The 7th Balkan Conference on Operational Research
"BACOR 05"
Constanta, May 2005, Romania

OPTIMAL TIME AND SPACE COMPLEXITY ALGORITHM FOR CONSTRUCTION OF ALL BINARY TREES FROM PRE-ORDER AND POST-ORDER TRAVERSALS

ADRIAN DEACONU

"Transilvania" University of Brasov, Romania

Abstract
A linear time and space algorithm for construction of a binary tree from the pre-order and post-order traversals is presented.
The solution is not always unique. The number of solutions is calculated and an optimal time and space method to find all the solutions is shown.

Keywords*: binary tree, traversals, construction of tree*

1. INTRODUCTION

The problem of construction the binary tree from the pre-order (or post-order) and in-order traversals is well-known [1][2][3][4][5]. The linear time and space was achieved. The case of pre-order and post-order is less studied [5][6]. The solution in this case is not even unique. We shall present an algorithm to find a solution in linear time and space and, starting from this one, we'll be able to find all the solutions in optimal time and space complexity.

The method in this paper for finding one solution is adapted from the algorithm for finding the tree from pre-order and post-order traversal vectors [6].

All the notations we shall use are according to the textbook of Ahuja, Magnanti and Orlin [7].

It is well-known the following propriety for the post-order traversal vector:

Propriety 1 The reversed post-order vector is a root-right-to-left traversal vector.

We shall name the root-left-to-right traversal as *first pre-order traversal* and the root-right-to-left traversal as *the second pre-order traversal*, because the nodes are analyzed similarly to pre-order traversal: root and then it's successors, but in reversed order (right-to-left).

In this paper we shall refer to the first and the second pre-order traversal vectors as the pre-order (traversal) vectors of the trees.

It is easy to observe that a pre-order (first or second) traversal vector starts with the root. So, we have the following propriety:

Propriety 2 In a (first or second) pre-order traversal vector the first element is the root of the binary tree.

Let $(a_i)_{i \in N}$ be the first pre-order traversal vector and $(b_i)_{i \in N}$ the second pre-order traversal vector. We suppose for each i and $j \in \{1, 2, ..., n\}$, $i \neq j$ that $a_i \neq a_j$ and $b_i \neq b_j$.

It is obviously that:

$$\{ a_i \mid i = 1, 2, ..., n \} = \{ b_j \mid j = 1, 2, ..., n \} = N, \tag{1.1}$$

where N is the set of nodes of the binary tree, $|N| = n$.

The following propriety of the first pre-order traversal vector helps us to prove the main theorems of this paper (theorem 1 and theorem 2).

Propriety 3 If u and v are consecutive nodes in the first pre-order traversal vector, then we have one of the following situations:

i) v is a son of the node u (see fig. 1.1). If u has two sons, then v is the left one.

or:

ii) v is the son to the right of the node w and u is the last node in the first pre-order traversal vector of the left partial tree T_1 of w (see fig. 1.2).

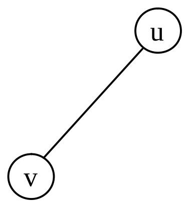

Fig. 1.1: First situation of propriety 3

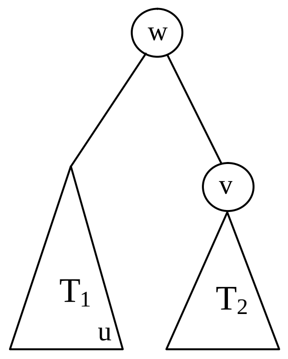

Fig. 1.2: Second situation of propriety 3

2. FINDING A SOLUTION

The algorithm we shall present is a adapted from the algorithm for reconstruction of the tree from the pre-order and post-order traversals [6].

We shall prove two theorems, which will lead us to the method to construct a binary tree from the pre-order and post-order traversals.

For each consecutive pair of nodes, u and v, from the first pre-order traversal vector using one of the next two theorems we can find the father denoted w of the node v. The method of finding w involves one of the situations described in propriety 3.

The main idea is to find a criterion for each situation that can appear and then to prove that there is only one node with this propriety.

Theorem 1

1. If a_{k-1} is before a_k ($k \in \{2, 3,..., n\}$) in the second pre-order traversal vector, i.e. $\exists\, i, j \in \{1, 2, ..., n\}$ so that $a_{k-1} = b_i$, $a_k = b_j$ and $i < j$, then the node a_k is a son of a_{k-1}. If a_{k-1} has two sons, then a_k is to the left of a_{k-1}.
2. If two nodes, u and v, are consecutive in both (the first and the second) pre-order traversal vectors, then the node v is the only son of u.

Proof. 1. Let k be an arbitrary value in the set $\{2, 3, ..., n\}$.
We make two notations:
$u = a_{k-1} = b_i$ and $v = a_k = b_j$. (2.1)

Let's suppose that the nodes (u and v) are respecting to the second situation from propriety 3. Then the second pre-order traversal vector (denoted $pre2$) is:
$pre2 = (..., w, ..., v, ..., u, ...)$. (2.2)

Since $u = b_i$ and $v = b_j$ (see (2.1)) it results from (2.2) that $i > j$, which $i < j$.

So, the consecutive nodes (u and v) of the first pre-order traversal vector are respecting the first situation in propriety 3.

2. The node v is a son of u (see **1.**).
Let's suppose that u has 2 sons. Let $t\ (\neq v)$ be the son to the right of u. Then, in the second pre-order traversal vector, the node t is next after u and, because $t \neq v$, the nodes u and v can not be consecutive (contradiction).

Theorem 2 If a_{k-1} is after a_k ($k \in \{2, 3, ..., n\}$) in the second pre-order traversal vector, i.e. $\exists\, i, j \in \{1, 2, ..., n\}$ so that $a_{k-1} = b_i$, $a_k = b_j$ and $i > j$, then the node a_k is the son to the right of a_p, where $p \in \{1, 2, ..., k\text{-}1\}$ and all the nodes in the first pre-order traversal vector (if exist) between a_p and a_k are after $b_j = a_k$ in the second pre-order traversal i.e. $\forall\, q \in \{p+1, p+2, ..., k\text{-}1\}, \exists\, h \in \{j+1, j+2, ..., n\} : a_q = b_h$.

Proof. We make the same notations as in the proof of theorem 1 (see (2.1)). We shall prove that the father denote w of v has the propriety that it is before v in the first pre-order traversal vector and all the nodes (if exist) between w and v (in the first pre-order traversal vector) are after v in the second pre-order traversal vector.

If u has more than a son (see the second part of the theorem 1), then u and v can not be respecting situation 1 in propriety 3, because u is after v in the second pre-order traversal vector. So, the nodes must be in the second situation (see fig. 1.2).

In the situation 2, the first pre-order traversal vector is:
$pre1 = (..., w, ..., u, v, ...)$. (2.3)

So, the father w is before his son v in the first pre-order traversal vector.

Let $t = a_q$ be a node between $w = a_p$ and $v = a_k$ ($p < k\text{-}1$) in the first pre-order traversal vector, where $q \in \{p+1, p+2, ..., k\text{-}1\}$. It results that the node t must be in the left partial binary tree of w.

If t is in T_1, then t must be after v in the second pre-order traversal vector.

So far, we proved that the father w of v has the propriety that it is before v in the first pre-order traversal vector and all the nodes (if exist) between w and v (in the first pre-order traversal vector) are after v in the second pre-order traversal vector. Next we shall prove that the father w of v is the only node with this propriety.

Let's suppose there exists another node denoted $w' \neq w$ with the same propriety. So, the node w' is before v in the first pre-order traversal vector. We have two situations:

i) The node *w'* is before *w* in the first pre-order traversal vector
or
ii) The node *w'* is between *w* and *v* in the first pre-order traversal vector.

In the situation i) the node *w* is between *w'* and *v*. All the nodes between *w'* and *v* are after *v* in the second pre-order traversal vector and so, the node *w* must be after *v* (contradiction with the propriety of *w*).

In the situation ii), because the node *w'* is between *w* and *v*, it must be after *v* in the second pre-order traversal vector (contradiction with the presumed propriety of *w'*).

In both cases we obtained contradiction. So, the father *w* of *v* is the only node with the above propriety. The theorem is proved.

The theorems 1 and 2 lead us to an algorithm for finding a binary tree using the pre-order traversal vectors.

The pseudo-code of the algorithm for finding a binary tree from the pre-order traversal vector is:

```
1.  a[1] is the root of the tree;
2.  a[1] pushes the stack s;
3.  For k:=1 to n do
4.    Pre2Pos[b[k]] := k;
    end for;
5.  m:=0;
6.  For k:=2 to n do
7.    left := true;
8.      While the last node u from the stack s is after
        a[k] in the second pre-order vector, i.e.
        Pre2Pos[u] > Pre2Pos[a[k]] do
9.          Pop node u out of the stack s;
10.         left := false;
      end while;
11.   Let w be the last node in the stack s;
12.   If left then
13.     If Pre2Pos[w]+1 = Pre2Pos[a[k]] then
        a[k] is the only son of w (left or right);
        m:=m+1; u[m] := w;
17.     else a[k] is the left son of w
      end if;
18.   else a[k] is the son to right of w;
      end if;
19.   The node a[k] pushes the stack s;
    end for;
```

Let's study now the algorithm.

The first node of any pre-order traversal vector is the root of the tree.

The first pre-order traversal vector components are sequentially (from the second to the last) analyzed and then inserted in the tree using the method given by the theorems 1 or 2, depending the situation.

One of the tricky linear time idea to do this in linear time is to decide in *O(1)* if a node is before or after another node in the second pre-order traversal vector. This can be easily done using a vector denoted *Pre2Pos*.

Pre2Pos[u] is the position of the node *u* in the second pre-order vector *b*.

The vector *Pre2Pos* can be built in linear time directly from the second pre-order traversal vector *b*.

If we treat directly the situations presented in the theorem 2, then the worst-case complexity of the method is greater then *O(n)* (for each such situation we have to verify if the nodes between a_p and a_k in the first pre-order traversal vector is after a_k in the second traversal vector). In order not to increase the complexity of the method we shall use a denoted *s* stack.

A node enters the stack *s* after it is analyzed. Nodes exit the stack only when a second situation of propriety 3 appears.

Let's notice that for the current node *a[k]*, before taking out any node from the stack, the last node from the stack is *a[k-1]* (previous analyzed and inserted).

If *a[k-1]* is before *a[k]* in the second pre-order vector, then we have the first situation in the propriety 3 and no node is popped out of the stack. So, the last node in the stack (which is *a[k-1]*) is the father of *a[k]* (see theorem 1).

If *a[k-1]* is after *a[k]* in the second pre-order vector then we have the second situation of the propriety 3. All the nodes in the stack which are after *a[k]* are in the second situation of the propriety 3 they are popped out of the stack.

After all these nodes are popped out of the stack, the last node in the stack is before *a[k]* in the second pre-order traversal vector. This node has the propriety that all the nodes between it and *a[k]* in the first pre-order traversal vector are after *a[k]* in the second pre-order traversal vector, because they have been already analyzed (they entered in the stack before). The current node *a[k]* is the son to the right of the last node in the stack (see theorem 2).

In the end of the algorithm, the variable *m* will store the number of situations where a node has only a son. The elements of the vector *u* are the nodes, which have only a son.

Theorem 3 The algorithm presented above constructs a binary tree from the pre-order traversals vectors.

Proof. The correctness of the algorithm is given by the observations above the algorithm and from the theorems 1 and 2.

If no iteration of *while ... do* in line 8 of the algorithm takes place, then the node *a[k]* is the left son of the last node *w* = *a[k-1]* in the stack, according to theorem 1, else *a[k]* is the son to the right of the last node *w* in the stack.

If there are pops out of the stack in the line 8, then the node *a[k]* is the son to the right of *w*, because in this case *a[k]* is not the first son of *w* and the first son was previously inserted in the binary tree to the left.

If the nodes *w* = *a[k-1]* and *a[k]* are consecutive in the second pre-order traversal vector, then the node *a[k]* is the only son of *w*. In this case, we can not decide if *a[k]* is the son to the left or to the right for *w* (see the second part of theorem 1).

Theorem 4 The algorithm finds a binary tree from the pre-order traversal vectors in linear time and space.

Proof. Every component *a[k]* ($k \in \{2, 3, \ldots, n\}$) of the first pre-order traversal vector enters once in *O(1)* in the stack *s* and, because of this, it can not exit the stack *s* more than once. So, the total number of iterations of all while loops in the whole algorithm is *O(n)*.

The worst-case time complexity of the whole algorithm is *O(n)*.

There are three vectors with *n* components involved and in the stack *s* all *n* nodes enter once, so the algorithm uses linear space.

It is easy to see that the construction of a binary tree from the traversal vectors can not be done in a complexity less than *O(n)*. So, the algorithm above has an optimal time and space complexity.

3. FINDING ALL THE SOLUTIONS

If two nodes, *u* and *v*, are consecutive in both (the first and the second) pre-order traversal vectors, then the node *v* is the only son of *u*. We can not decide which son of *u* (left or right) is *v*.

For the binary trees this do matter, because the case of *v* being the left son of *u* is different than the case of *v* being the son to the right for *u*.

So, having the first and the second traversal of a binary tree if there are situations described in the second part of the theorem 1, the solution is not unique. We can calculate the number of solutions.

Theorem 5 The number of solutions for the problem of construction of a binary tree from the pre-order and post-order traversals is 2^m, where *m* is the number of situations described in the second part of the theorem 1 that appear.

Proof. All the situations described in the second part of the theorem 1 are independent. So, for each of these situations the number of solutions multiplies by 2.

Let's take an example (see fig. 3.1).

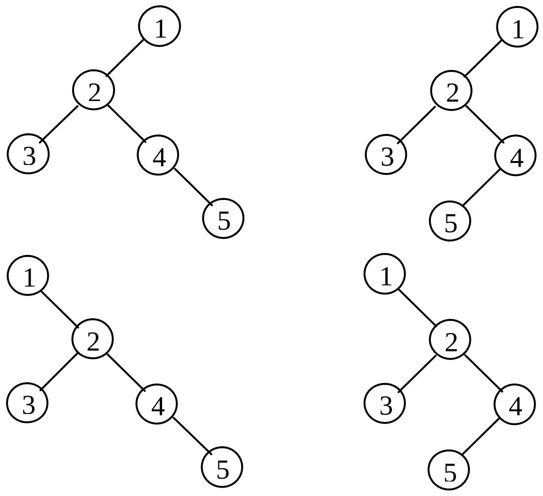

Fig. 3.1: Binary trees having the same pre-order and post-order vectors

In the figure 3.1, the first pre-order traversal vector is $a = (1, 2, 3, 4, 5)$ and the second pre-order traversal vector is $b = (1, 2, 4, 5, 3)$, then $m = 2$, because the pair of nodes $(1, 2)$ and $(4, 5)$ are consecutive in both pre-order traversal vectors. The total number of solutions for this example is $2^m = 4$.

We shall present now an optimal time and space complexity method to find all the solution for the problem for construction of the binary trees from the pre-order traversals.

We saw that a solution can be found in linear time and space. Starting from this one we can find all 2^m solutions. We shall use a vector d with m elements 0 or 1. Each element of d corresponds to a node, which has only a son. If $d[j] = 0$, then the son of the node $u[j]$ is to the left (or to the right) and if $d[j] = 1$, then the son of $u[j]$ is the other part (to the right and respectively to the left).

We shall generate all 2^m possible values for the vector d and for each such instance we shall find the corresponded solution. We'll take care to do this in $O(2^m)$ time and using linear space.

A binary tree can be kept in three vectors: *ls*, *rs* and *info*, each having n elements, as follows:

ls[k] is the son to the left of the node k. If k has no son, then $ls[k] = 0$.

rs[k] is the son to the right of the node k. If k has no son, then $rs[k] = 0$.

info[k] is the information of the node k.

The algorithm to find all the solutions for the problem of construction of the binary tree from the pre-order traversals is:

```
1.  Find a solution ls, rs, info and the vector u;
2.  For i:=1 to m do d[i]:=0;
    end for;
3.  For i:=1 to 2^m-1 do
4.    t:=1; j:=1;
5.        while t=1 do
6.          If d[j]=1 then d[j]:=0;
7.          else d[j]:=1; t:=0;
8.          end if;
9.          ls[u[j]] <-> ld[u[j]]; (interchange)
10.         j:=j+1;
11.       end while;
12.       In ls, lr, info is kept a new solution;
13. end for;
```

Theorem 6 The algorithm finds all the solutions for the problem of construction of the binary tree from the pre-order traversals.

Proof. All 2^m possible values for the vector d are generated and for each instance the corresponded solution is found.

Theorem 7 All the solutions for the problem of construction of the binary tree from the pre-order traversals are found in optimal time and space complexity.

Proof. The initial solution is found linear time and space (see theorem 4).

We shall calculate now the total number of *while* loops, which will give us the time complexity for the second part of the algorithm.

The iterations of a *while* sequence ends when *t* becomes *0*. This happens on the position of the first *0*.

Let's suppose that in the *i-th* iteration of *for*, the first *0* in *d* is on the position *k*, where $k \in \{1, 2, ..., m\}$. There are 2^{m-k} such cases. In such case, *while* sequence has *k* iterations.

So, the total number of iterations of *while* is:

$$\sum_{k=1}^{m} k \cdot 2^{m-k} \ . \tag{3.1}$$

Let's prove now that $O\left(\sum_{k=1}^{m} k \cdot 2^{m-k}\right) = O\left(2^m\right)$:

$$\lim_{m\to\infty} \frac{\sum_{k=1}^{m} k \cdot 2^{m-k}}{2^m} = \lim_{m\to\infty} \sum_{k=1}^{m} \frac{k}{2^k} \in \mathbf{R}_+^* \tag{3.2}$$

The sum $\sum_{k=1}^{m} \frac{k}{2^k}$ is convergent because:

$$\begin{cases} \sum_{k=1}^{m} \frac{k}{2^k} > 0, \ \forall m \in N^* \\ 0 < \frac{k}{2^k} < \frac{1}{k^2}, \ \forall k \geq k_0, \ k_0 \in N^* \\ \lim_{m\to\infty} \sum_{k=1}^{m} \frac{1}{k^2} \in R_+^* \end{cases} . \tag{3.3}$$

We can calculate the limit (3.2):

$$\lim_{m\to\infty} \frac{\sum_{k=1}^{m} k \cdot 2^{m-k}}{2^m} = \lim_{m\to\infty} \frac{\sum_{k=0}^{m-1} (m-k) \cdot 2^k}{2^m} = \lim_{m\to\infty} \frac{\sum_{k=0}^{m} (m+1-k) \cdot 2^k - \sum_{k=0}^{m-1} (m-k) \cdot 2^k}{2^{m+1} - 2^m} =$$

$$\lim_{m\to\infty} \frac{2^m + \sum_{k=0}^{m-1} 2^k}{2^m} = \lim_{m\to\infty} \frac{2^{m+1} - 1}{2^m} = 2 \tag{3.4}$$

The complexity of the whole algorithm is $O(n+2^m) = O(max\{n, 2^m\})$.

It is obviously that an optimal time algorithm can not have the complexity less than $O(max\{n, 2^m\})$, because the elements of the traversal vectors must be analyzed at least once (in $O(n)$) and the number of solutions is 2^m.

The algorithm uses linear space (see theorem 4).

This means that the algorithm for finding all the solutions for the problem of construction the binary trees from the pre-order traversals has an *optimal time and space complexity.*

BIBLIOGRAPHY

[1] H.A. Burgdor, S. Jajodia, F.N. Springsteel, Y. Zalcstein (1987), “Alternative Methods for the Reconstruction of Trees from their Traversals”, BIT 27.2, 134-140;
[2] G. Chen, M.S. Yu, L. T. Liu (1988), “Two algorithms for constructing a binary tree from its traversals”, Inf. Process. Lett. 28, No.6, 297-299;
[3] E. Mäkinen (1989), “Constructing a binary tree from its traversals”, BIT 29, No.3, 572-575;
[4] E. Mäkinen (2000), “Constructing a binary tree efficiently from its traversals”, Int. J. Comput. Math. 75, No.2, 143-147;
[5] A. Andersson, S. Carlsson (1990), “Construction of a tree from its traversals in optimal time and space”, Inf. Process. Lett. 34, No.1, 21-25;
[6] A. Deaconu (2004), “Iterative Algorithm for Construction of a Tree from its Pre-order and Post-order Traversals in Linear Time and Space”, Analele Stiintifice Univ. “Alexandru Ioan Cuza” din Iasi (serie noua), ed. Univ. “Al. I. Cuza”, Tomul IV;
[7] R. K. Ahuja, T. L. Magnanti, J. B. Orlin (1993), “Network flows: Theory, Algoritms and Applications”, Prentice Hall.