On Constructing a Binary Tree from Its Traversals

Xiang, Limin
Department of Computer Science and CommunicationEngineering, Kyushu University

Lawi, Armin
Department of Computer Science and CommunicationEngineering, Kyushu University : Graduate Student

Ushijima, Kazuo
Department of Computer Science and Communication Engineering, Graduate School of Information Science and Electrical Engineering, Kyushu University

https://doi.org/10.15017/1500431

出版情報：九州大学大学院システム情報科学紀要. 5 (1), pp.13-18, 2000-03-24. 九州大学大学院システム情報科学研究院

九州大学大学院
システム情報科学研究科報告
第5巻 第1号 平成12年3月

Research Reports on Information Science and
Electrical Engineering of Kyushu University
Vol.5, No.1, March 2000

On Constructing a Binary Tree from Its Traversals

Limin XIANG* , Armin LAWI** and Kazuo USHIJIMA*

(Received December 9, 1999)

Abstract: Many algorithms have been presented for constructing a binary tree from its traversals. This problem can be solved by a sequential algorithm of linear time, such as E. Makinen's and A. Andersson and S. Carlsson's algorithms. If the number of comparison operations is used as a measure of time complexity, the linear coefficient of E. Makinen's is 3 in its best case and 5 in its worst case, and that of A. Andersson and S. Carlsson's is 4 in its best case and 7 in its worst case. In this article, we give a more efficient sequential algorithm for the problem, and the linear coefficient is 3 in any case.

Keywords: Data structures, Binary trees, Tree traversals, Tree construction, Design of algorithms, Analysis of algorithms

1. Introduction

"If we are given the preorder and the inorder of the nodes of a binary tree, the binary tree structure may be constructed[[9]p.331]." "Postorder and inorder together characterize the structure. But preorder and postorder do not[[9]p.564]." Up to now, many sequential and parallel algorithms have been presented for constructing a binary tree from its traversals[1),2),3),5),7),10),11),13),14),15)]. H. A. Burgdorff et al.[2)] presented an $O(n^2)$ solution, G.H.Chen et al.[3)] and W. Slough and K. Efe[13)] gave $O(n\log n)$ methods respectively, E. Makinen[10)] and A. Andersson and S. Carlsson[1)] raised $O(n)$ algorithms respectively, and other researchers paid attention to parallel algorithms for this problem on EREW PRAM[5),7)], CREW PRAM[11)], CRCW PRAM[15)] and BSR[14)].

Of all the sequential algorithms for the problem of constructing a binary tree from its traversals (the CBTfIT problem for short), E. Makinen's and A. Andersson and S. Carlsson's linear algorithms are the best solutions. In any sequential algorithm for the CBTfIT problem, the binary tree is constructed mainly by comparing its traversals and checking integer variables for the recurrent control. Therefore, the number of comparison operations is used as a measure of time complexity in this article. Thus, for the CBTfIT problem with n nodes in a tree, E. Makinen's algorithm is of $3n-2$ complexity in its best case and of $5n-5+(1-(-1)^n)/2$ complexity in its worst case, and A. Andersson and S. Carlsson's algorithm is of $4n$ complexity in its best case and of $7n-3$ complexity in its worst case.

By discussing properties further for the traversals of a binary tree, we obtain some new results, and based on them we propose a more efficient sequential algorithm for the CBTfIT problem, which is of $3n-2$ complexity in its best case and of $3n-1$ complexity in its worst case.

2. Properties

In the same way as in most literatures we assume that the nodes of a binary tree are labeled with distinct alphanumeric labels. Thus, the traversal (*preorder*, *inorder* or *postorder*) of a binary tree with n nodes is a sequence with n distinct alphanumeric labels. We denote the set of n distinct alphanumeric labels by Σ_n and the set of all the permutations on set Σ_n by $\mathcal{P}(\Sigma_n)$, then we can discuss the CBTfIT problem in the relation category.

Definition 1.

(1) If $\mathcal{A} \in \mathcal{P}(\Sigma_n)$, then $\mathcal{A}$ is called a sequence, and $\mathcal{A}^c$ is the converse of $\mathcal{A}$ (obviously $\mathcal{A}^c \in \mathcal{P}(\Sigma_n)$.)

(2) $\mathbf{PTS}_n$ is a relation on $\mathcal{P}(\Sigma_n)$, s.t., $<\mathcal{A},\mathcal{B}>\in \mathbf{PTS}_n$, denoted by $\mathbf{PTS}_n(\mathcal{A},\mathcal{B})$, if and only if $\mathcal{A}$, $\mathcal{B} \in \mathcal{P}(\Sigma_n)$, and $\mathcal{B}$ can be obtained by **P**assing $\mathcal{A}$ **T**hrough a **S**tack.

(3) $\mathbf{piT}_n$ is a relation on $\mathcal{P}(\Sigma_n)$, s.t., $<\mathcal{A},\mathcal{B}>\in \mathbf{piT}_n$, denoted by $\mathbf{piT}_n(\mathcal{A},\mathcal{B})$, if and only if $\mathcal{A}$, $\mathcal{B} \in \mathcal{P}(\Sigma_n)$, and $\mathcal{A}$ and $\mathcal{B}$ may be the **P**reorder and **I**norder of a binary **T**ree respectively. The binary tree with the preorder A and the inorder B is denoted by $\mathbf{T}_{pi}(\mathcal{A},\mathcal{B})$.

(4) $\mathbf{ipT}_n$ is a relation on $\mathcal{P}(\Sigma_n)$, s.t., $<\mathcal{A},\mathcal{B}>\in$

* Department of Computer Science and Communication Engineering
** Department of Computer Science and Communication Engineering, Graduate Student

$\mathbf{ipT}_n$, denoted by $\mathbf{ipT}_n(\mathcal{A},\mathcal{B})$, if and only if $\mathcal{A}$, $\mathcal{B} \in \mathcal{P}(\Sigma_n)$, and $\mathcal{A}$ and $\mathcal{B}$ may be the **I**norder and **P**ostorder of a binary **T**ree respectively.

Thus, the result on [[9]p.331] can be expressed as "$\mathbf{piT}_n(\mathcal{A},\mathcal{B}) \Leftrightarrow \mathbf{PTS}_n(\mathcal{A},\mathcal{B})$" that sequences $\mathcal{A}$ and $\mathcal{B}$ may be the preorder and inorder traversals of a binary tree respectively if and only if sequence $\mathcal{B}$ can be obtained by passing sequence $\mathcal{A}$ through a stack. New properties on $\mathbf{PTS}_n$, $\mathbf{piT}_n$ and $\mathbf{ipT}_n$ can be given as follows.

Theorem 2.

(1) When $n = 1, 2$, $\mathbf{PTS}_n$ is equivalent.

(2) When $n \geq 3$,

(a) $\mathbf{PTS}_n$ is reflexive;

(b) $\mathbf{PTS}_n$ is neither symmetric nor anti-symmetric;

(c) $\mathbf{PTS}_n$ is not transitive.

(3) $\mathbf{ipT}_n(\mathcal{A},\mathcal{B}) \Leftrightarrow \mathbf{PTS}_n(\mathcal{A},\mathcal{B})$.

(4) $\mathbf{PTS}_n(\mathcal{A},\mathcal{B}) \Leftrightarrow \mathbf{PTS}_n(\mathcal{B}^c,\mathcal{A}^c)$.

Proof. Since proofs for **(1)** and **(2)** are simple, they are omitted here. Proof for **(3)** can be obtained easily, similar to that for "$\mathbf{piT}_n(\mathcal{A},\mathcal{B}) \Leftrightarrow \mathbf{PTS}_n(\mathcal{A},\mathcal{B})$" [[9]p.564]. Therefore, only the proof for **(4)** is given as follows.

If $\mathbf{PTS}_n(\mathcal{A},\mathcal{B})$, i.e., sequence $\mathcal{B}$ with n elements can be obtained by passing $\mathcal{A}$ through a stack, then the operations for passing $\mathcal{A}$ through a stack can be described by an *admissible sequence* of n **S**'s and n **X**'s[[9]pp.242-243], where **S** stands for *moving an element from the input into the stack*, and **X** stands for *moving an element from the stack into the output.* "An admissible sequence is one in which the number of **X**'s never exceeds the number of **S**'s if we read from the left to the right[[9]p.536]." Let $\mathcal{S}_{SX}$ be the admissible sequence for $\mathbf{PTS}_n(\mathcal{A},\mathcal{B})$, since "no two different admissible sequences give the same output permutation [[9]p.243]," $\mathcal{S}_{SX}$ is the only one admissible sequence for $\mathbf{PTS}_n(\mathcal{A},\mathcal{B})$. If we exchange all **S**'s for all **X**'s in $\mathcal{S}^c_{SX}$, i.e., the converse sequence of $\mathcal{S}_{SX}$, to get a new sequence of n **S**'s and n **X**'s which we denote by $\mathcal{E}(\mathcal{S}^c_{SX})$, then $\mathcal{E}(\mathcal{S}^c_{SX})$ is an admissible sequence, too, and $\mathcal{E}(\mathcal{S}^c_{SX})$ is just the admissible sequence for $\mathbf{PTS}_n(\mathcal{B}^c,\mathcal{A}^c)$. This completes the proof. □

Corollary 3.

(1) $\mathbf{piT}_n(\mathcal{A},\mathcal{B}) \Leftrightarrow \mathbf{PTS}_n(\mathcal{B}^c,\mathcal{A}^c)$.

(2) $\mathbf{ipT}_n(\mathcal{A},\mathcal{B}) \Leftrightarrow \mathbf{PTS}_n(\mathcal{B}^c,\mathcal{A}^c)$.

Proof.

(1). From $\mathbf{piT}_n(\mathcal{A},\mathcal{B}) \Leftrightarrow \mathbf{PTS}_n(\mathcal{A},\mathcal{B})$ and **Theorem 2.(4)**.

(2). From **Theorem 2.(3)** and **(4)**. □

3. Algorithm

From **corollary 3.(1)**, we know that the preorder $\mathcal{A}$ can be regarded as the sequence formed by using a stack to change the order of elements in the inorder $\mathcal{B}$ conversely. In other words, we can design a match algorithm in which the inorder $\mathcal{B}$ is to be scanned conversely, and the order of elements in the inorder $\mathcal{B}$ is to be changed by a stack in order to match elements in the preorder $\mathcal{A}$ conversely. Such a match algorithm can be described as follows.

Algorithm Match

```
procedure Match;
begin
  push α to a stack as the bottom element;
  push inorder[n] to the stack;
  inindex:=n − 1;
  preindex:=n;
  while (preindex> 1) do
  begin
    while (top≠preorder[preindex]) do
    begin
      push inorder[inindex] to the stack;
      inindex:=inindex−1;
    end;
    pop the top element;
    preindex:=preindex−1;
  end;
  if (inindex= 1) then
  begin
    { inorder[1] matches preorder[1], }
    { inorder[1] (preorder[1]) is the label of the root, and }
    { the root has not the left subtree. }
  end else
  begin
    { the top element matches preorder[1] }
    { the top element (preorder[1]) is the label of the }
    { root, and the root has the left subtree. }
    pop the top element;
  end;
  pop the top element (α) ;
end.
```

Since the preorder $\mathcal{A}$ and the inorder $\mathcal{B}$ can be expressed as $\mathcal{A} = r\mathcal{A}^L\mathcal{A}^R$ and $\mathcal{B} = \mathcal{B}^L r \mathcal{B}^R$ respectively, where, r is the label of the root, and $\mathbf{T}_{pi}(\mathcal{A}^L,\mathcal{B}^L)$ and $\mathbf{T}_{pi}(\mathcal{A}^R,\mathcal{B}^R)$ are the left and right subtrees of r respectively, the algorithm **Match** can be understood as the following steps.

1. Using the stack to match $\mathcal{B}^R$ with $\mathcal{A}^R$;
2. pushing r of $\mathcal{B}$ to the stack;
3. using the stack with the bottom element r to match $\mathcal{B}^L$ with $\mathcal{A}^L$;
4. popping r from the stack to match r of $\mathcal{A}$.

Further, we know in the algorithm **Match** that

1. when an element b_i of $\mathcal{B}$ is pushed to the stack, the right-subtree of b_i can be determined;
2. if b_i is popped as soon as it is pushed to the stack, the left-subtree of b_i is empty;
3. if b_i is popped as soon as b_j is popped, b_j is the left-son of b_i;
4. when b_i is popped from the stack, the subtree with the root b_i can be determined.

Therefore, by modifying the algorithm **Match**, we can obtain an algorithm using a *stack* to construct a binary tree from its traversals. If we pay attention to that the *left pointer* of a node is not used after the node is created until the node is popped from the *stack*, we can only use a *pointer* **VirtualStack** instead of the *stack*. Thus, we obtain the following algorithm **ConstructTree**.

Algorithm ConstructTree

```
procedure ConstructTree;
begin
   VirtualStack:=CreateNode(α);
   CurrentNode:=CreateNode(inorder[n]);
   CurrentNode↑.right:=nil;
   CurrentNode↑.left:=VirtualStack;
   VirtualStack:=CurrentNode;
   SubTree:=nil;   inindex:=n − 1;   preindex:=n;
   while (preindex> 1) do
   begin
      while (VirtualStack↑.label≠preorder[preindex]) do
      begin
         CurrentNode:=CreateNode(inorder[inindex]);
         CurrentNode↑.right:=SubTree;
         CurrentNode↑.left:=VirtualStack;
         VirtualStack:=CurrentNode;
         SubTree:=nil;
         inindex:=inindex−1;
      end;
      CurrentNode:=VirtualStack;
      VirtualStack:=CurrentNode↑.left;
      CurrentNode↑.left:=SubTree;
      SubTree:=CurrentNode;
      preindex:=preindex−1;
   end;
   if (inindex= 1) then
   begin
      root:=CreateNode(inorder[1]);
      root↑.left:=nil;
      root↑.right:=SubTree;
   end else
   begin
      root:=VirtualStack;
      VirtualStack:=root↑.left;
      root↑.left:=SubTree;
   end;
   dispose(VirtualStack); { i.e., α }
end.
```

Similarly, based on $\mathbf{piT}_n(A,B) \Leftrightarrow \mathbf{PTS}_n(A,B)$, $\mathbf{ipT}_n(A,B) \Leftrightarrow \mathbf{PTS}_n(A,B)$ or $\mathbf{ipT}_n(A,B) \Leftrightarrow \mathbf{PTS}_n(B^c,A^c)$, the corresponding algorithms can be easily obtained for the match and the tree construction.

4. Analysis and contrast

The number of comparison operations is used as a measure of time complexity for the CBTfIT problem in this article. In this section, the algorithm **ConstructTree** is analyzed and contrasted with the best two previous sequential algorithms, i.e., A. Andersson and S. Carlsson's and E. Makinen's.

4.1 Lower and upper bounds

(1) A. Andersson and S. Carlsson's Algorithm[1)]

The algorithm is shown in **Appendix B**. There are 5 comparisons in turn, i.e.,

1. C_{e1} : not Empty(IN),
2. C_{d1} : current↑.data≠First(IN),
3. C_{e2} : not Empty(IN),
4. C_{nl} : current↑.right=nil, and
5. C_{d2} : First(IN)≠current↑.right.data .

C_{e1}, C_{e2} and C_{nl} can be implemented simply by *integer* comparisons, and C_{d1} and C_{d2} are *label* comparisons. The best case for the algorithm is that the binary tree to be constructed is a *rightchain tree*[16)], and in the case the number for *integer* comparisons is $3n$ and the number for *label* comparisons is n, while the worst case for the algorithm is that the binary tree is a *leftchain tree*[7),16)], and in the case the number for *integer* comparisons is $4n-1$ and the number for *label* comparisons is $3n-2$. In the general, A. Andersson and S. Carlsson's Algorithm needs $4n$ comparisons in its best case and $7n-3$ comparisons in its worst case.

(2) E. Makinen's Algorithm[10)]

The algorithm is shown in **Appendix C**. There are 6 comparisons in turn, i.e.,

1. C_{i1} : preindex< n,
2. C_{d1} : preorder[preindex]=inorder[inindex],
3. C_{d2} : inorder[inindex]≠top↑.label,
4. C_{d3} : inorder[inindex]=top↑.label,
5. C_{d4} : inorder[inindex]=top↑.label, and
6. C_{i2} : preindex≤ n.

C_{i1} and C_{i2} are *integer* comparisons, and C_{d1}, C_{d2}, C_{d3} and C_{d4} are *label* comparisons. The best case for the algorithm is that the binary tree to be constructed is a *leftchain tree*, and in the case the number for *integer* comparisons is n and the number for *label* comparisons is $2n-2$, while the worst case

Table 1 Data for average linear coefficients

n	C_n	AC_n	M_n	XU_n	$ALCac_n$	$ALCm_n$	$ALCxu_n$	XU_n / AC_n	XU_n / M_n
1	1	4	1	2	4.0000000000	1.0000000000	2.0000000000	0.5000000000	2.0000000000
2	2	19	9	9	4.7500000000	2.2500000000	2.2500000000	0.4736842105	1.0000000000
3	5	76	43	38	5.0666666667	2.8666666667	2.5333333333	0.5000000000	0.8837209302
4	14	294	180	149	5.2500000000	3.2142857143	2.6607142857	0.5068027211	0.8277777778
5	42	1128	722	574	5.3714285714	3.4380952381	2.7333333333	0.5088652482	0.7950138504
6	132	4323	2847	2202	5.4583333333	3.5946969697	2.7803030303	0.5093684941	0.7734457323
7	429	16588	11143	8448	5.5238095238	3.7106227106	2.8131868132	0.5092838196	0.7581441264
8	1430	63778	43472	32461	5.5750000000	3.8000000000	2.8375000000	0.5089686099	0.7467105263
9	4862	245752	169390	124982	5.6161616162	3.8710635769	2.8562091503	0.5085696149	0.7378357636
10	16796	948974	659906	482222	5.6500000000	3.9289473684	2.8710526316	0.5081509082	0.7307434695
11	58786	3671864	2571726	1864356	5.6783216783	3.9770229770	2.8831168831	0.5077410274	0.7249434815
12	208012	14233964	10028504	7221634	5.7023809524	4.0175983437	2.8931159420	0.5073522738	0.7201107962
13	742900	55271760	39135972	28022188	5.7230769231	4.0523076923	2.9015384615	0.5069892473	0.7160212604
14	2674440	214958115	152851675	108909140	5.7410714286	4.0823412698	2.9087301587	0.5066528426	0.7125151883

for the algorithm is that the binary tree is a *full tree* (when n is odd) or a tree with only one internode without the left subtree (when n is even), and in the case the number for *integer* comparisons is $n + \lfloor (n-1)/2 \rfloor$ and the number for *label* comparisons is $3n - 3 + \lfloor (n-1)/2 \rfloor$. In the general, E. Makinen's Algorithm needs $3n - 2$ comparisons in its best case and $5n - 5 + (1 - (-1)^n)/2$ comparisons in its worst case.

(3) Our Algorithm

Theorem 4.

The algorithm **ConstructTree** needs $3n - 2$ comparisons in its best case and $3n - 1$ comparisons in its worst case.

Proof.

Note that in the algorithm **ConstructTree**,

1. there are only three clauses including comparison operations, i.e., the **while** clause with '(preindex> 1)', the **while** clause with '(VirtualStack↑.label≠preorder[preindex])' and the **if** clause with '(inindex= 1)';
2. the initial values of the variables *preindex* and *inindex* are n and $n-1$ respectively;
3. the variables *preindex* and *inindex* are subtracted by 1 for each time in the body of its **while** clause respectively;
4. the final value of the variable *preindex* is 1, while the final value of the variable *inindex* is 1 when preorder[1]=inorder[1] or 0 when preorder[1]≠inorder[1];
5. the **if** clause is executed for only one time.

Therefore, for the algorithm **ConstructTree**, the number of integer comparisons is $n+1$, and the number of label comparisons is $(n-1)+(n-2) = 2n-3$ when preorder[1]=inorder[1] or $(n-1)+(n-1) = 2n-2$ when preorder[1]≠inorder[1]. This completes the proof. □

4.2 Average linear coefficients

For the three linear algorithms above, we can create a table to show their average linear coefficients by

1. using an algorithm in 17) or 18) to enumerate binary trees,
2. obtaining its preorder and inorder traversals for each tree,
3. constructing the tree from the traversals by each of the three algorithms, and
4. adding up the number of comparisons.

Such a table is given in **Table 1**, where,

- C_n: the nth Catalan Number, i.e., the number of binary trees with n nodes[9],
- AC_n: the number of comparisons needed by A. Andersson and S. Carlsson's algorithm to construct all the binary trees with n nodes,
- M_n: the number of comparisons needed by E. Makinen's algorithm to construct all the binary trees with n nodes,
- XU_n: the number of comparisons needed by our algorithm to construct all the binary trees with n nodes,
- $ALCac_n$: the average linear coefficient of A. Andersson and S. Carlsson's algorithm, i.e., $(AC_n/C_n)/n$,
- $ALCm_n$: the average linear coefficient of E. Makinen's algorithm, i.e., $(M_n/C_n)/n$, and
- $ALCxu_n$: the average linear coefficient of our algorithm, i.e., $(XU_n/C_n)/n$.

From **Table 1**, it is known that

1. $XU_n < AC_n$ and $XU_n < M_n$ when $n > 2$,
2. $ALCac_n > 5.7$, $ALCm_n > 4$, and $ALCxu_n < 3$ ($\lim_{n\to\infty} ALCxu_n = 3$) when $n > 12$, and
3. XU_n/AC_n is about 50% and XU_n/M_n is less than 73% when $n > 10$.

5. Conclusion

The intimate relation has been revealed further between the stack and the binary tree, and more efficient sequential algorithm has been derived for the CBTfIT problem.

Algorithms in 2), 3) and 13) for computing the *inorder-preorder sequence* need $O(n^2)$ or $O(n\log n)$ time, while based on the intimate relation between the stack and the binary tree, an efficient linear algorithm can be obtained easily by modifying the Algorithm **Match**. Such an algorithm is given in **Appendix A**.

As for the efficient algorithm for computing the *preorder-inorder sequence*, it can be obtained by replacing

"ipSequence[preindex]:=top" and
"ipSequence[1]:=top" with
"piSequence[top]:=preindex" and
"piSequence[top]:=1" respectively

in **Appendix A**.

In the same way based on the intimate relation between the stack and the binary tree, a *binary bit-pattern*(or *bit-string*)[4),12),19)] representing a binary tree can be regarded as the admissible sequence for passing the preorder of the binary tree through a stack into the inorder of the binary tree, in which **1** stands for **S** and **0** stands for **X**.

References

1) A. Andersson and S. Carlsson, *Construction of a tree from its traversals in optimal time and space*, Inform. Process. Lett., 34 (1990), pp. 21-25.
2) H. A. Burgdorff, S. Jajodia, F. N. Springsteel and Y. Zalcstein, *Alternative methods for the reconstruction of trees from their traversals*, BIT, 27 (1987), pp. 134-140.
3) G. H. Chen, M. S. Yu and L. T. Liu, *Two algorithms for constructing a binary tree from its traversals*, Inform. Process. Lett., 28 (1988), pp. 297-299.
4) M. C. Er, *Enumerating Ordered Trees Lexicographically*, Comput. J., 28 (1985), pp. 538-542.
5) N. Gabrani and P. Shankar, *A note on the reconstruction of a binary tree from its traversals*, Inform. Process. Lett., 42 (1992), pp. 117-119.
6) T. Hikita, *Listing and counting subtrees of equal size of a binary tree*, Inform. Process. Lett., 17 (1983), pp. 225-229.
7) V. Kamakoti and C. P. Rangan, *An optimal algorithm for reconstructing a binary tree*, Inform. Process. Lett.,42 (1992), pp. 113-115.
8) G. D. Knott, *A numbering system for binary trees*, Comm. ACM, 20 (1977), pp. 113-115.
9) D. E. Knuth, *Fundamental Algorithms, The art of Computer Programming, vol.1 Third Edition*, Addison-Wesley, Reading Mass., 1997.
10) E. Makinen, *Constructing a binary tree from its traversals*, BIT, 29 (1989), pp. 572-575.
11) S. Olariu, M. Overstreet and Z. Wen, *Reconstructing a binary tree from its traversals in doubly logarithmic CREW time*, J. Parallel Distrib. Comput., 27 (1995), pp. 100-105.
12) A. Proskurowski, *On the Generation for Binary Trees*, J. ACM, 27 (1980), pp. 1-2.
13) W. Slough and K. Efe, *Efficient algorithms for tree reconstruction*, BIT, 29 (1989), pp. 361-363.
14) I. Stojmenovic, *Constant time BSR solutions to parenthesis matching, tree decoding, and tree reconstruction from its traversals*, IEEE Trans. Parallel and Distributed Systems, 7 (1996), pp. 218-224.
15) Z. Wen, *New algorithms for the LCA problem and the binary tree reconstruction problem*, Inform. Process. Lett., 51 (1994), pp. 11-16.
16) L. Xiang and K. Ushijima, *Properties on Leftchain Trees*, Research Reports on Information Science and Electrical Engineering of Kyushu University, ISSN 1342-3819, 2 (1997), pp. 9-13.
17) L. Xiang, C. Tang and K. Ushijima, *Grammar-oriented enumeration of binary trees*, Comput. J., 40 (1997), pp. 278-291.
18) L. Xiang and K. Ushijima, *Grammar-oriented enumeration of arbitrary trees and arbitrary k-ary trees*, IEICE Trans. Inf. & Sys., E82-D (1999), pp. 1245-1253.
19) S. Zaks, *Lexicographic Generation of Ordered Trees*, Theoretical Comput. Sci., 10 (1980), pp. 63-82.

Appendix A. Algorithm GETipSEQUENCE

```
procedure GETipSEQUENCE;
begin
  inorder[0]:=α;
  push 0 to a stack as the bottom element;
  push n to the stack;
  inindex:=n − 1;
  preindex:=n;
  while (preindex> 1) do
  begin
    while (inorder[top]≠preorder[preindex]) do
    begin
      push inindex to the stack;
      inindex:=inindex−1;
    end;
    ipSEQUENCE[preindex]:=top;
    pop the top element;
    preindex:=preindex−1;
  end;
  if (inindex= 1) then ipSEQUENCE[1]:=1 else
  begin
    ipSEQUENCE[1]:=top;
    pop the top element;
  end;
  pop the top element (0) ;
end.
```

Appendix B.

A. Andersson and S. Carlsson's algorithm[1) p.24]

```
procedure NonRecursive(var T: Tree; var PRE,
IN: NodeList);
var current, right-anc: Tree;
begin
  T:=CreateNode(First(PRE));
  Delete first element of PRE;
  current:=T;
  while not Empty(IN) do
    if current↑.data≠First(IN) then begin
      { current has a nonempty left subtree }
      right-anc:=current;
      current↑.left:=CreateNode(First(PRE));
      Delete first element of PRE;
      current:=current↑.left;
      current↑.right:=right-anc
    end
    else begin
      { current's left subtree has been constructed }
      Delete first element of IN;
      if not Empty(IN) and
      (current↑.right=nil or First(IN)≠
      current↑.right.data) then begin
        { current has a right subtree }
        right-anc:=current↑.right;
        current↑.right:=
        CreateNode(First(PRE));
        Delete first element of PRE;
        current:=current↑.right;
        current↑.right:=right-anc
      end
      else begin
        { current's right subtree is empty }
        right-anc:=current↑.right;
        current↑.right:=nil;
        current:=right-anc
      end
    end
end
```

Appendix C.

E. Makinen's algorithm[10) pp.574-575]

```
procedure TreeConstruction;
begin
  inindex := 1;
  preindex := 1;
  new(CurrentNode);
  CurrentNode ↑ .label := preorder[1];
  root := CurrentNode;
  while preindex < n do begin
    if preorder[preindex] = inorder[inindex]
    then begin
      preindex := preindex + 1;
      inindex := inindex + 1;
      if inorder[inindex] ≠ top ↑ .label
      then begin
        new(CurrentNode ↑ .right);
        CurrentNode := CurrentNode ↑ .right;
        CurrentNode ↑ .label := preorder[preindex] end
      end
      else begin
        preindex := preindex + 1;
        push a pointer to CurrentNode to the stack;
        new(CurrentNode ↑ .left);
        CurrentNode := CurrentNode ↑ .left;
        CurrentNode ↑ .label := preorder[preindex] end;
    if inorder[inindex] = top ↑ .label
    then begin
      while inorder[inindex] = top ↑ .label do begin
      CurrentNode := top;
      pop the top element from the stack;
      inindex := inindex + 1 end;
      if preindex ≤ n
      then begin
        new(CurrentNode ↑ .right);
        CurrentNode := CurrentNode ↑ .right;
        CurrentNode ↑ .label := preorder[preindex];end;
      end;
  end;{do}
end; {TreeConstruction}
```