n + \binom{n}{1}x^{n-1}a + \ldots + \binom{n}{r}x^{n-r}a^r + \ldots. \qquad (32)$$

(This is really an expression that holds for n any integer -- When n is a positive integer the series terminates with $\binom{n}{n}x^0a^n = a^n$.)

b. $\binom{-r}{k} = (-1)^k \binom{r+k-1}{k}$ for k an integer.

Then $\frac{1}{1-x} = (1-x)^{-1} = 1 - \binom{-1}{1}x + \binom{-1}{2}x^2 - \ldots$

$$= 1 + \binom{1}{1}x + \binom{2}{2}x^2 + \ldots$$

$$= 1 + x + x^2 + \ldots,$$

i.e. $\frac{1}{1-x}$ generates $[1,1,1,\ldots]$. We have an interesting special case of $F(x)G(x)$ when $F(x) = a_o + a_1x + a_2x^2 + \ldots$ and $G(x) = \frac{1}{1-x} = 1 + x + x^2 + \ldots$ Then $\frac{1}{1-x}F(x)$ is the generatin function of $[a_o, a_o+a_1, a_o+a_1+a_2, \ldots]$. For this reason $\frac{1}{1-x}$ is sometimes called the summing operator.

Example: Find $\sum_{k=1}^{n} k^2$. Here the approach is to find first the generating function of $[0^2,1^2,2^2,\ldots]$, apply the summing operator to it, and pick out the coefficient of the term in x^n from the result.

Solution: Since $\frac{1}{1-x}$ generates $[1,1,1,\ldots]$, we have that $x\frac{d}{dx}\left(\frac{1}{1-x}\right)$ generates $[0,1,2,\ldots]$, and $x\frac{d}{dx}\left(x\frac{d}{dx}\left(\frac{1}{1-x}\right)\right)$ generates $[0^2,1^2,2^2,\ldots]$. The differentiation results in $\frac{x(1+x)}{(1-x)^3}$, which, on application of the summing operator becomes $\frac{x(1+x)}{(1-x)^4}$. The binomial theorum (32) tells us that $\frac{1}{(1-x)^4} = \sum_{i=0}^{\infty}\binom{3+i}{i}x^i$. Hence the coefficient of x^n is $\binom{3+n-1}{n-1} + \binom{3+n-2}{n-2}$, which, on evaluation, becomes $\frac{n(n+1)(2n+1)}{6}$.

Let us return to the solution of recurrence relations. Again take the Fibonacci recurrence (19), $a_n - a_{n-1} - a_{n-2} = 0$, which is valid for $n \geq 3$ with boundary values $a_1 = a_2 = 1$. Choosing $a_o=1$ makes it valid for n=2 as well. We multiply the equation by x^n and sum it from n=2 to $n=\infty$:

$$\sum_{n=2}^{\infty} a_n x^n - \sum_{n=2}^{\infty} a_{n-1}x^n - \sum_{n=2}^{\infty} a_{n-2}x^n = 0. \qquad (33)$$

Now, in terms of F(x) as defined by (31),

$$[F(x)-a_o-a_1x] - x[F(x)-a_o] - x^2F(x) = 0,$$

or

$$F(x) = \frac{x}{1-x-x^2},$$

and we next have to restore F(x) to the form (31). By the method

of partial fractions, we get

$$F(x) = \frac{x}{(1-cx)(1-dx)}$$

$$= \frac{1}{c-d}\left[\frac{cx}{1-cx} - \frac{dx}{1-dx}\right],$$

where $c = \frac{1+\sqrt{5}}{2}$, $d = \frac{1-\sqrt{5}}{2}$. But $\frac{cx}{1-cx} = cx + c^2x^2 + c^3x^3 + \ldots$.

Hence $$F(x) = \frac{1}{c-d} \sum_{k=0}^{\infty} (c^k - d^k)x^k ,$$

$$a_n = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right].$$

Now consider the pancake cutting problem, as defined by recurrence (20). Carry out the summation:

$$\sum_{n=1}^{\infty} a_n x^n = \sum_{n=1}^{\infty} a_{n-1}x^n + \sum_{n=1}^{\infty} nx^n. \qquad (34)$$

Deal first with the last term. We have

$$\frac{1}{1-x} = 1 + x + x^2 + x^3 + \ldots .$$

Hence, $$\frac{1}{(1-x)^2} 1 + 2x + 3x^2 + 4x^3 + \ldots ,$$

and $$\frac{x}{(1-x)^2} = x + 2x^2 + 3x^3 + 4x^4 + \ldots = \sum_{n=1}^{\infty} nx^n.$$

Substitution into (34) gives $F(x) - a_o = xF(x) + \frac{x}{(1-x)}2$, and, since $a_o = 1$,

$$F(x) = \frac{x}{(1-x)^3} + \frac{1}{(1-x)} .$$

We know that $\frac{1}{(1-x)^3}$ generates $\left[\binom{2}{0}, \binom{3}{1}, \ldots, \binom{2+k}{k}, \ldots\right]$, and $\binom{2+k}{k} = \frac{1}{2}(k+1)(k+2)$. Hence the sequence generated by $\frac{x}{(1-x)^3}$ is $[0,1,3,\ldots,\frac{1}{2}k(k+1),\ldots]$, and, since $\frac{1}{1-x}$ generates $[1,1,1,\ldots,1,\ldots]$, we have

$$a_n = \tfrac{1}{2}n(n+1) + 1.$$

2.4 Recurrence relations with variable coefficients

This is a very complicated topic. Here we shall look at just one equation, the particularly simple linear first-order relation

$$a_{n+1}\, s(n) - a_n t(n) = u(n).$$

On dividing through by $s(n)$ this becomes

$$a_{n+1} - a_n\, g(n) = f(n)\ . \tag{35}$$

We want to express (35) in a form that we know how to deal with. Suppose first that $f(n) = 0$. Then

$$a_{n+1} - a_n\, g(n) = 0\ , \tag{36}$$

where $g(n)$ is still a variable coefficient. However, if (36) is divided by $\Pi_{k=1}^{n}\, g(k)$, the result is

$$\left[a_{n+1}/\Pi_{k=1}^{n}\, g(k)\right] - \left[a_n/\Pi_{k=1}^{n-1}\, g(k)\right] = 0, \tag{37}$$

and this is a homogeneous equation with constant coefficients in the variable

$$a_n/\Pi_{k=1}^{n-1}\, g(k). \tag{38}$$

The characteristic equation of (37) is linear, and the solutio is therefore an arbitrary constant A. The change of variable has now to be put into effect throughout (35). This results in

$$\left[a_{n+1}/\Pi_{k=1}^{n}\, g(k)\right] - \left[a_n/\Pi_{k=1}^{n}\, g(k)\right] = f(n)/\Pi_{k=1}^{n}\, g(k). \tag{39}$$

It can be shown that the solution of (39) is

$$A + \textstyle\sum_{k=1}^{n-1} \left[f(k)/\Pi_{i=1}^{k}\, g(i)\right]$$

in terms of variable (38). Hence

$$a_n = \Pi_{k=1}^{n-1}\, g(k)\ \left[A + \textstyle\sum_{k=1}^{n-1} \left\{ f(k)/\Pi_{i=1}^{k}\, g(i)\right\}\right]. \tag{40}$$

Examples:

1. $a_{n+1} - a_n n = 1$. Here $g(n)=n$, $f(n)=1$, and we have

$$a_n = (n-1)! \left[A + \sum_{k=1}^{n-1} 1/k!\right].$$

2. $n^2 a_{n+1} - (n+1)^2 a_n = n(n+1)$, with $a_1=1$.

Divide through by n^2: $a_{n+1} - \left(\frac{n+1}{n}\right)^2 = \frac{n+1}{n}$.

$$a_n = \prod_{k=1}^{n-1} \left(\frac{k+1}{k}\right)^2 \left[A + \sum_{k=1}^{n-1} \frac{k+1}{k} \Big/ \prod_{i=1}^{k} \left(\frac{i+1}{i}\right)^2\right]$$

$$= n^2 \left[A + \sum_{k=1}^{n-1} \frac{k+1}{k} / (k+1)^2\right]$$
$$= n^2 \left[A + \sum_{k=1}^{n-1} \frac{1}{k(k+1)}\right]$$
$$= n^2 \left[A + \sum_{k=1}^{n-1} \left(\frac{1}{k} - \frac{1}{k+1}\right)\right]$$
$$= n^2 \left[A + \frac{n-1}{n}\right].$$

Hence $a_n = n^2 A + n(n-1)$, and use of $a_1=1$ gives $A=1$.

The solution: $a_n = n^2 + n(n-1) = n(2n-1)$.

2.5 Effect of balancing on binary search trees

Let a binary search tree contain N nodes, and let n_i be the number of nodes located on level i. Originally the N records could have been entered in N! different sequences, giving rise to N! binary trees (which need not all be distinct). The different possibilities for three records, with keys 1,2,3, are shown in Fig. 10.

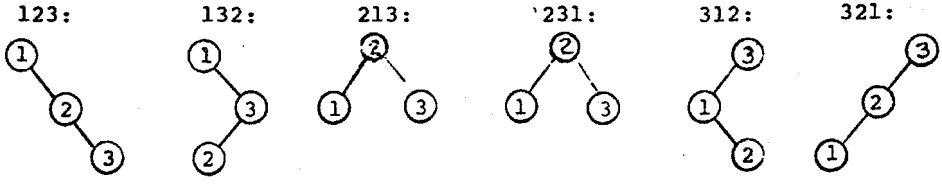

Figure 10

For a given tree the total number of comparisons required to find all N records divided by N is the expected number of comparisons required to find an arbitrary record in the tree. However, since there are N! trees, these expected numbers for particular trees have

to be averaged out over the N! trees for the overall expected number of comparisons. Denote this quantity by C_N, and denote by C_N' the expected number of comparisons in a balanced binary search tree. We have

i	1	2	3	4	5	
C_i	1	3/2	17/9	53/24	149/60	
C_i'	1	3/2	5/3	2	11/5	(41)
ratio	1	1	1.13	1.10	1.13	

Derivation of C_N' is easy:

$$C_N' = \frac{1}{N} \sum_{i=0}^{M} (i+1)n_i \qquad (M = \lfloor \log_2 N \rfloor),$$

where, by definition of a balanced binary tree,

$$n_i = 2^i \quad \text{for} \quad i < M, \qquad \text{and} \qquad n_M = N - (2^M - 1).$$

Hence

$$\begin{aligned} C_N' &= \frac{1}{N}(M+1)(N+1-2^M) + \frac{1}{N} \sum_{i=0}^{M-1} (i+1)2^i \\ &= (M+1)\left(1+\frac{1}{N}\right) - \frac{(M+1)2^M}{N} + \frac{(M-1)2^M+1}{N} \\ &= (M+1)\left(1+\frac{1}{N}\right) - \frac{2^{M+1}}{N} + \frac{1}{N} \\ &= M, \qquad \text{for large } N. \end{aligned} \tag{42}$$

There are three stages to the determination of C_N, namely the setting up of a recurrence equation, the solution of this recurrenc and the simplification of the solution by means of an approximatior The first of these is the most difficult.

When a search tree contains N-1 nodes, the number of positions that can be occupied by an N-th node on level i is

$$2n_{i-1} - n_i, \qquad (i<0),$$

as illustrated by Fig. 11. Moreover, if we look at the entire tree

there are precisely N locations at which the N-th node can be placed: each occupied position suspends 2 arcs, and each of N-2 nodes is at the end of some arc (the N-1 nodes except the root); hence the number of locations available for the new node is $2(N-1)-(N-2) = N$. On the assumption that the new node is equally likely to occupy any one of these N possible locations, the probability that the new node will go into level i is

$$\frac{1}{N}\,(2n_{i-1} - n_i).$$

With the new node added there are now N-1 nodes for which the expected number of comparisons is C_{N-1}, and the added node for which it is $\frac{i+1}{N}(2n_{i-1} - n_i)$ summed over all levels. Hence

$$C_N = \left[(N-1)C_{N-1} + \sum_{i=1}^{N-1} \frac{i+1}{N}(2n_{i-1}-n_i)\right]/N.$$

Then

$$\begin{aligned} N^2C_N - N(N-1)C_{N-1} &= \sum_{i=1}^{N-1} (i+1)(2n_{i-1} - n_i) \\ &= \sum_{i=1}^{N-1} in_{i-1} + 2\sum_{i=1}^{N-1} n_{i-1} + \sum_{i=1}^{N-1}(in_{i-1}-in_i) - \sum_{i=1}^{N-1} ni. \end{aligned} \tag{43}$$

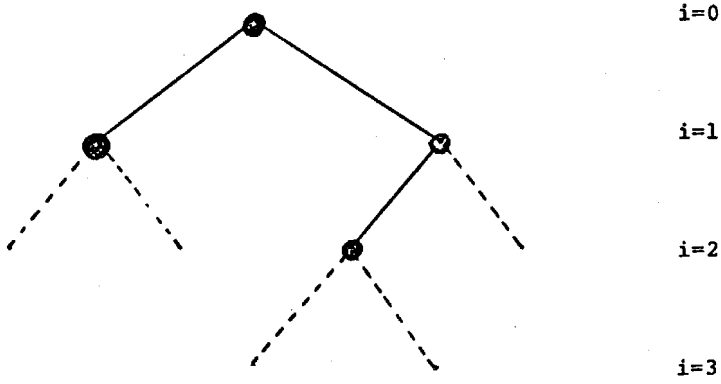

Figure 11

The reason for splitting up the $\sum 2in_{i-1}$ is to produce $\sum_{i=1}^{N-1} (in_{i-1} - in_i)$, which becomes $(\sum_{i=0}^{N-2} n_i - (N-1)n_{N-1})$, and, since $n_{N-1} = 0$, ultimately becomes $\sum_{i=0}^{N-2} n_i = N-1$. Also $\sum_{i=1}^{N-1} n_{i-1} = \sum_{i=0}^{N-2} n_i = N-1$. Hence (43) becomes

$$\begin{aligned} N^2C_N - N(N-1)C_{N-1} &= (N-1)C_{N-1} + (N-1) + \sum_{i=0}^{N-2} n_i + (N-1) - \sum_{i=1}^{N-1} n_i \\ &= (N-1)C_{N-1} + 2(N-1) + n_o - n_{N-1} \\ &= (N-1)C_{N-1} + 2(N-1) + 1, \end{aligned}$$

and, after some rearrangement,

$$C_N = (1-\tfrac{1}{N^2})\, C_{N-1} + \tfrac{2}{N} - \tfrac{1}{N^2}.$$

This put into form (35) becomes

$$C_{N+1} = (1 - \tfrac{1}{(N+1)^2})\, C_N + \tfrac{2}{N+1} - \tfrac{1}{(N+1)^2}. \qquad (44)$$

Now, in accordance with (40), equation (44) has the solution

$$\begin{aligned} C_N &= \Pi_{k=1}^{N-1} \left[1 - \tfrac{1}{(k+1)^2}\right] \left[A + \sum_{k=1}^{N-1} \left\{ \tfrac{2k+1}{(k+1)^2} / \Pi_{i=1}^{k} \left[1 - \tfrac{1}{(i+1)^2}\right] \right\}\right] \\ &= \Pi_{k=2}^{N} \left[1 - \tfrac{1}{k^2}\right] \left[A + \sum_{k=2}^{N} \left\{ \tfrac{2k-1}{k^2} / \Pi_{i=2}^{k} \left[1 - \tfrac{1}{i^2}\right] \right\}\right]. \end{aligned}$$

It can be shown that $\Pi_{k=2}^{n} \left(1-\tfrac{1}{k^2}\right) = \tfrac{n+1}{2n}$. Hence

$$\begin{aligned} C_N &= \left(\tfrac{N+1}{2N}\right) \left[A + \sum_{k=2}^{N} \left(\tfrac{2k-1}{k^2}\right)\left(\tfrac{2k}{k+1}\right)\right] \\ &= \left(\tfrac{N+1}{2N}\right) \left[A + \sum_{k=2}^{N} \tfrac{2(2k-1)}{k(k+1)}\right], \quad N \geq 2. \end{aligned}$$

In now remains to use $C_2 = 3/2$ to establish that A=1. Then

$$C_N = \tfrac{N+1}{2N} \left[1 + \sum_{k=2}^{N} \tfrac{2(2k-1)}{k(k+1)}\right]$$

$$= \left(\frac{N+1}{2N}\right)\left[1 + \sum_{k=2}^{N}\left\{\frac{6}{k+1} - \frac{2}{k}\right\}\right]$$

$$= \left(\frac{N+1}{2N}\right)\left[\sum_{k=3}^{N}\frac{4}{k} + \frac{6}{N+1}\right]$$

$$= \left(\frac{N+1}{2N}\right)\left[4H_N - 4 - 2 + \frac{6}{N+1}\right]$$

$$= 2\left(1+\frac{1}{N}\right)H_N - 3, \qquad (45)$$

where H_N is the harmonic number $H_N = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{N}$.

Finally it remains to simplify (45) by removing the summation that is implicit in H_N. There exists the expansion

$$H_N = \log_e N + 0.577216 + \frac{1}{2N} - \frac{1}{12N^2} + \frac{1}{120N^4} - \ldots$$

Hence, for large N,

$$C_N \approx 2H_N \approx 2\log_e N. \qquad (46)$$

Thus,

$$C_N/C'_N = 2\log_e N/\log_2 N = 2\log_e 2 = 1.39.$$

As (41) shows, the ratio is even smaller for small N. Our conclusion is that one need not be greatly concerned with the lack of balance in a binary search tree, or, expressed more precisely, increase in search costs due to lack of balance has to be weighed against the cost of keeping the search tree balanced. However, one should keep in mind that in the worst case, when the height of the binary search tree that holds the N records is N-1, the expected number of probes required to retrieve an item is $\frac{1}{2}(N+1)$ - the probability that the tree has height N-1 is $2^{N-1}/N!$.

2.6 Analysis of search in AVL trees

Recurrence relation (28) gives a_n, the number of nodes in the most skewed AVL tree of height n:

The particular solution we look for is of the form dn, and we do in fact obtain

$$K_n^{(p)} = \left(\frac{3+\sqrt{5}}{5+\sqrt{5}}\right) n.$$

Since only the particular solution depends on n we can make a further approximation, taking just the particular solution for the total solution. Thus

$$K_n \approx \left(\frac{3+\sqrt{5}}{5+\sqrt{5}}\right) n \qquad (52)$$

The n in (52) is height of the AVL tree. We want to express K_n in terms of $a_n = N$, the number of nodes in this AVL tree. Going back to the solution of (28), we see that for large n

$$\log a_n \approx n \log \left(\frac{1+\sqrt{5}}{2}\right),$$

giving

$$K_N \approx \log N \left(\frac{3+\sqrt{5}}{5+\sqrt{5}}\right) / \log \left(\frac{1+\sqrt{5}}{2}\right) \qquad (53)$$

Using base 2 logarithms in (53) we obtain

$$K_N = 1.04 \log_2 N. \qquad (\text{c.f. } C'_N = \log_2 N)$$

This is a very encouraging result, particularly if we keep in mind that it pertains to the worst case. Empirical studies show that on the average $\hat{K}_N = \log_2 N + 0.25$.

In the derivation of (53) a total of four approximations wer made. Keep in mind that the courage to make approximations is a sign of mathematical maturity.

2.7 Insertions and deletions in AVL trees

Insertion of a new record in a balanced binary search tree

holding N records is in the worst case an O(N) process in that the restructuring necessitated by the insertion may change every parent-successor relationship in the tree. An example of this is provided by Fig. 12.

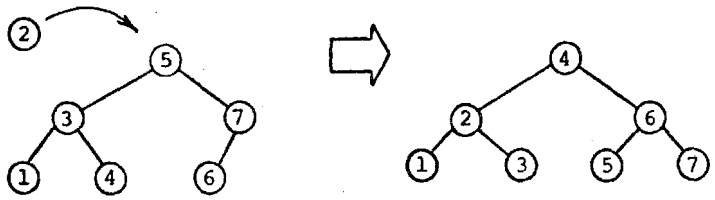

Figure 12

In an AVL tree, on the other hand, restoration of height balance after an insertion is a local phenomenon confined to a path from root to a terminal node, and, since the height of an AVL tree is O(log N), insertion is an O(log N) process. In what follows it will be assumed that each node in the tree carries a "balance flag", which is $\oslash$ if the height of the left subtree of the node exceeds that if the right subtree, $\ominus$ if the right subtree has greater height, and $\ominus$ if both subtrees are of the same height. All terminal nodes carry balance flag $\ominus$.

If height balance is violated by the insertion of a new (terminal) node, it can be restored by the single application of one of two rotation operations defined below. On the path from the root of the tree to the point of insertion of the new node consider a subpath (A,B,C), where A is the root of a subtree that does not have the AVL property, but the subtree rooted at B does have this property. The cause for the lack of balance is the newly inserted node, and to restore balance it would suffice to shift just this one node in such a way that its level number is decreased. However, we also have to maintain the search tree property, which complicates matters somewhat, and we find that conservation of the search tree property requires that nodes A,B, C, be rearranged, and that the subtrees suspended from these nodes (if there are any) be shifted around as well. There are four cases, corresponding to the four forms that subpath (A,B,C) can take. In two of the cases we speak of a single rotation

of nodes A, B, C, in the other two of a double rotation of these nodes. Fig. 13 shows the rotations (a and b are single rotations, c and d are double rotations), and Fig.14 shows the rearrangement of the subtrees. Figs. 15 and 16 show two examples: they represer respectively, cases c and a.

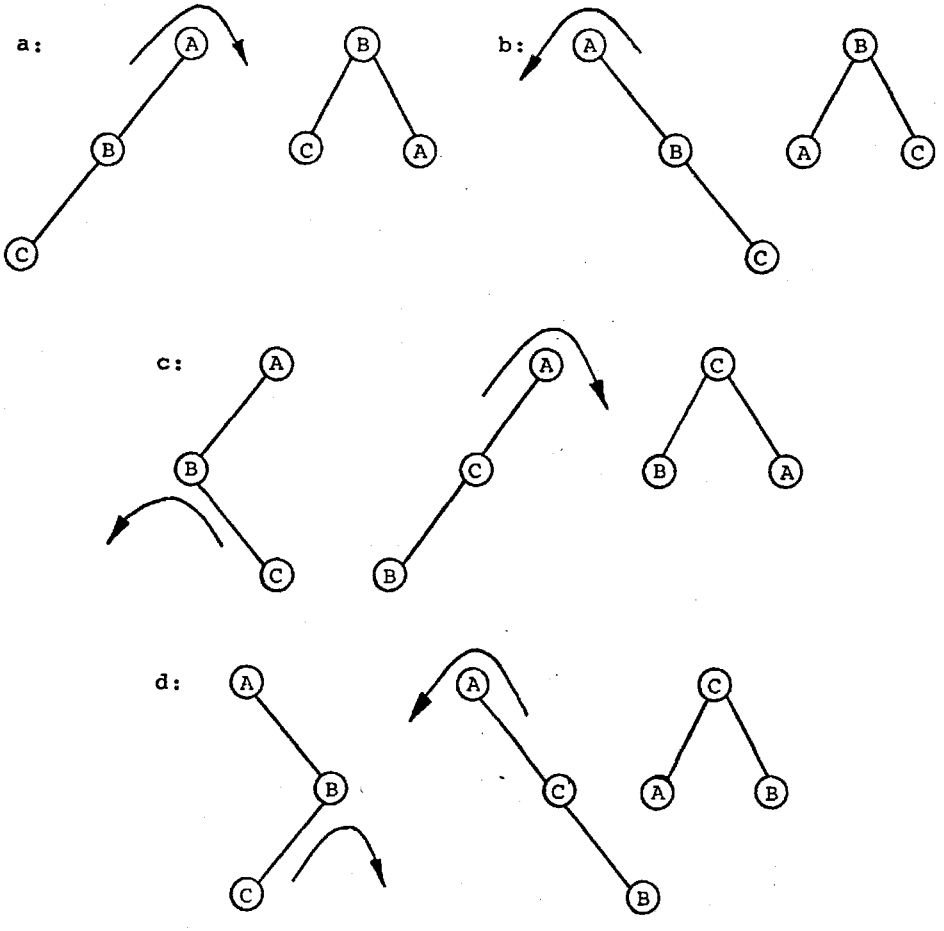

Figure 13

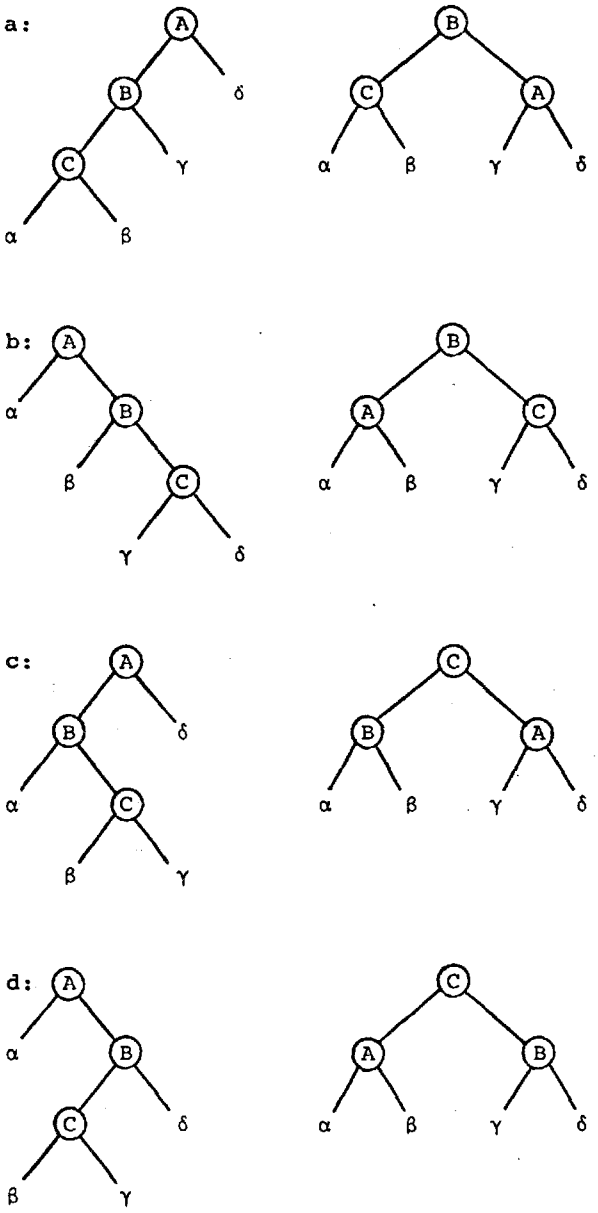
a:
A
B
δ
C
γ
α
β
B
C
A
α
β
γ
δ
b:
A
α
B
β
C
γ
δ
B
A
C
α
β
γ
δ
c:
A
B
δ
α
C
β
γ
C
B
A
α
β
γ
δ
d:
A
α
B
C
δ
β
γ
C
A
B
α
β
γ
δ

Figure 14

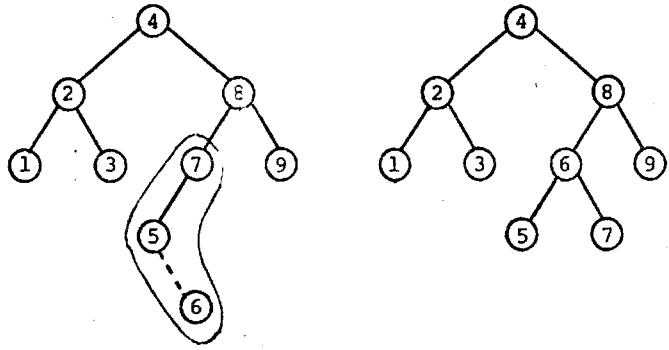

Figure 15

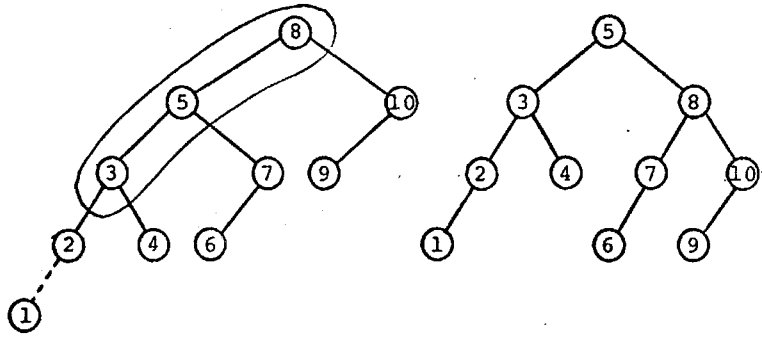

Figure 16

The following restructuring process takes place after an insertion: For every arc on the path from root to newly inserted node, moving upward toward the root, perform the action indicated by the table of Fig. 17. Note that in the instances in which rotation is required the flags of C and B will have been altered before A is reached. These flags may then have to be reset. The flag settings follow the scheme of Fig. 18.

Deletion differs from insertion in that deletion can take place anywhere in the tree, i.e., the deleted node may be interna One way of solving this problem is to initiate an inorder traver of the right subtree of the node that is to be deleted, and to sh

Balance flag of tail of arc	Orientation of arc	Orientation of arc examined before this arc	Action
⊖	R		set flag ⊘
⊖	L		set flag ⊘
⊘	R		set flag ⊖ ; stop
⊘	L	L	a: rotate; stop
⊘	L	R	c: d.rotate; stop
⊘	R	L	d: d.rotate; stop
⊘	R	R	b: rotate; stop
⊘	L		set flag ⊖ ; stop

Figure 17

Cases a,b: set flags of A,B to ⊖ ; leave flag of C unchanged

Case c:

setting of flag of C	new settings of flags of A	B	C
⊖	⊖	⊖	⊖
⊘	⊘	⊖	⊖
⊘	⊖	⊘	⊖

Case d:

setting of flag of C	new settings of flags of A	B	C
⊖	⊖	⊖	⊖
⊘	⊖	⊘	⊖
⊘	⊘	⊖	⊖

Figure 18

the key associated with a node in this traversal sequence into its predecessor node in terms of the inorder sequence. Ultimately the key associated with a terminal node is so shifted, and this terminal node becomes vacant. Now a procedure analogous to that for insertion can be used to restore balance in the tree. No more than O(log N) nodes are examined, i.e., deletion is also an O(log N) process.

3. GRAPHS AND K-TREES

3.1 Representation of digraphs by K-trees

Consider the digraph of Fig. 19. For each node, let us generate a tree consisting of the node as root and the arcs originating from this node. The result is Fig. 20. There trees are called atomic K-trees, and the atomic K-tree having node x as root is the atomic K-tree of node x. The following algorithm merges atomic K-trees into one or more composite K-tree(s). For our example the result is the K-tree shown in Fig. 21. Since we are dealing here with both digraphs and K-trees representing digraphs, a distinction must be made between nodes of the original digraph and nodes in the K-tree representation. We shall use the terms D-node and K-node to differentiate between the two.

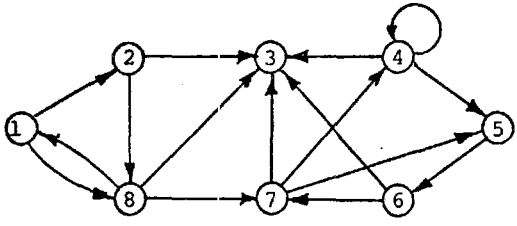

Figure 19

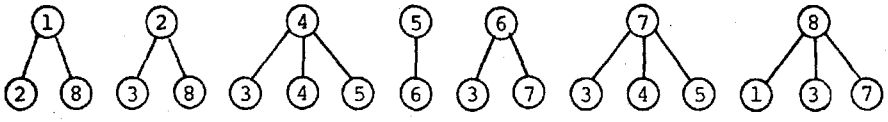

Figure 20

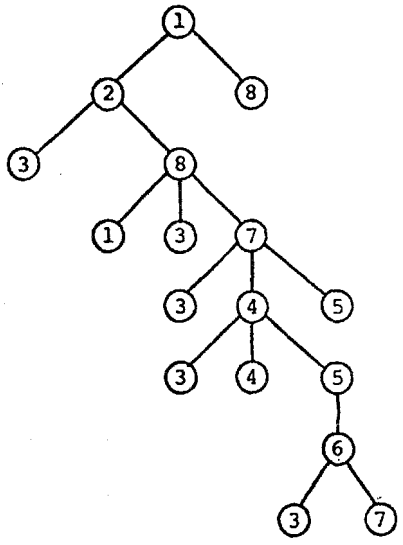

Figure 21

ALGORITHM. The input to the algorithm is a list of atomic K-trees $T=t_1,t_2,\ldots,t_n$, where t_i is the atomic K-tree of node i (for some D-node j there may not exist a t_j because no arcs originate from j in the digraph). The output consists of lists K and K' of composite K-trees. Lists T', K, and K' are assumed to be initially empty.

```
comment   determine D-nodes with zero indegree;
for i:=1 to n do idnonzero[i]:= false;
for i:=1 to n do
if t_i exists then
   begin for all terminal nodes of t_i do
         begin y:=label of terminal node;
               idnonzero[i]:=true
         end
   end;
for i:=1 to n do
if idnonzero[i]∧t_i exists then transfer t_i from T to T';
while T not empty do
```

```
begin comment: tree building phase;
     k:=any atomic K-tree still in T;
     delete this atomic K-tree from T;
     for all terminal nodes in k while k traversed under preorder do
     begin y:=label of terminal node;
          if t_y is in T' then
          begin replace terminal node y by atomic K-tree t_y;
               delete t_y from T'
          end
     end;
     add k to list K
end tree building phase;
while T' not empty do
begin comment: merge atomic K-trees that may still be in T';
     k:=any atomic K-tree still in T';
     delete this atomic K-tree from T';
     for all terminal nodes in k while k traversed under preorder do
     begin y:=label of terminal node;
          if t_y is in T' then
          begin replace terminal node y by atomic K-tree t_y;
               delete t_y from T
          end
     end;
     add K-tree k to list K'
end;
```

Any set of trees that contain precisely the arcs of a digraph D is a K-tree representation of D, but it helps to narrow this definition somewhat. Let a list of K-trees $k_1, k_2, \ldots, k_u$ represent D. This list is a depth-first representation of D iff the following holds for every D-node s from which arcs originate: there is a unique internal K-node s, and, if this internal K-node is in K-tree k_i, then there is no terminal node s in any of $k_1, k_2, \ldots, k_{i-1}$, and in k_i no terminal node s precedes the internal s in the preorder sense. If, moreover, u is equal to the cardinality of the node base of D, then the depth-first K-tree representation is minimal.

The node base of a digraph comprises all nodes with zero

indegree, and one node form each nontrivial strong component that has zero indegree as a whole. It is easy to see that the roots of the K-trees in a minimal representation constitute a node base. The output of the algorithm above consists of list K, followed by list K', where either of the lists may be empty, and this output has the depth-first property. Moreover, if K' contains no more than one K-tree, there is minimality. If K' contains more than one K-tree, then a determination has to be made of whether or not the roots of the K-trees belong, as D-nodes, to strong components of zero indegree. If they do not, then the K-trees rooted at these nodes can be absorbed by other members of K', and a minimal representation results. More detail of this at the start of Section 3.3.

The digraph of Fig. 22 provides an example of a digraph for which our algorithm builds a representation consisting of two K-trees, although a representation in terms of a single K-tree is possible.

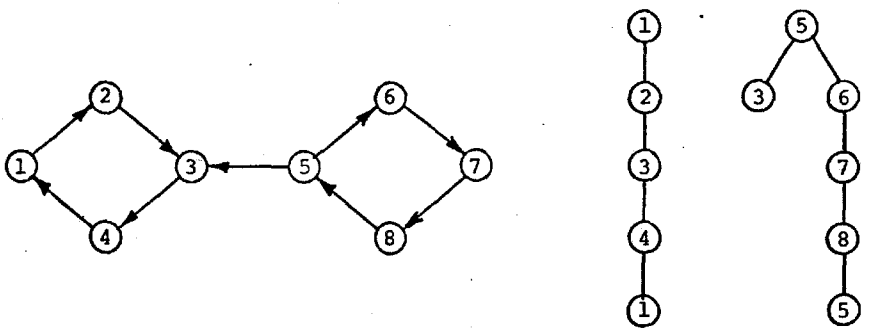

Figure 22

For many algorithms based on K-trees it is not necessary to go through this second phase to ensure that a minimal representation is obtained. For an ayclic digraph the basic algorithm produces a minimal representation. For an acyclic digraph it can be shown that in a depth-first K-tree representation all arcs belonging to a given strong component of the digraph define a connected substructure of a single K-tree of that representation.

As a corollary of the second result we have that a digraph is cyclic iff a K-tree in its depth-first representation contains a path of the form (a,...,a).

The primary advantage of the K-tree representation of a digraph is that is contains precisely the arcs of the original digraph, but in such a way that much of the structure of the digraph is still explicitly there. This contrasts with a direct representation of a digraph by a linked storage structure, which retains all the structural information, but is awkward to operate on, and the representation of the digraph by its adjacency matrix in which the structure becomes totally obscured. Adjacency lists conserve no more structure than does the adjacency matrix; the compression of rows of a matrix into adjacency lists is a standard technique for the representation of sparse matrices in general.

3.2 Depth-first search algorithms and K-trees

In this section the utility of K-trees will be demonstrated by means of algorithms for the critical path analysis of a scheduling network. All the algorithms will have time complexity O(max(m,n)), where m is the number of arcs and n is the number of nodes. For any reasonable digraph, m is at least n/2, and O(max(m,n)) becomes O(m). It will be seen that some of the algorithms are similar to recursive algorithms known as depth-first search algorithms, which have been extensively described in the literature. However, there are significant differences. First, the K-tree approach permits heavy use to be made of the standard tree traversal disciplines, which are well known and well understood. Second, because of the dependence on tree traversals, algorithms can be written non-recursively in a natural manner. Third, the reliance on tree traversals permits one to regard the K-tree and a small number of operations on the K-tree as an abstract data type, a point that will be taken up again in in Chapter 4.

A scheduling network is a 6-tuple $\langle E, e_s, e_t, A, W, w \rangle$, where E is a set of nodes (for convenience we put $E = \{1,2,\ldots,n\}$) e_s is the only node with zero indegree, e_t is the only node with zero outdegree, A is the set of arcs, W is a set of weights, w is a function on A into W that associates the weights with the arcs, and the digraph $D = \langle E,A \rangle$ is acyclic. Then $|E| = n$, and we set $|A| = m$. Members of E, A, and W are called, respectively, events, activities, and durations. The weight associated with a arc $\langle x,y \rangle$ will be denoted by $w[x,y]$. Events e_s and e_t are called start and termination, respectively. If, for every $\langle x,y \rangle \in A$, the relation $x < y$ holds, then the activities are in topological order, and $e_s = 1$ and $e_t = n$. With each node $y \in E$ there are associated two times, the earliest event time eet[y], and the latest event time let[y]. Assuming topological order, they are defined as follows:

$$eet[1] = 0;$$
$$eet[y] = \max_{x<y}(eet[x] + w[x,y]), \quad y \neq 1;$$
$$let[n] = eet[n];$$
$$let[y] = \min_{y>x}(let[x] - w[x,y]), \quad y \neq n.$$

With activities in topological order, the earliest event times can be evaluated for all nodes in the order $1,2,\ldots,n$, and the latest event times in the order $n, n-1,\ldots,1$. With each activity $\langle x,y \rangle \in A$ there is associated a time, called the float of the activity, defined

$$float[x,y] = let[y] - eet[x] - w[x,y].$$

An activity with zero float is critical, and a path $(e_s,\ldots\ldots,e_t)$ consisting entirely of critical activities is a critical path. The purpose of critical path analysis is to evaluate the floats, and hence identify critical paths.

Before one begins with the critical path analysis proper, the absence of certain anomalies has to be established. The anomalies are: (1) the presence of cycles in the network; the

presence of "holes" due to (2) more than one node in the network having zero outdegree, and (3) more than one node having zero indegree.

Anomaly (3) exists iff there is more than one K-tree in list K when the tree-building algorithm of Section 3.1 stops. Tests for the other two anomalies can be easily built into the tree-building algorithm. Here, however, we shall put the tests for anomalies (1) and (2) into our next algorithm, which finds a topological ordering of the nodes of an acyclic digraph. Although the depth-first K-tree representation of a scheduling network consists of a single K-tree, topological order can be imposed on the nodes of any acyclic digraph. For this reason the input to the algorithm will be permitted to consist of a depth-first representation containing more than one K-tree.

ALGORITHM. An O(m) nonrecursive depth-first K-tree algorithm for the topological ordering of nodes in an acyclic digraph. When the algorithm stops, topno[1],..., topno[n] contain the labels that have been given to D-nodes 1,...,n, respectively, to induce topological order.

```
for i:=1 to n do topno[i]:=0;
j:=1; cycles:=false;
for all K-trees in list K in right-to-left order do
    for all K-nodes while K-tree traversed in R(postorder) do
    begin
        y:=label of K-node;
        if K-node y is internal then begin topno[y]:=j; j:=j+1 end
        else if topno[y]≠0 then cycles:=true
    end;
comment: cycles = true implies anomaly (1);
if j=n then topno[y]:=n
else begin comment: anomaly(2) exists;
          for i:=1 to n do
          if topno[i]=0 then begin topno[i]:=j; j:=j+1 end
     end;
```

Of the n D-nodes of a scheduling network, n-1 should have nonzero outdegree. The D-nodes with nonzero outdegree are in one-one correspondence with internal K-nodes, and j is incremented each time one of the latter is processed. Hence $j \neq n$ after the traversal has been completed implies the existence of fewer than n-1 D-nodes with nonzero outdegree, i.e., more than one D-node with zero outdegree. The latter are identified by searching through topno for elements that are still zero.

The demonstration that the algorithm detects the presence of cycles is only slightly more complicated. The algorithm subjects the K-trees of list $K=k_1,k_2,\ldots,k_u$ to reverse postorder traversal in the order $k_u,k_{u-1},\ldots,k_1$. Under this strategy all terminal K-nodes labeled y are processed before the one internal K-node y is processed, with a single exception: if there exists a path (y,...,y) in a K-tree, the terminal K-node y on this path is processed after the internal K-node y has already been processed. Hence, in terms of our algorithm, $topno[y] \neq 0$ when a terminal K-node is being processed implies the existence of path (y,...,y), i.e., it implies cyclicity of the digraph.

Actually there is no need to have the events topologically ordered by a separate algorithm. For critical path analysis it suffices to sweep through the single K-tree-representing the scheduling network just twice: a reverse postorder traversal gives earliest event times; a subsequent postorder traversal gives latest event times and floats.

ALGORITHM. A depth-first K-tree algorithm for critical path analysis. It is assumed that the network contains no anomalies. Earliest and latest event times for the n events are stored in arrays eet and let. Although w and float are indicated as two dimensional arrays, they are actually "arc data". Every K-node except the root of the K-tree is the head of an arc: the K-node labeled y whose parent is K-node x can therefore be made to represent arc $\langle x,y \rangle$. This means that w[x,y] and float[x,y] can in practice be represented by appropriate fields in a record that we associate with arc $\langle x,y \rangle$ by associating it with this K-node y.

```
for i:=1 to n do eet[i]:=0;
for all K-nodes except the root while K-tree traversed
    under R(postorder) do
begin y:=label of K-node;
      x = label of parent of y;
      eet[y]:= max(eet[y], eet[x]+w[x,y])
end;
comment: y is still the label of the K-node reached last under
    R(postorder) traversal, and no let that will be computed
    can exceed eet[y];
for i:=1 to n do let[i]:= eet[y];
for all K-nodes except the root while K-tree traversed under
    postorder do
begin y:=label of K-node;
      x:= label of parent of y;
      let[x]:=min(let[x], let[y]-w[x,y]);
      float[x,y]:=let[y]-eet[x]-w[x,y]
end;
```

In other critical path algorithms the eet of a node y is computed in one go, which requires rather costly bookkeeping in that one has to access at this time the tails of all arcs that terminate at y. In our algorithm the K-tree is itself the bookkepping device. Suppose there are j arcs with D-node y as their head. Then, in the K-tree, there are j K-nodes labeled y, and the labels of the parent K-nodes of these j nodes identify the tails (in terms of the network) of the j arcs. We begin with eet[y] set to zero. Then, whenever one of the j K-nodes labeled y is visited during the tree traversal, the sum of the eet of the tail of the arc and the duration associated with the arc is compared with the current value of eet[y], and the larger of the two becomes the new value of eet[y]. The computation of the eet[y] by successive adjustments spread out in time is similar to the method used by Dijkstra in his algorithm for the shortest paths from a single source in a digraph. Computation of latest event times is precisely analogous

In a proof of the algorithm the crucial step is to show that whenever one of the K-nodes labeled y is visited, the eet of its parent be already established, and it is easy to show that this

holds true in a depth-first K-tree traversed under reverse postorder. By symmetry it then follows that latest event times are also computed correctly. Finally, to justify the initialization of all elements of let to the value of eet[y] it has to be shown that the K-node processed last under reverse postorder carries the label of D-node e_t.

If one is interested merely in the determination of critical paths, then there is no need to apply the algorithm in its entirety. The earliest event time of a node is the length of the longest path form e_s to this node. In particular, eet[e_t] is the length of the longest path from e_s to e_t. All longest paths form e_s to e_t are critical, and only these paths.

ALGORITHM. Given a depth-first K-tree of an acyclic digraph D that contains the source as its root, and all nodes reachable from the source. The algorithm computes in array length the lengths of longest paths form the source to all other D-nodes reachable from the source, and specifies the longest paths in array before. One longest path from the source to node i is defined, in reverse order, by i, before[i], before[before[i]],... . If before[j]=0 when the algorithm stops, then j is the source, or it is a node that is not reachable in D from the source.

```
for i:=1 to n do length[i]:=before[i]:=0;
for all K-nodes except the root while K-tree traversed
                                        under R(postorder) do
begin y:=label of K-node;
      x:=label of parent of y;
      newlength:=length[x] + w[x,y];
      if newlength > length[y] then
      begin length[y]:=newlength;
            before[y]:=x
      end
end;
```

As given above, the algorithm determines just one longest path from the start node to any other node reachable from it.

To generalize the algorithm, one has to make before[y] a list of node labels, and refine the comparison step to the following:

```
if newlength > length[y] then
begin length[y]:=newlength;
      make before[y] an empty list;
      add x to list before [y]
end
else if newlength=length[y] then add x to list before[y];
```

If all length [i] are initialized to a number greater than any feasible path length, length[root] is then set to zero, and greater than (>) replaced by less than (<), the algorithm becomes one for shortest paths in an acyclic digraph. Fig. 23 shows the K-tree of a acylic digraph with arc lengths as shown. The algorithm would compute the length of the shortest path (1,...,5) as 5 instead of the correct 4.

Although our topological ordering algorithm detects cyclicity and either the tree building algorithm or the critical path algorithm can be modified to do so as well, there still remains the task of identifying all arcs that lie on cycles. Strong component analysis seems to provide the fastest means of identifying such arcs: an arc lies on a cycle iff it belongs to a strong component of the digraph. The algorithm given below is therefore a strong component algorithm that identifies arcs that lie on cycles. It is an easy matter to modify it so that it defines strong components in terms of nodes: delete the statement that effects output of the arcs, and output the node labels as they are popped up from the stack. The labels that are popped up in the one pop-up sequence define a strong component.

ALGORITHM. A non-recursive strong component algorithm for a depth-first K-tree representation of a digraph. The output consists of arcs that belong to strong components (if there are any such arcs). Each D-node is pushed down exactly once. Each arc is traversed once under preorder, once under postorder, and the traversals are intertwined. The first time "advance to next K-node under preorder" is executed, the "advance" is to

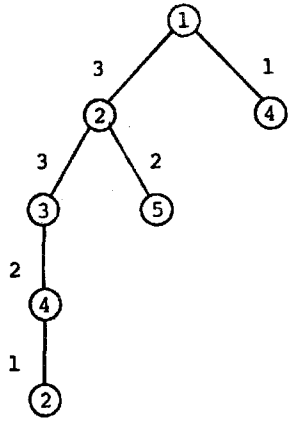

Figure 23

the root of the K-tree. The K-node that is the first to be processed each time the repeat...until alldone loop is entered is the one that was the last to be processed in the preorder phase of the algorithm.

```
for i:=1 to n do begin ordinal[i]:=0; root[i]:=true end;
count:=0;
for all K-trees in left-to-right order do
begin rootdone:=false;
      repeat terminal:=alldone:=false;
          repeat advance to next K-node under preorder;
                 y:=label of K-node;
                 if ordinal[y]=0 then
                 begin ordinal[y]:= count:= count+1;
                       push down y
                 end;
                 if K-node terminal then terminal:=true
          until terminal;
          repeat advance to next K-node under postorder;
                 if K-node is root of K-tree then
                 begin alldone:= rootdone:= true;
                       pop up labels until stack empty
                 end
                 else
                 begin y:=label of K-node;
                      if y on stack then
                      begin x:=label of parent of y;
                            if root[y]∧(ordinal y > ordinal[x]) then
                                 pop up labels until y has been
                                                      popped up
                            else begin output<x,y>;
                                   if ordinal[y]<ordinal[x] then
                                   begin ordinal[x]:=ordinal[y];
                                         root[x]:=false
                                   end
                                 end
                      end;
                      if next K-node in postorder sequence not
                                                      yet visited
```

```
                    under preorder then alldone:=true
                end
            until alldone
        until rootdone
    end;
```

It was stated earlier that in a depth-first K-tree representation all arcs that belong to a given strong component of the digraph define a connected substructure of a single K-tree of that representation. This connected substructure is itself a tree. Denote it by S', and suppose that its root is labeled a. In the preorder traversal phase ordinal numbers are assigned to D-nodes in an increasing sequence. Hence the ordinal assignment made to any D-node in the same strong component as a is initially greater than that made to a. If S' represents a nontrivial strong component, then it must contain a path (a,...,b,a). In the postorder phase the ordinal assignment that was made to a is propagated upward along this path using the criterion that if the current ordinal of a node q is smaller than that of its parent p, then the ordinal assignment of D-node p is changed to that of q. D-node a is the only node in S' whose ordinal assignment does not get changes this way. Moreover, on reaching root node a of S', the ordinal associated with its parent is smaller than that associated with a (unless a is the root of the K-tree and hence has no parent). Thus the root of s is identified by the double criterion the (i) its ordinal assignment has not been changed (Boolean array root is used to detect this), and (ii) its parent carry a smaller ordinal assignment that it itself carries.

When the root of S has been identified all node labels up to and including a are popped up from the stack, and they define S. Here, although our concern is with the arcs that make up S', the stack is still needed. Suppose in the postorder phase arc <c,a> is being examined, and label a is not on the stack. This means that <c,a> cannot belong to a strong component. Output of this arc is therefore prevented, and so is the upward propagation of the ordinal associated with a, which might otherwise take place. Note that the output of the arcs of S may be interspersed

with the output of arcs belonging to other strong components, i.e., arcs that define the one strong component need not be written out in a contiguous sequence.

The K-tree of Fig. 24 shows that both of the conditions given above are needed to identify a node as the root of a strong component. Without condition (i) node v would be interpreted as the root of a strong component. Without condition (ii), terminal K-node s would trigger the popping up of nodes u,t,s from the stack. Consequently node v, which belongs to the same strong component, would not be recognized as such.

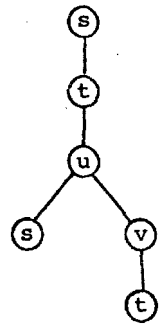

Figure 24

3.3 Algorithms that change the structure of K-trees

Our basic K-tree building algorithm changes the structure of a K-tree. For the construction of K-trees by this algorithm an operation is required that takes an atomic K-tree and grafts it on at a terminal node of the composite K-tree. Further, if we were required to produce a minimal K-tree representation of a digraph, the K-trees in list K' after the basic algorithm stops would be partitioned into a set A of K-trees whose roots belong, as D-nodes, to strong components of zero indegree, and set B of K-trees whose roots do not define such strong components. Then the K-trees of set B would be completly absorbed into those of set A by traversing the latter, and at each terminal node checking whether an internal node with the same label exists within a K-tree belonging to B. If so, the entire subtree rooted at the internal node would be pruned away from its present

location and grafted on at the terminal node that has been reached in the traversal of the K-tree belonging to A.

We next have a K-tree algorithm for the simple cycles of a digraph. This is not a particularly efficient algorithm, and it is included here only as another illustration of pruning and grafting. The best input data to this algorithm is a set of K-trees each of which represents one of the nontrivial strong components of the digraph. The following procedure is applied to every K-tree in this set.

```
push down root;
for all K-nodes except root while K-tree traversed under preorder do
begin y:=label of K-node;
      x:=label of parent of y;
      pop up from stack all nodes up to but not including x ;
      if y is internal then push down y
      else if y not on stack then
              begin prune subtree suspended from internal K-node
                                                    y (if it exists)
                    and graft it on at the terminal node y being
                    processed;
                 push down y which has been marked
              end
           else if neither y nor any node below it in the stack
                        carries a marker
                then output cycle defined by y and the nodes above
                        it in the stack, and the terminal node y
                        being processed
end;
```

Consider now the scheduling of a job on three machines, where some time is required on each of the machines, which may be used in any order. The changeovers from machine to machine differ in cost, and this may make some orders in which the machines are used more expensive than others. The costs are given by matrix C below, where c_{ij} is the cost of changing over from machine i to machine j, and c_{oj} is the initial tooling-up cost for machine j.

$$\begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{bmatrix} 41 & 32 & 35 \\ - & 3 & 8 \\ 4 & - & 9 \\ 9 & 8 & - \end{bmatrix} \\ \quad 1 \quad 2 \quad 3$$

The exhaustive enumeration of Fig. 25 shows that the best sequence is 2-1-3, with cost 44. The method of backtracking is used to derive the best sequence without an enumeration of every possibilty. Under backtracking the tree of Fig. 25 is built in stages, and, hopefully, a solution can be established before the entire tree has been constructed.

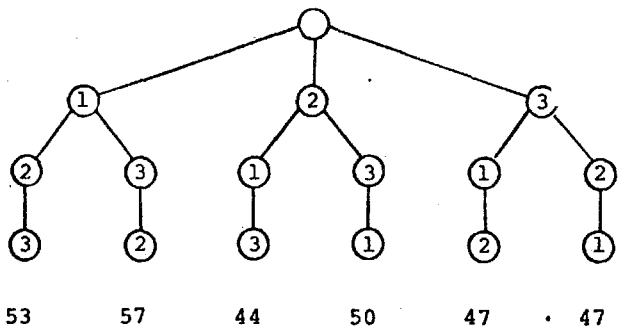

Fig. 25

There are several basic backtracking schemes, which differ in the way the node is selected from which the tree is to be extended. Here we consider only one such strategy because its implementation presents the most interesting data structuring problems. This is the method of best cost in which one branches from the node that has the currently lowest cost value. Costs are evaluated for all successors of the root of the tree. Thereafter branching takes place from the node that has the lowest cost value associated with it.

ALGORITHM. Best cost backtracking algorithm for the n-machine job scheduling problem. Priority queue pqueue is used to find the node of lowest cost value. When the algorithm halts an optimal

solution is given by the path in K from its root to the node pointed to by P.

```
L:=∞;
y:=start-up node;
set cost of start-up node to zero;
K:=atomic K-tree of y;
done:=false;
repeat
    for all successors of y do
    begin x:=node being processed;
          cost:=(cost of y) + (cost of yx);
          if cost < L then
             if x not on level n then
                enter in pqueue a record consisting of cost and of a
                pointer to node x in K, using the value of cost to
                determine the position of the record in pqueue
             else begin L:=cost;
                        P:=pointer to terminal node x
                  end;
          if pqueue empty then done:=true
          else if (lowest cost value in pqueue) ≥ L then done:=true
               else
                  begin extract lowest cost record from pqueue;
                        use it to gain access to a node in K;
                        y:=the node accessed in K;
                        attach atomic K-tree to y in K
                  end
   end
until done;
```

Two remarks. First, the same atomic K-tree may be attached in more than one place at the same time. Second, the storage requirements for the tree may become very high, which suggests that a procedure should be built into the program that prunes away parts of the tree for which there is no further need.

3.4 K-trees of undirected graphs

An edge {a,b} of an undirected graph may be represented by the pair of arcs <a,b> and <b,a>. Fig. 26 shows a graph and the K-tree of the arcs obtained when edges are replaced by arc pairs in this manner. However, since one of the arcs then invariably implies the other, there is redundancy. A systematic way should therefore be sought for selecting just one of the arcs in the pair.

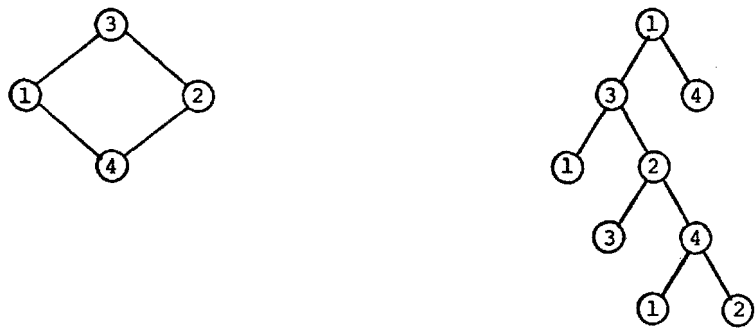

Figure 26

One approach suggests itself immediately: for the representative of {a,b} take that ordered pair (arc) of a and b in which the first coordinate is smaller than the second. Under this scheme, if the K-tree of the arcs is next interpreted as representing a digraph, the digraph that it represents has to be acyclic because it is topologically ordered. It is debatable whether this is the best approach. Applying it to the graph of Fig. 26 we obtain the representation shown in Fig. 27, and this representation consists of two K-trees. It may be argued that an increase in the number of constituents of a representation (K-trees) cannot be regarded as a simplification.

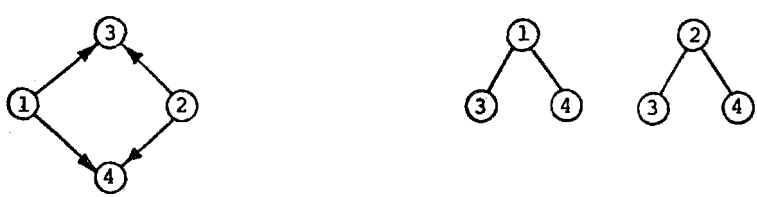

Figure 27

We therefore try a different approach. For each node x in the graph construct an atomic K-tree consisting of x as root and nodes adjacent to x as leaves. Then apply the tree-building algorithm of Section 3.1. For a connected graph the result must be a single tree (because all arcs that belong to the same strong component must be in the same K-tree). Fig. 26 provides an example of the approach to this point. Next subject the K-tree to the following "clean-up" procedure.

```
for all terminal nodes while K-tree traversed under preorder do
begin  x:=label of node;
       y:=label of parent of x;
       z:=label of parent of y;
       if x=z then delete <y,x> from K-tree
       else delete <x,y> from K-tree
end;
```

Fig. 28 shows a K-tree before and after application of this procedure.

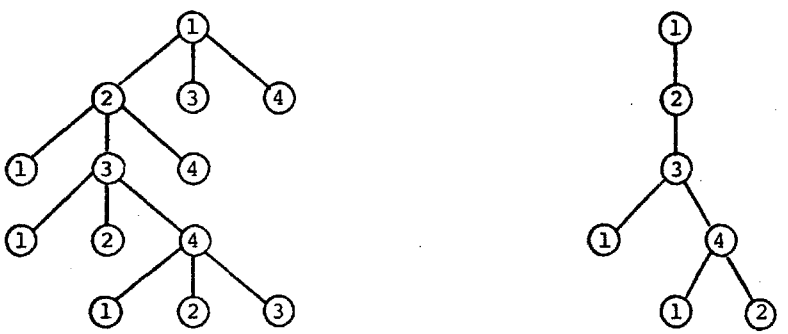

Figure 28

This representation has the interesting property that it holds within itself in easily accessible form the fundamental circuits of the graph. They are all paths of form (a,...,a) in the K-tree. For the example of Fig. 28 they are (1,2,3,1), (1,2,3,4,1), (2,3,4,2).

3.5 K-trees and K-formulas

A K-tree can be represented by a K-formula, and the latter is very easily constructed: carry out a preorder traversal of the K-tree, listing nodes in the order they are encountered; in the case of an internal node precede the node label by as many K-operators (stars) as there are arcs originating from this node. The K-tree of Fig. 21 yields

123***813***73***434*5**63758 (54)

Such a representation is very convenient if it is to be stored in secondary memory or transmitted between the nodes of a computer network. The utility of the K-formular representation increases if one can avoid too many translations between the representations, and one would therefore wish occasionally to perform operations directly on a K-formula. Operations that induce structural changes do not lend themselves well to this, but the operations associated with depth-first search can be easily expressed in terms of K-formulas.

Consider in the K-tree of Fig. 21 internal node 4. Suppose one were required to find its right neighbour. It can be seen easily enough in Fig. 21 that this neighbour is terminal node 5, but its identification in the K-formula is not all that obvious. One has to skip across the representation of the subtree rooted at 4, namely

***434*5**637

and take the next symbol.

The representation of a subtree of a K-tree is a K-formula in its own right, but how does one identify this K-formula? A test derives from a definition of K-formulas as algebraic objects:

a. A node symbol is a K-formula.

b. If α and β are K-formulas, then $*\alpha\beta$ is a K-formula.

From this it follows that string $s_1 s_2 \ldots s_i \ldots s_m$ is a K-formula

iff, letting n_i and k_i denote respectively the number of node symbols and of K-operators in $s_1 \ldots s_i$, the following holds:

$$n_i \leq k_i, \qquad i=1,2,\ldots,m-1 \qquad (55)$$

$$n_m = k_m+1.$$

K-formula (54) contains 33 symbols. The storage requirements can be reduced somewhat if one introduces a bit vector of 33 elements in which an entry is 0 if the corresponding symbol in the K-formlua is a K-operator and 1 if it is a node symbol. Another convenient representation, which does not, however, reduce storage requirements, is

2	2	0	3	0	0	3	0	3	0	0	1	2	0	0	0	0
1	2	3	8	1	3	7	3	4	3	4	5	6	3	7	5	8

(56)

where above each node symbol is a count of the stars that precede it.

Reverse postorder is symmetrical to preorder in that R(post) = C(pre). Hence a K-formula "symmetrical" to (54) can be produced in a reverse postorder traversal of the K-tree of Fig. 21:

182***8***75***4*5**673433313 (57)

This beocmes, in analogy to (56),

2	0	2	3	3	0	3	1	2	0	0	0	0	0	0	0	0
1	8	2	8	7	5	4	5	6	7	3	4	3	3	3	1	3

(58)

We give now algorithms for the preorder and postorder traversals of a K-formula in representation (56). The two arrays will be assumed to be named star and node, respectively, and to be both of size top. Both algorithms make use of stacks node-stack and starstack. The greatest depth to which they are used in either algorithm is equal to the height of the K-tree from

which the K-formula derives. The complexity of either algorithm is O(top). The preorder algorithm seems redundant in that the preorder sequence is simply node[1], node[2],..., node [top]. This algorithm would be used only if, during the traversal, it would be required to identify the parent of the node currently being processed. In both algorithms, at the time node k is being processed, the top element in nodestack is the parent of k.

ALGORITHM. Preorder traversal of a K-formula stored under representation (56).

```
process node[1];
for k:=1 to n-1 do
begin if star[k]≠0 then begin nodestack ← node[k];
                             starstack ← star[k]
                       end;
      process node[k+1];
      i ← starstack;
      i:=i-1;
      if i≠0 then starstack ← i else pop nodestack
end;
```

The "pop nodestack" indicates that a datum is popped up and discarded.

ALGORITHM. Postorder traversal of a K-formula stored under representation (56).

```
for k:=1 to top do
    if star[k]≠0 then begin nodestack ← node[k];
                            starstack ← star[k]
                      end
    else begin process node[k];
               moretodo:=true;
               while moretodo ∧ nodestack≠0 do
               begin i ← starstack;
                     i:=i-1;
                     if i≠0 then begin starstack ← i;
                                       moretodo:=false
```

```
                    end
            else begin x ← nodestack;
                         process x
                 end
        end
end;
```

Reverse postorder traversal is effected by applying the preorder algorithm to representation (58). Again it should be noted that (58) is already in reverse postorder, i.e., the algorithm would be used only if parents of the nodes that are being processed had to be identified.

Binary trees cannot be represented directly by K-formulas because of the left or right orientations associated with their arcs. Here one should first convert the binary tree to a general tree by a reverse Knuth transformation. However, no arc with right orientation can be suspended form the root of a Knuth transform. Consequently, before one applies the transformation, one should introduce a dummy arc in the binary tree. Fig. 29 shows where the dummy arc is added, and Fig. 30 is the K-tree that results when next the reverse transformation is applied.

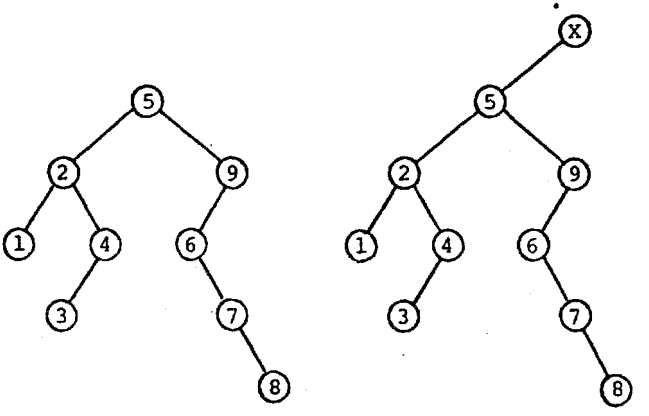

Figure 29

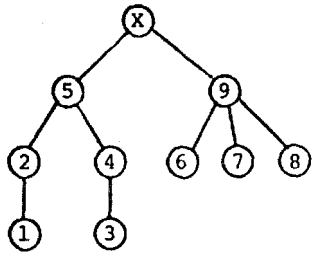

Figure 30

A binary search tree is not a good candidate for implementation as a K-formula because the whole point to using binary search trees is that is is easy to add new nodes to them. This is certainly not so for K-formulas. Similarly, a binary tree functioning as a priority queue is expected to be constantly changing. Still, there exist binary trees which do not undergo structural changes. One example is the ancestor tree of a person in which for every node, the left and right successors represent, respectively, mother and father of the individual represented by this node. In some cases only the mother (or father) might be known, and then only the left (right) successor is present. In such a binary tree one might be interested in determining the parent node of a given node, or the left or right successor of the node.

An algorithm was given above for the determination of the parent of a node. Note that this is the parent of a node in terms of the K-tree, not in terms of the binary tree from which the K-tree derives. This parent node is still important because its determination is the first step in the determination of the parent node in terms of the binary tree.

Consider a node symbol v in a K-formula K. This v is the leading node symbol of a string embedded in K that is itself a K-formula representing the subtree rooted at node v in the K-tree represented by K (if v is terminal, then the embedded K-formula is merely the symbol v). Denote this embedded K-formula by k_i. Now if we consdier two nodes u and v, where u is the parent of v in terms of the K-tree represented by K, then we have in K an embedded K-formula with leading node symbol v, in which there is

the further embedded K-formula k_i with leading node symbol v, i.e., we have

$$\ldots*\ldots*uk_1k_2\ldots k_i\ldots k_t\ldots$$

where $k_1,\ldots,k_t$ are K-formulas, and u is preceded by t K-operators.

Now, in terms of the binary tree from which the K-tree represented by K derives, the parent of v is the leading node of k_{i-1} if i > 1 or u if i=1. Node u is found using the procedure given above, and one then skips $k_1,\ldots,k_{i-2}$ to reach k_{i-1}, where $k_1,\ldots,k_{i-2}$ are identified as well-formed K-formulas by use of (55). A left successor of v exists only if v is preceded by K-operators, and it is then the next node symbol to the right of v. A right successor exists only if $i\neq t$, and it is the leading node of k_{i+1}. Since a well-formed K-formula may be embedded as a proper substring of k_i to the right of symbol v, the only way of finding the right successor of v is again to identify u, and then skip over $k_1,\ldots,k_i$.

4. A PHILOSOPHY OF DATA STRUCTURES

4.1 Trees as abstract data structures

There is still considerable uncertainty as to what is meant by abstract data structure , but all seem agreed that the definition of an abstract data structure has more to do with operations than with relationships between components of the structure. As an example, consider the mathematical object acyclic digraph, which is to be topologically sorted. We can introduce the data structure K-tree, about which all that we need to know is that it comprises certain elements, and that an element may have successors. We need further an operator that selects the successors of an element one by one in a particular order (right-to-left).

Just as it would be very unusual for a programmer to be concerned whether a negative integer is represented by its absolute value and sign, or by the 2's complement of the absolute value, so here consideration of the storage structure or implementation is largely irrelevant as far as the topological sort program is concerned. The storage structure could be an adjacency matrix, a set of arc lists, a general tree, a binary tree, or even a K-formula. This permits changes to be make to storage representations without affecting higher level programs.

What complicates matters is that we need a fairly large number of operations. In addition to operations that provide access to individual elements in a structure, one needs operations that associate data records with the elements and permit one to access the data, and operations that enable structural changes to be made. The first observation then is that a set of operations, including the access operations, has to be predefined, but how far is one to go? The n nodes of a tree can be processed in

n! ways, and surely we cannot provide this many traversal procedures. Perhaps the solution here for binary trees is to predefine just twelve traversals, made up of the three classical orders, RL-level, RL-down, LR-up, and the converses of these six.

There are two problems that have to be solved. First, what happens when a traversal is broken off prematurely? An example: Suppose a heap that is represented by an explicit binary tree as in Fig. 8 is to be converted into an array. The binary tree could be traversed in LR-levelorder, and each node processed by placing its successors into their appropriate locations in the array. If the height of the tree is k, then one can terminate the process after all nodes on level k-1 have been processed because no node on level k can have any successors. Second, how do we deal with intermeshed traversals? An example of this is provided by the strong component algorithm of Section 3.2 .

First suppose that each traversal discipline is implemented by by providing a start and a next operator for the discipline. Intermeshed traversals can be implemented in a straightforward manner. The two operators hide from the user the fact that there still exists some device that drives the traversal. On premature termination of the traversal everything has to be restored to the state that prevailed before the traversal began. In our case, however, premature termination cannot even be detected unless complicated scope rules are defined for the operators, e.g., an operator is assigned to a structure with this assignment holding just within the block in which it is made.

Our second solution is to use the following construct, which was introduced in Section 3.2:

for all K-nodes while K-tree traversed under t do

Premature termination is now easily dealt with: exit from the loop triggers a clean-up. Intermeshing of traversals appears to have become next to impossible. Even for a single traversal there are difficulties. Both the traversal and the processing of the sequence of nodes are essentially procedures, and we have an intermeshing of the two. We would provide a composite traversal package -- the discussion of Section 1.3, particularly Fig. 5,

suggests how this can be done efficiently. Then, however, we have still to provide the programmer with a facility for defining traversals additional to the twelve predefined traversals. The best approach seems to be the use of coroutines. In any case, the provision of traversals as components of data structures would bring about changes in the programming language that is to be used of more than superficial nature.

Another problem relates to traversals and changes to the structures being traversed. In the algorithm for cycles of Section 3.3 the K-tree being traversed undergoes structural changes. We certainly could not in a case like this consider the implementation of a reverse traversal by pushing down an entire direct traversal sequence and then popping up from the stack the reverse sequence. More to the point, what should we permit? Many traversals are undertaken precisely to change the structure that is being traversed. Too restrictive an attitude can therefore not be tolerated. A very permissive approach, on the other hand, exposes the programmer to dangers. For example, it would seem wrong to permit a subtree to be pruned away from the node that is currently being processed, this to be grafted on at a node that has already been processed, and for the traversal to resume from this point of attachment. It is this context that would also give most difficulty with any attempts at program validation.

Note now that the nodes of a K-tree do not possess identifying names. They do carry labels, but these labels identify nodes in the digraph or graph from which the K-tree derives -- the label is part of the data record associated with a K-node. We introduce the concept of an active element. During the execution of a program many different elements in many different structures may be accessed, but one particular element in one particular structure is the last element to have been reached. This is the active element, and it remains active until a different element is reached subsequently. This permits us to access the data in the record associated with an element even though the element is not named. The active element is also useful in indicating an element at which structural changes are to be undertaken.

Let us now consider pointer data. Pointer variables have

been criticized in much the same way as goto statements. One concern has been with dangling pointers, i.e., pointers that continue to point to objects that have ceased to exist. Another problem may be caused when pointers introduce aliases. Suppose a file is to be updated, and the old file retained for security reasons. If there were no additions or deletions, the simplest approach would be to copy the file and then to traverse the copy and make changes. The intended purpose would not be achieved at all if the "copying" were just the setting of a pointer to the old file. This, admittedly is an extreme and contrived example, but it illustrates quite well the type of problem that can arise.

Hence, in general, pointers are dangerous. We certainly demonstrated sufficiently well in Chapter 3 that K-trees can be constructed and used without explicit pointers. The only exception came with the best cost backtracking algorithm of Section 3.3, in which pointers were used for indexing from the priority queue to the K-tree. This is one use of pointers that seems difficult to avoid. However, one can build in sufficient protection to prevent abuses. Thus, one might permit only the currently active element to be assigned to a pointer variable, and let this assignment, a deallocation operation, and one or two predicates be the only means of making reference to a pointer. Whenever an element that has a pointer associated with it becomes active, a move operation makes active the element that is being pointed to. To prevent dangling pointers one might require that all pointers that point to an element be deallocated before the element can be erased. Alternatively, erasure of the element might be made to trigger deallocation of all pointers that reference it. Both of these approaches could become very costly if the node being erased were the root of a large subtree. Strong typing as in Pascal would have to be required.

4.2 Levels of abstraction

These days it is difficult to find an article on data structures which does not have on its first or second page some

part or variant of the following:

```
type      queue
maps      new:                    ⟶ queue,
          push:  queue × item     ⟶ queue,
          pop :          queue    ⟶ queue,
          peek:          queue    ⟶ item,
          null:          queue    ⟶ Boolean;
forall    q ε queue,  i ε item let
          null (new) = true,
          peek (new) = error,
          pop  (new) = error,
          null (push(q,i)) = false,
          peek (push(q,i)) = if null(q) then i
                             else peek(q),
          pop (push(q,i)) = if null(q) then new
                            else push(pop(q),i)
endtype;
```

This is an example of the axiomatic approach to the definition of data structures. The properties of the operations are expressed by axioms, i.e., expressions that relate the operations to each other. Standish finds ten potential advantages in the axiomatic approach. He states that it might eventually be expected

1. To permit clear, rigorous definition of the concepts of data representations and data types.
2. To specify precisely the requirements on data representations.
3. To specify precisely the requirements on programs that manipulate data representations.
4. To permit the widest possible selection of implementation detail.
5. To simplify program maintenance by allowing changes in superstructure or underpinnings to be made independently.

7. To permit concise definition of the semantics of data definition facilities.
8. To establish a framework for tackling the problem of automatic synthesis or selection of data representations.
9. To capture the behavior of composite information of structures that are pervasive in programming.
10. To offer a better basis for machine independence.

A second approach to abstract data structures is to provide an operational specification. Instead of defining the operations in terms of each other, they are described by procedures expressed in some programming language. The two disadvantages of this approach are that the operational specifications may eventually become very long, and that it is difficult to keep out extraneous detail from them, which may limit the independence of data structures and storage structures somewhat.

On the other hadd, sooner or later one does have to execute computer instructions, and there has not as yet been found an algorithm to take one from abstract specification (59) to computer instructions, Moreover, a new constraint has been introduced. The selection of operations that are to be provided cannot be but influenced by the ease with which the corresponding a axiomatic specification can be set up. So, in (59) the sequence of operations peek and pop has replaced what we have traditionally understood by the pop operation.

The last observation suggests that a system should be made to provide an operation that does combine the two, and, in general, that a superstructure can be created on top of the axiomatic definition, where, because we are composing operations whose properties are preciesly known, a lot of dependability is built in. Otherwise we could not cope with anything as complicated as our 12 or 18 binary tree traversals.

A program is now expressed in terms of abstract data structures, and mappings have to be provided for going from this level to that of the storage structures. The mappings must certainly be correctness preserving, and one would hope that they would be complexity preserving as well. Moreover, the operations cannot

now be regarded in isolation. For example, if the program based on a tree traversal requires examination of the path from root to active node, then threads would not provide an efficient means for driving the traversal. Suppose the parent of a node is to be found. In our algorithms, n order that O(m) complexity be maintained, this has to be done in constant time. Again, if threads were used, this determination could cost O(n), where n is the number of D-nodes.

FURTHER READING

Texts on recurrence relations:

S. Goldberg, Introduction to Difference Equations, Wiley, New York, 1958.

H. Levy and F. Lessman, Finite Difference Equations, Macmillan, New York, 1961.

Good introduction to generating functions and recurrences:

C. L. Liu, Introduction to Combinatorial Mathematics, McGraw-Hill, New York, 1968.

Advanced text on recurrences:

J. Aczel, Lectures on Functional Equations and Their Applications, Academic Press, New York, 1966.

Balancing of AVL trees after a deletion is treated in:

E. M. Reingold, J. Nievergelt, and N. Deo, Combinatorial Algorithms: Theory and Practice, Prentice-Hall, Englewood Cliffs, N.J., 1977.

A proof of the cycles algorithm can be found in:

A. T. Berztiss, Data Structures: Theory and Practice, 2nd Ed., Academic Press, New York, 1975.

An excellent example of how one programs in a definitional and a operational environment is provided by Codd's relational algebra and relational calculus in:

C. J. Date, An Introduction to Database Systems, Addison-Wesley, Reading, Mass., 1975.