BIT 26 (1986), 266–276

A TAXONOMY OF BINARY TREE TRAVERSALS

ALFS BERZTISS

Department of Computer Science, University of Pittsburgh, Pittsburgh, PA 15260, USA

Abstract.

A survey of the literature shows that eleven binary tree traversals have been defined. We systematize this work by proposing a classification that consists of twenty-six traversals grouped into seven categories. Three generator schemas are provided that allow all of the traversals to be implemented.

CR categories: G.2.2.

Keywords and phrases: Binary tree, generator, program schema, traversal, tree.

1. Introduction.

The binary tree is a most interesting data structure, both in its own right and on account of its links to directed graphs. The directed graph serves as a model for any kind of relation, but digraph algorithms are commonly expressed in terms of implicit [21] or explicit [4] directed trees, and the latter can be converted to binary trees by a Knuth transformation [12]. This has resulted in the binary tree having been extensively studied.

However, no systematic classification of binary tree traversals appears to have been undertaken. There are $n!$ permutations of the nodes of a binary tree of n nodes, and each such permutation represents a different traversal of the binary tree, but we are only interested in traversals that can be described so that the length of the description is independent of n. In Section 2 we identify 26 such traversals, and group them into seven classes. In Section 3 the literature on binary tree traversals is surveyed. We find that eleven kinds of traversals of binary trees are either recorded in the literature, or they represent the Knuth transform equivalents of general trees that are mentioned in the literature. These eleven kinds of traversals belong to six of our seven classes.

Section 4 contains three traversal schemas that suffice to specify programs for all our traversals. Generators, such as are provided by Icon [9] are a good implementation tool for the traversals. In Section 5 we discuss the implications of our observations on the specification of data structures.

Received April 1986.

2. Traversals of trees and binary trees.

A rooted directed tree in which no more than two arcs originate from a node is a binary tree. Whenever two arcs originate from a node, one of the arcs goes to the left and the other to the right. Moreover, in a binary tree, even if just one arc originates from a node, a left or right orientation is still imposed on this arc.

Every rooted directed tree can be transformed into a binary tree by means of the following procedure, which has become known as the *Knuth transformation* [12]:

> Denote the arcs originating at internal node x in a tree in left to right order by $\langle x, y_1\rangle$, $\langle x, y_2\rangle$, ..., $\langle x, y_t\rangle$. Assign left orientation to $\langle x, y_1\rangle$, and replace the remaining arcs by $\langle y_1, y_2\rangle$, $\langle y_2, y_3\rangle$, ..., $\langle y_{t-1}, y_t\rangle$, 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.

Figure 1 shows a tree and Figure 2 its Knuth transform.

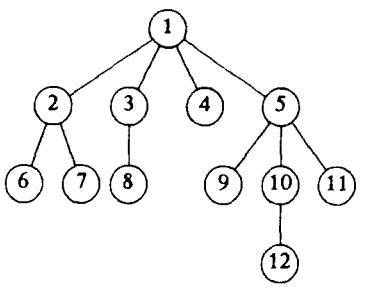


Fig. 1. Rooted tree.

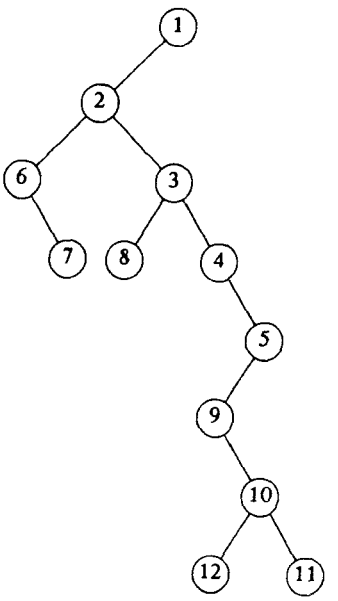


Fig. 2. Knuth transform of 1.

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 Figure 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

Let us define 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 Figure 2 we have, for example,

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

Note also that

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

i.e. that R(post) = C(pre). Similarly, R(in) = C(in), and R(pre) = C(post). This relation between the three classical traversal disciplines and their converses is shown by the diagrams of Figure 3. Reference to *C(preorder)* is made by Knuth [12], and to converses of all three classical traversal disciplines by Tremblay and Sorenson [23].

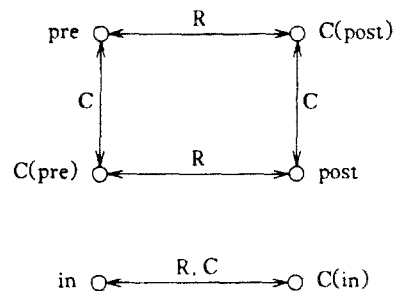


Fig. 3. Relation between different traversals.

The definitions of preorder and postorder are easily extended to arbitrary trees. Preorder traversal: At each node of the tree, process the node and then proceed to the successors of the node in left to right order, and analogously for postorder. For the tree of Figure 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

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 by a type indicator *LR*, and one with the opposite direction by *RL*. We then have, for the tree of Figure 1,

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

An LR (or RL) levelorder traversal can be implemented as follows: Process and push down the root of the tree on a queue; thereafter, while the queue is not empty, pop up a node, and process and push down its children in left to right (right to left) order. Now, if a stack is substituted for the queue, this procedure defines two further disciplines, which we respectively designate *LR-stackorder* and *RL-stackorder*. For the tree of Figure 1 we have

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

(The observation that different traversal orders can be obtained with the same procedure, depending on whether a stack or a queue is used, appears to have been first made by Pawlak [18].)

Let us now see what these traversals become in the Knuth transform. It is well known that preorder traversal is invariant under the transformation, and that T-post is equivalent to inorder traversal in the transform. In the equivalent to T-LR-level we are moving diagonally down from upper left to lower right (LR-down). The equivalent of T-RL-level is RL-up. Traversal T-LR-stack becomes reverse postorder in the transform, but there is no simple characterization of the traversal equivalent to T-RL-stack – we designate it RL-upstack. Then

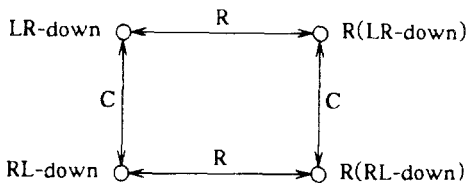
LR-down	1	2	3	4	5	6	7	8	9	10	11	12
RL-up	1	5	4	3	2	11	10	9	8	7	6	12
RL-upstack	1	5	4	3	2	7	6	8	11	10	9	12

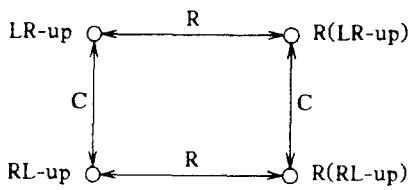
Levelorders and stackorders can, of course, be applied to a binary tree in its own right. We obtain

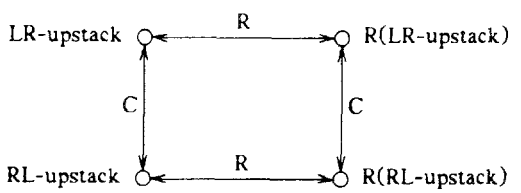
LR-level	1	2	6	3	7	8	4	5	9	10	12	11
LR-stack	1	2	6	3	8	4	5	9	10	12	11	7

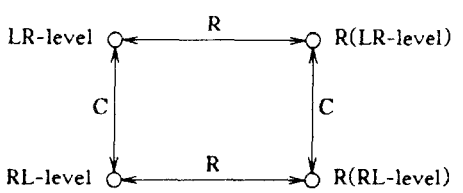
Now, if we apply operators C and R to the five traversals just defined, we obtain in each case three new traversals, as shown in Figure 4. Thus, Figures 3 and 4 depict relationships between 26 distinct traversals of binary trees. The converses of the five traversals defined above may not be obvious. They are therefore written out in full:

RL-down	1	2	6	3	8	7	4	5	9	10	12	11
LR-up	6	2	1	7	8	3	4	9	5	12	10	11
LR-upstack	6	2	1	8	3	7	4	9	5	12	10	11
RL-level	1	2	3	6	4	8	7	5	9	10	11	12
RL-stack	1	2	3	6	7	4	8	5	9	10	11	12









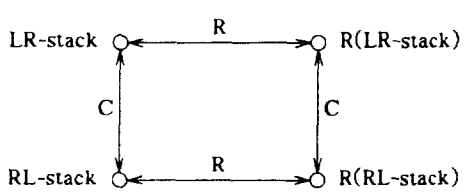


Fig. 4. Further traversals of binary trees.

3. Significance of binary tree traversals.

Traversals of trees and binary trees have been studied in the following contexts: (i) The information stored in a binary search tree is retrieved in lexicographical order when the binary tree is traversed under inorder (see, e.g. [13]). Somewhat similarly, suppose one has implemented a priority queue as a heap [13], that the heap is stored first as a binary tree, and that it is to be converted to vector form. This would be done by means of an LR-levelorder traversal of the binary tree. (ii) If a binary tree is used to represent an expression composed of binary operations, then the classical preorder and postorder traversals of the tree produce the expression in prefix and postfix forms, respectively (see, e.g., [3]). In this same context Pawlak [18] has investigated LR-levelorder, LR-stackorder, and their "duals", which in our terminology become R(RL-level) and R(RL-stack), respectively. This work was extended by Blikle [6] to include RL-downorder as well (some proofs of [6] have been simplified by Levy [15]). (iii) The correspondences between trees and formulas as representatives of composite expressions has led to an investigation of linear representations of trees in general. Here one should note again the work of Blikle [6], and that of Meyers [17]. (iv) In a study of marking algorithms for garbage collection Thorelli [22] makes use of T-LR-levelorder. (v) Berztiss [4] has investigated the conversion of digraphs into trees, and the expression of depth-first search algorithms for digraphs in terms of traversals of the corresponding trees. The algorithms make use of T-preorder, T-postorder, and R(T-postorder). For the exploration of digraphs in breadth-first manner one would make use of T-LR-levelorder or T-RL-levelorder.

The Knuth transformation enables us to convert trees into binary trees. Consequently we need only consider traversals of binary trees, and we have just determined that the following binary tree traversals have been found worthy of study, in their own right or because of the equivalence to them of certain traversals of arbitrary trees:

1. preorder			postorder
2. inorder	R(inorder)		
3. LR-down		RL-down	
4.		RL-up	
5.			
6. LR-level			R(RL-level)
7. LR-stack			R(RL-stack)

These traversals belong to six of the classes of Figures 3 and 4. Only the upstack class seems not to have attracted interest as yet.

4. Schemas for binary tree traversals.

In what follows we shall make use of three data types, *bintree* (short for binary tree), *stack*, and *queue*. For type *bintree* we shall require functions *left* and *right*, which return the left or right subtree, respectively, of the root node of the binary tree supplied as an argument to the function, the function *data*, which returns the data associated with the root node, and *isnull*, which returns false or true depending on whether the binary tree supplied as argument is empty or not. We shall assume the data associated with the nodes of the binary tree to be of type *xtype*.

The stack and queue operations have the same names. We shall therefore refer to stacks and queues collectively as pd-stores. Function *new* returns an empty pd-store, function *push* with arguments *pd* and *x* returns pd-store *pd* with item *x* added to it, function *read* returns the accessible element of the pd-store, leaving the pd-store unchanged, and *pop* returns the pd-store with the accessible element removed from it.

It has become customary to specify data types in an abstract manner by means of algebraic equations [8, 11]. It is then assumed that the data types will be used in a functional programming environment. Here we leave open the mode of precise specification of the data types, and assume a procedural programming style. In other words, we shall make free use of assignments. However, all our operations are pure functions without side effects.

The traversal sequences will be delivered by generators defined in terms of the operations listed above. A number of modern programming languages provide a generator facility [9, 14, 16, 19, 20]. However, in all except Icon [9], the generator is tied to a for-loop, and it becomes impossible to synchronize several cooperating generators. A solution to generator synchronization is advanced in [5], and the Icon design also allows synchronization.

Execution of a procedure that invokes a generator is intermeshed with execution of the generator. When the invoking procedure requests the generator to deliver a datum, the latter generates this datum, then suspends operation and returns to the calling procedure. The point from which the return is made will be indicated by a *suspend* statement. Operation is resumed from this point on the next invocation of the generator. A generator is therefore a specialized coroutine [2].

In what follows we shall assume the syntax and semantics of Pascal as far as practicable. For simplicity we shall assume that our generators are tied to a for-loop, and exit from the loop is to coincide with termination of the generator. Our generators will take the form of generator schemas, i.e., the generator of one traversal sequence will be transformable into that of another by substitution of components. Such an approach has been taken by Adams [1] and Burkhard [7] in their formulation of algorithms for the three classical binary tree traversal disciplines.

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

```
generator traverse(T: treetype, item: xtype);
begin {X, Y, Z} end;
```

where

```
X: item := data(T); suspend;
Y: if not isnull(left(T)) then traverse(left(T));
Z: if not isnull(right(T)) then traverse(right(T));
```

Recursion can be removed from *traverse* by the introduction of an explicit stack, and the schema that results refined specifically into a generator for preorder, say. The idea behind the preorder generator is to move down the tree by means of pointers, saving right pointers on a stack *pds*. However, a much richer variety of possibilities is obtained by introducing two pushdown stores. Now, instead of delivering data directly a pointer is taken off *pds* or reached in tracing left pointer chains, the data are saved up in a second pd-store.

```
generator preorder(T: treetype, item: xtype);
var   subtree: treetype;
      pds1: pushdowntypeA;
      pds2: pushdowntypeB;
begin
  pds1:= new; pds2:= new;
  pds1:= push(pds1, T);
  repeat
    subtree:= read(pds1); pds1:= pop(pds1);
    repeat
      pds2:= push(pds2, subtree);
      if not isnull(right(subtree)) then
        pds1:= push(pds1, right(subtree));
      subtree:= left(subtree)
    until isnull(subtree);
    repeat
      subtree:= read(pds2); pds2:= pop(pds2);
      item:= data(subtree)
      suspend
    until empty(pds2)
  until empty(pds1)
end;
```

The behavior of *preorder* is expressed in the following table:

pds1	*pds2*	Traversal
stack	queue	preorder
stack	stack	LR-uporder
queue	queue	RL-downorder
queue	stack	LR-upstack

Schemas *traverse* and *preorder* give one corner of every diagram in Figures 3 and 4, except for the ones containing RL-stackorder and RL-levelorder. We therefore propose a further schema:

```
generator stackorder(T: treetype, item: xtype);
var   subtree: treetype;
      pds: pushdowntype;
begin
  pds:= new;
  pds:= push(pds, T);
  item:= data(T);
  suspend;
  repeat
    subtree:= read(pds); pds:= pop(pds);
    if not isnull(right(subtree)) then
    begin
      pds:= push(pds, right(subtree));
      item:= data(right(subtree));
      suspend
    end;
    if not isnull(left(subtree)) then
    begin
      pds:= push(pds, left(subtree));
      item:= data(left(subtree));
      suspend
    end;
  until empty(pds)
end;
```

This generator produces RL-stackorder when *pds* is a stack, and RL-levelorder when it is a queue. This means that all seven diagrams can be covered by *traverse*, *preorder*, and *stackorder*. Indeed, *traverse* generates all six traversals of Figure 3.

As regards Figure 4, we have generators for five basic disciplines, and the remaining 15 traversals can be derived from these five by means of operators R and C. Here it should be noted that the two operators are different: C operates on an algorithm (it interchanges the meaning of left and right); R operates on the node sequence produced by an algorithm. This implies that the operators can be composed in only one way: $R(C(t))$ is meaningful, but $C(R(t))$ is not. The most convenient implementation of $R(t)$ is to produce sequence t, and then to reverse it. Gries and Gehani [10] suggest that programming languages should provide an operator such as our R as a standard feature.

5. Final remarks.

We have classified 26 traversals, and defined generator schemas that may be used (in principle) to produce any of the traversal sequences. Of course, these traversals do not exhaust all possibilities. However, we feel that any traversal discipline outside our seven classes will become increasingly difficult to describe. Indeed, only the six classical traversals can be easily described because they are the only ones for which a straightforward recursive specification exists. In describing the traversals of Figure 4, we either had to appeal to drawings of binary trees or we had to produce a generator.

This has serious implications for program verification. Verification consists of taking a formal specification and showing a program to be consistent with the specification in some well defined system of inference. As regards the data type of binary tree, one has to specify a traversal formally, and show that a program implements the traversal as specified. The binary tree is quite well understood in a mathematical sense. Therefore, if any operations are to be amenable to formal specification, one would expect operations on a binary tree to be among the best candidates. However, we have shown that there exist traversals that have practical significance, but are difficult to specify.

REFERENCES

1. E. N. Adams, *Another representation of binary tree traversal*, Inf. Proc. Letters 2 (1973), 52–54.
2. R. R. Atkinson, B. H. Liskov and R. W. Scheifler, *Aspects of implementing CLU*, Proc. ACM Annual Conf., Washington DC, 1978, pp. 123–129.
3. A. T. Berztiss, *Data Structures: Theory and Practice*, 2nd ed. Academic Press, New York, 1975.
4. A. T. Berztiss, *Depth-first K-trees and critical path analysis*, Acta Inf. 13 (1980), 325–346.
5. A. T. Berztiss, *Data abstraction, controlled iteration, and communicating processes*, Proc. ACM Annual Conf., Nashville TN, 1980, pp. 197–203.
6. A. Blikle, *Addressless units for carrying on loop-free computations*, J. ACM 19 (1972), 136–157.
7. W. A. Burkhard, *Nonrecursive tree traversal algorithms*, Computer J. 18 (1975), 227–230.
8. J. A. Goguen, J. W. Thatcher and E. G. Wagner, *An initial algebra approach to the specification, correctness, and implementation of abstract data types*, in Current Trends in Programming Methodology, Vol. 4: Data Structuring (R. T. Yeh, ed.). Prentice-Hall, Englewood Cliffs NJ, 1978, pp. 80–149.
9. R. E. Griswold and M. T. Griswold, *The Icon Programming Language*. Prentice-Hall, Englewood Cliffs NJ, 1983.

10. D. Gries and N. Gehani, *Some ideas on data types in high-level languages*, Comm. ACM 20 (1977), 414–420.
11. J. V. Guttag, E. Horowitz and D. R. Musser, *The design of data type specifications*, in Current Trends in Programming Methodology, Vol. 4: Data Structuring (R. T. Yeh, ed.). Prentice-Hall, Englewood Cliffs NJ, 1978, pp. 60–79.
12. D. E. Knuth, *Tha Art of Computer Programming, Vol. 1: Fundamental Algorithms*. Addison-Wesley, Reading MA, 1968, p. 329 (Exercise 4).
13. D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*. Addison-Wesley, Reading MA, 1973.
14. B. W. Lampson, J. J. Horning, R. L. London, J. G. Mitchell and G. J. Popek, *Report on the programming language Euclid*, ACM SIGPLAN Notices 12, 2 (Feb. 1977).
15. L. S. Levy, *Ordering of trees – simplified proofs*, Proc. 7th Ann. Princeton Conf. Inf. Sciences and Systems 1973, 400–402.
16. B. Liskov, R. Atkinson, T. Bloom, E. Moss, J. C. Schaffert, R. Scheifler and A. Snyder, *CLU Reference Manual*. Springer-Verlag LNCS 114, Springer-Verlag, Berlin, 1981.
17. W. J. Meyers, *Linear representation of tree structure*, Proc. 3rd Ann. ACM Symp. Theory Computing 1971, 50–62.
18. Z. Pawlak, *New class of mathematical languagues and organization of addressless computers*. Proc. Colloq. Foundations Math., Math. Machines and their Applications, Tihany, Hungary, 1962, 227–238.
19. L. A. Rowe and K. A. Shoens, *Data abstraction, views and updates in Rigel*, Proc. ACM SIGMOD 1979 Internat. Conf. Management Data, 71–81.
20. M. Shaw (ed.), *Alphard: Form and Content*. Springer-Verlag, New York, 1981.
21. R. Tarjan, *Depth-first search and linear graph algorithms*, SIAM J. Comput 1 (1972), 146–160.
22. L. E. Thorelli, *Marking algorithms*, BIT 12 (1972), 555–568.
23. J. P. Tremblay and P. G. Sorenson, *An Introduction to Data Structures with Applications*. McGraw-Hill, New York, 1976.