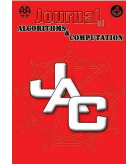
NAKHODA

Journal of Algorithms and Computation

journal homepage: http://jac.ut.ac.ir



An improved algorithm to reconstruct a binary tree from its inorder and postorder traversals

Niloofar Aghaieabiane*[1], Henk Koppelaar†[2] and Peyman Nasehpour‡[3]

[1]Department of Engineering, School of Computer Science, New Jersey Institute of Technology, Newark, New Jersey, the USA.
[2]Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology, Delft, The Netherlands.
[3]Golpayegan University of Technology, Department of Engineering Science, Golpayegan, Iran.

ABSTRACT

It is well-known that, given inorder traversal along with one of the preorder or postorder traversals of a binary tree, the tree can be determined uniquely. Several algorithms have been proposed to reconstruct a binary tree from its inorder and preorder traversals. There is one study to reconstruct a binary tree from its inorder and postorder traversals, and this algorithm takes running time of $O(n^2)$. In this paper, we present InPos an improved algorithm to reconstruct a binary tree from its inorder and postorder traversals. The running time and space complexity of the algorithm are an order of $\theta(n)$ and $\theta(n)$ respectively, which we prove to be optimal.

Keyword: Binary tree, Preorder traversal, Inorder traversal, Postorder traversal, Time complexity, Space complexity.

ARTICLE INFO

Article history:
Received 30, March 2017
Received in revised form 18, December 2017
Accepted 30 February 2018
Available online 01, June 2018

AMS subject Classification: 05C78.

*niloofaraghaie@ut.ac.ir, na396@njit.edu
†Koppelaar.Henk@gmail.com
‡Corresponding author: P. nasehpour. Email: nasehpour@gut.ac.ir, nasehpour@gmail.com

1 Abstract continued

The InPos algorithm not only reconstructs the binary tree, but also it determines different types of the nodes in a binary tree; nodes with two children, nodes with one child, and nodes with no child. At the end, the InPos returns a matrix-based structure to represent the binary tree, and enabling access to any structural information of the reconstructed tree in linear time with any given tree traversals.

2 Introduction

Binary trees [9, 22] are fundamental structures in computer science (CS). Tree traversals are therefore among one of the most important topics in CS [3]. In fact, many algorithms work on binary trees by using its traversals. We mentione a few examples here. The time complexity of combinatorial Gray coding by a twisted tree and new tree traversal, has been abdicated to $O(1)$ time [24]. In [12], it has been shown that k-pebble tree transducer, a kind of tree traversal, can be typechecked in $(k+2)$-fold exponential time for both ranked and unranked tree. In [13], A method for semi-algorithmic verification of programs has been proposed that manipulate balanced trees using a tree traversals.
In general, there are three binary tree traversals; inorder, preorder, and postorder. These binary tree traversals are also termed tree walk in [9]. The inorder traversal first visits the value of the root of a subtree between visiting the values of its left subtree and respectively visiting those in its right subtree. Similarly, a preorder traversal visits the root before the values in either subtree, and the postorder traversal prints the root after the values in its subtrees [9].
The problem of reconstructing a binary tree from its inorder preorder traversals, is first proposed by Knuth [17]. This reconstruction is also used in program inversion [8]. Nowadays it is common knowledge that given the inorder traversal of a binary tree, along with one of the preorder or postorder traversals, one can identify and reconstruct the tree uniquely. Regarding a binary tree T with n nodes, several studies have been presented to rebuild a binary tree from its inorder and preorder traversals as well as inorder postorder traversals. Primary studies used the inorder-preorder sequence (i-p sequence for short), which is annunciated in [14] and also is called tree-permutation in [16]. These algorithms are based on two stages. In the first stage the i-p sequence is built, and in the second stage the binary tree will be reconstructed using the i-p sequence. We review here all the results from the literature to date.
Two algorithms were proposed by [4] to rebuild a binary tree from its inorder and preorder as well as inorder and postorder traversals. Both algorithms take $O(n^2)$ time. Shortly thereafter [6, 7], presented two algorithms to reconstruct a binary tree from its inorder and preorder traversals. In one of these two algorithms binary search [5] and in the other hashing techniques [18] are applied. The second algorithm has running time and space $O(nlogn)$ and $O(n)$ respectively. The first one has running time of $O(n)$ but non-optimal space complexity. Solugh and Effe [23] also used this approach but with a slight twist; firstly, they applied the preorder traversal in the second stage of their algorithm. Moreover, instead of using the search method aforementioned, they applied the recursive property of the i-p sequence. The running

time of the algorithm is linear.
In Mäkinen [20], an iterative algorithm is presented to rebuild a binary tree from its inorder and preorder traversals. The algorithm uses a stack of controlled pointers, and reconstructs a binary tree using this stack and moving on the preorder and inorder traversals. Besides, a certain combination of the two traversals were applied. The algorithm has time and space complexity of an order of $O(n)$. Four algorithms were proposed in [2], to rebuild an order binary tree from its preorder traversal along with structural information, like inorder or postorder traversals. Firstly, they proved that given preorder and postorder traversals of an ordered binary tree, the tree will be reconstructed. Secondly, they showed that given preorder traversal along with inorder traversal of a directed tree, the tree will be rebuilt. Thirdly, they illustrated with only preorder traversal of a binary search tree, the tree will be reconstructed. All these three algorithms take linear time and need an extra stack space which is commensurate with the height of the reconstruction tree. Finally, they proposed an iterative algorithm to reconstruct a directed tree from its preorder and postorder traversals in optimal space by using empty pointers instead of a stack [2]. Cameron et al. [5] applied dual-order traversal [21], the combination of inorder and preorder traversals, to obtain the i-p sequence. The time and space complexity of their algorithm are in orders of $O(n)$ and $O(h)$, where h is height of the tree respectively.
Recently, two studies [3, 10] focus on an algorithm to rebuild a binary tree from its inorder and preorder traversals. The earlier study [10] proposes an iterative algorithm which takes the $O(n)$ and $O(nlogn)$ for time and space respectively. In the later study [3] the authors 'prove' that the algorithm proposed in [10] works correctly in most of the cases, however, in some cases it returns a binary tree which is not correct. To address this problem, they use some restrictions on the algorithm which take a constant time. Consequently, the modified algorithm [3] takes the same time and space complexity as the original critisized algorithm and is not better. In the best case time the algorithm takes $O(n)$, though.
In [11] an algorithm to reconstruct all binary trees from its preorder and reversal of preorder is presented, which takes a linear time and space.
Here in this paper, we consider binary trees [9, 22] rather than ordered binary trees [2], such that the position of the nodes does matter; whether an internal node is either a left or right child. We will present our novel algorithm InPos to reconstruct a binary tree from its inorder and postorder traversals, which, as is well-known, enables to reconstruct the binary uniquely. But InPos not only reconstructs the binary tree, it also determines different types of each node: whether it has no child (which also is termed leaf of a binary tree), one child, or is a node with two children. At the end InPos returns a matrix-based structure, such that any structural information of the reconstructed binary tree can be gained in linear time by any given tree traversals.
In section 3, we give InPos and exemplify how this algorithm works. In section 4, we present our results and in section 5 we give our discussion in which we discuss the some of the presented algorithms with details. Finally, in section 6 our conclusion is given. Although in this paper we work on tree traversal sequences rather than the shape of the tree, in order to exemplify our discussion in some cases we have depicted the shape of the tree or some parts of it abstractly.

3 Algorithm

In this section, we propose the InPos algorithm to reconstruct a binary tree from its inorder and postorder traversals. Then, within an example we will show how the algorithm works. For each type each node we classify it into one of three following categories.

i nodes with two children (two-child nodes), such that the node has exactly two children.

ii nodes with one child (one-child nodes), such that the node has exactly one child.

iii nodes with no child (no-child nodes), such that the node has no child, is a leaf.

3.1 Presenting the reconstruction algorithm

Here, we first give the general description of the InPos algorithm. Then, we explain the pseudo code of it. Based on postorder traversal, we have the following properties.

Property 1. *According to postorder traversal, the rightmost element of the postorder traversal is always the root node.*

Property 2. *According to postorder traversal, the parent node is always placed after its child (children).*

On the other hand, the inorder and postorder traversals together share some features. Considering a node in postorder traversal, the left and right subtrees of the node can be determined by inorder traversal. Theorem 1 reflects this feature.

Theorem 1. *If a node x is in postorder traversal, then in the inorder traversal all the elements at the left and at the right of of x are in the left and right subtrees of x respectively.*

Proof. For a given node, say x, according to inorder traversal, the left subtree of x will be visited, then the x element, and finally its right subtree. Therefore, all the elements at the left and at the right of the x in inorder traversal are placed in the left and right subtrees of x respectively.

□

Thus, regarding each node in postorder traversal, we can define the elements on its left and right subtrees respectively, based on inorder traversal.
Nodes with one child whose only child is a leaf also share a feature in both traversals. In fact, these two nodes will appear as two consecutives in both traversals, however, their orders depend on the position of the child; either left or right. Theorem 2 and Theorem 3 show this feature.

Theorem 2. *A one-child node x has a leaf left child y if and only if* y,x*are two consecutive elements both in inorder and postorder traversals.*

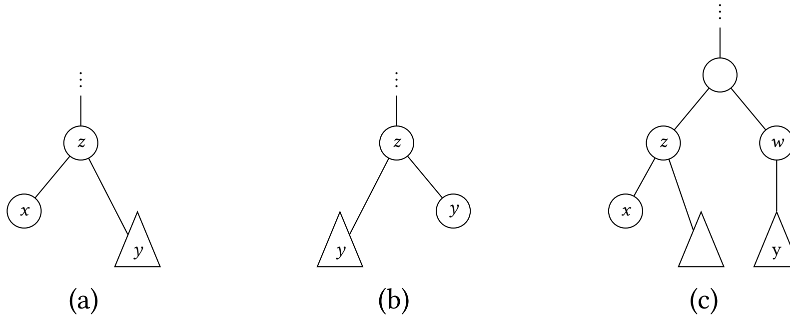


Figure 1: Parts of a binary tree considered in Theorem 1 and Theorem 2. Subtrees which do not contain any value means that those subtrees can be either empty or not. Vertical edges show that those edges can be either left or right. (a) Part of a binary tree in which the node y is in a right subtree of the node z. (b) Part of a binary tree in which the node y is in a left subtree of the node z. (c) Part of a binary tree in which the node y is placed in a subtree which is not rooted at node z.

Proof. ($\Rightarrow$): According to inorder traversal, since the node x does not have a right child and the node y is a leaf, the node y and then the node x will be visited. Likewise, in postorder traversal the node y and then the node x are visited respectively.
($\Leftarrow$): In contrary, let x is not a one-child node with the leaf left child y. We have the three following cases:

1. The node x is leaf and the node y is placed elsewhere. Let z be the parent of x. Without loss of generality, let x be the left child of the z. We have the two following cases:

   (a) The node y is placed in right subtree of the node z (Figure 1(a)). According to postorder traversal, in general, we have $\ldots,x,\ldots,y,\ldots,z,\ldots$. It is evident that the node y never precedes the node x, thus it is a contradiction.

   (b) The node y is place in a one of a subtrees rooted from a node except z, say w (Figure 1(c)). According to postorder traversal, in general, we have either $\ldots,x,z,\ldots,y,\ldots,w,\ldots$ or $\ldots,y,\ldots,w,\ldots,x,z,\ldots$. It is evident that the node y never precedes the node x, thus it is a contradiction.

2. The node x is node with one child. Let its child be z which is a leaf. Without loss of generality, let z is the left child of the x and the node y is placed elsewhere. According to postorder traversal, in general, we have $\ldots,z,x,\ldots$. It is evident that the node x always immediately followed by the node z. Thus, it is a contradiction. It should be noted that in this case by the assumption, we just can consider that the node z is a leaf.

3. The node x is node with two children z and w such that the nodes z and w are the left and right children of the x respectively. We have three following cases:

   (a) The node y is placed in the on of the subtrees of the node z. According to postorder traversal, in general we have $\ldots,y,\ldots,z,\ldots,w,x,\ldots$. It is evident that the node y never

precedes the node x, thus it is a contradiction.

(b) The node y is placed in the one of the subtrees of the node of the node w. According to postorder traversal, in general, we have, $\ldots,z,\ldots,y,\ldots,w,x,\ldots$. The node x always followed by the node w, which is contradiction.

(c) The node y is neither in left and right subtrees of the node x, so it is placed elsewhere. According to postorder traversal, in general, we have either $\ldots,y,\ldots,z,\ldots,w,x,\ldots$ or $\ldots,z,\ldots,w,x,\ldots,y,\ldots$. It is evident that the node x is never followed by the node y, which is contradiction.

Therefore, x is a node with one child y which is a leaf and the left child.

□

Theorem 3. *A one-child node* x *has a leaf right child* y *if and only if* x,y *and* y,x *two consecutive elements in inorder and postorder traversals respectively.*

Proof. (⇒): According to inorder traversal, since the node x does not have a left child and the node y is a leaf, in inorder traversal the node x and then the node y will be visited. On the other hand, in postorder traversal the node y and the node x are visited respectively.
(⇐): In contrary, let x is not a one-child node with the leaf right child y. We have the three following cases:

1. The node x is leaf and the node y is placed elsewhere. Let z be the parent of x. We have the two following cases:

   (a) Either the node y is placed in left subtree of the node z (Figure 1(a)), which according to inorder traversal, in general, we have $\ldots,y,\ldots,z,x,\ldots$ or the node y is place right subtree of the node z, which based on inorder traversal, in general, we have $\ldots,x,z,\ldots,y,\ldots$ (Figure 1(b)). It is evident in both conditions the node y never precedes the node x, thus it is a contradiction.

   (b) The node y is place in one of the subtree rooted from a node except z, say w and the node x is either the left or right child of the node z. According to postorder traversal, in general, we have either $\ldots,x,z,\ldots,y,\ldots,w,\ldots$ or $\ldots,y,\ldots,w,\ldots,x,z,\ldots$. It is evident that node z is always followed by the node z, thus it is contradiction.

2. The node x is node with one child. Let its child be z and is a leaf. Without loss of generality, let z is the right child of the x, and the node y is placed elsewhere. According to postorder traversal, in general, we have $\ldots,z,x,\ldots$. Since the node z always precedes the node x, it is a contradiction. It should be noted that in this case by the assumption, we just can consider that the node z is a leaf.

3. The node x is node with two children z and w such that the nodes z and w are the left and right children of the x respectively. We have three following cases:

   (a) The node y is placed in the on of the subtrees of the node z. According to postorder traversal, in general we have $\ldots,y,\ldots,z,\ldots,w,x,\ \ldots$. It is evident that the node w always precedes the node x, thus it is a contradiction.

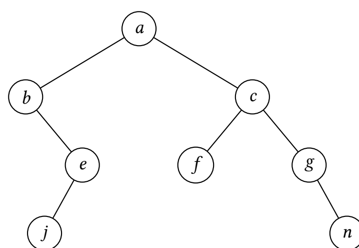


Figure 2: A binary tree. The inorder traversal is *b,j,e,a,f,c,g,n*, and the postorder traversal is *j,e,b,f,n,g,c,a*

(b) The node y is placed in the one of the subtrees of the node w. According to postorder traversal, in general, we have, $\ldots,z,\ldots,y,\ldots,w,x,\ldots$. The node w always precedes the node x, which is contradiction.

(c) The node y is neither in left and right subtrees of the node x, so it is placed elsewhere. According to postorder traversal, in general, we have either $\ldots,y,\ldots,z,\ldots,w,x,\ldots$ or $\ldots,z,\ldots,w,x,\ldots,y,\ldots$. It is evident that the node w always precedes the node x, which is contradiction.

Therefore, x is a node with one child y which is a right child.

□

The algorithm starts the reconstruction process by both postorder and inorder traversals together. It uses a stack and inorder traversal for control and postorder traversal to reconstruct a binary tree from its inorder and postorder traversals. It moves on postorder from the last to the first element, because of the Property 1 and Property 2. In fact, it starts the reconstruction process by considering the root node which is located at the end of the postorder traversal. It then, first reconstructs the right subtree of the root and then its left subtree recursively, since in postorder traversal based one Property 2 the right child of a node is visited after its left child. The algorithm, in each step considers a node from postorder traversal, say current node. It then, determines whether the child of the current node have a right subtree or not according to Theorem 1, Theorem 2, and Theorem 3 . In fact, according to these theorems we consider in the algorithm per iteration two consecutive elements in postorder traversal (for more information see the InPos algorithm). On the other hand, since for the current node all the elements in its left and right subtrees are placed before and after the current node in inorder traversal, it moves on the inorder traversal from the last element to the right. If the current node has a right child, it pushes the index child node of postorder traversal in the stack. By pushing the index of the node, it can access the node without any searching during the pop process. Thus, in the stack the index of the elements that have a right subtree will be pushed. The algorithm pops up an element from the stack, when the right subtree of the element at the top of the stack is already built, and its left subtree should be built. So the current node will be changed to the element at the top of the stack. In effect, it is evident when the current inorder traversal node is equal to the top of the stack, the right subtrees are already constructed, and so the left subtrees of the elements in the stack, if existing, should be rebuilt.

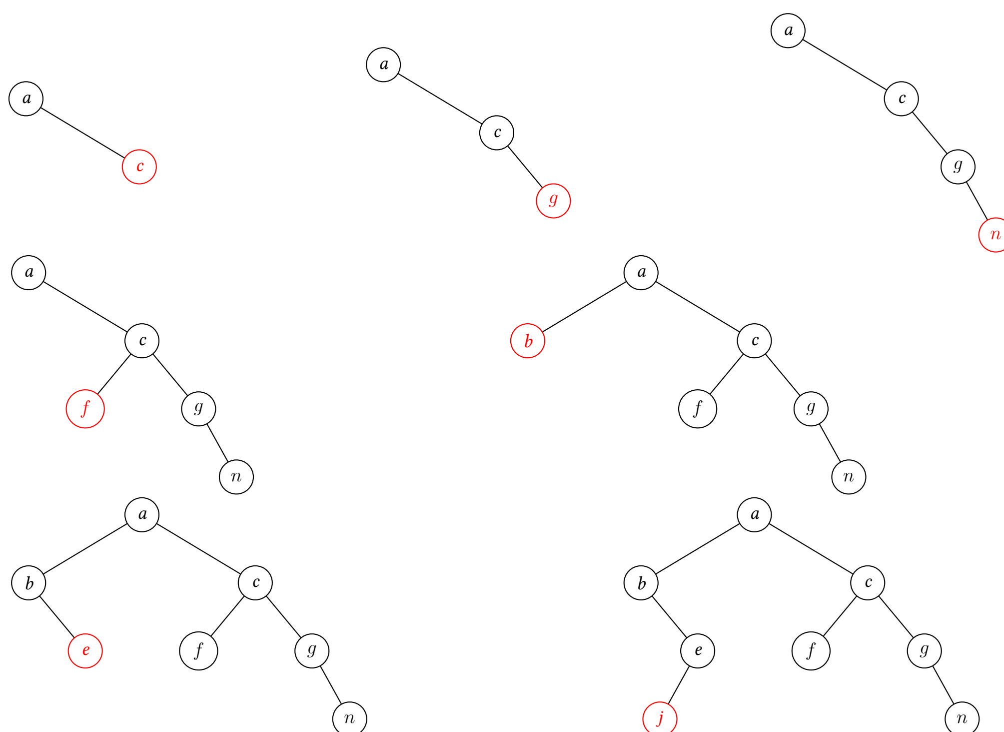


Figure 3: The process of the reconstruction of the binary tree depicted in Figure 2. The process of the reconstruction is shown one by one. In each step the added node is colored red.

As a result, in this condition it pops up the element at the top of the stack and changes the current node to the popped element. Thereafter, it initiates the attributes of the current node as well as the parent index of the child, and continue this process for either the child node or for the element popped up from the stack. Regarding a binary tree illustrated in Figure 2, the process of reconstruction is depicted in Figure 3, one by one. As is seen in Figure 3 the right subtree of each node firstly is rebuild, then the algorithm reconstructed the left subtrees, if they existed.

3.2 InPos algorithm

Here, we give the pseudo code of the InPos algorithm using the pseudo code style defined in [9]. The algorithm is implemented in the C# and MATLAB programming languages, and is available upon request. In this algorithm, we let In and Pos denote to inorder and postorder traversals respectively. Two variables $Left$ and $Right$ show whether the algorithm is working on a left or right subtree respectively. The variable $CurrentNode$ also shows the considered node in postorder traversal. Two functions, Push and Pop show the two operation on the stack. The function Push$(stack, x)$, inserts the element x at the top of the stack. Similarly, the operation Pop, returns and then removes the element at the top of the stack.
In InPos we consider four attributes per node; $parent$, $left$, $right$, and $type$. The attributes $parent$, $left$, and $right$, for a given node show the parent, left child, and right child indices

corresponding to the postorder traversal. The attribute $type$ indicates the type of the node, which is either 0, 1, and 2. These numbers correspond respectively to no-child, one-child, and two-child nodes. We note that there is difference between $Left$ and $left$ as well as $Right$ and $right$, so that the variables $Left$ and $Right$ depict that in which subtree the algorithm is working, while $left$ and $right$ are attributes that should be initiated during the reconstruction process.
Algorithm1 in initial steps, makes the type value of each node equivalent to 0. Two variables $count_{pos}$ and $count_{in}$ illustrate the indices of the considered nodes in inorder and postorder traversals respectively. As is said in advance, since the process of reconstruction starts with the end of the two traversals, the initial value of the $count_{in}$ is equal to the index of the last element in inorder traversal. The value of the $count_{pos}$ is at first equal to the last but two elements of the postorder traversal, since it is known that the last element is the root node, and during the reconstruction process the two postorder consecutives from the end to the first are considered.
The algorithm, then firstly checks whether the last elements if the inorder and postorder traversals are equal or not. By this comparison it recognizes that the root of the tree has a right subtree or not. According to Theorem 1, it is obvious, if they are equal, then the root does not have a right subtree, and so the algorithm should work on the left subtree of the root, and in this condition it sets the variable $Left$ equal to 1. Otherwise, it pushes the root node into to the stack, since it has the right subtree (for more information see section 2.1 Presenting the reconstruction algorithm). As it was mentioned before, by two variables $Left$ and $Right$ the algorithm identifies whether to work on either left or right subtree of a current node. At first, the variable $CurrentNode$ is equal to the last element of postorder, as the algorithm starts the reconstruction process from the root of the tree. In each iteration of the **for** loop (line 12), the position of a node is recognized and based on this its corresponding attributes are initiated. The **while** loop (line 13), checks the element at the top of the stack whether it is equivalent to the considered node in inorder traversals or not. If they are equal, the algorithm understands that the right subtree of the considered node in postorder is already rebuilt and so it pops up the element from top of the stack and initiates the $CurrentNode$ to this element, and sets the variable $Left$ equal to 1, since it has to work on the left subtree of the $CurrentNode$ in postorder traversal. In fact, if a node is popped up from the stack, the algorithm recognizes that the right subtrees are rebuilt and it should reconstruct the left subtree of the pushed nodes, one by one. Here, it uses a loop, because some nodes in the stack may not have any left subtree and then by the comparison in the **while** loop (line 13) it eliminates the nodes that are pushed and do not have a left subtree. Therefore, after this loop, the $CurrentNode$ is such node that its right subtree is already rebuilt and it has a left subtree that the algorithm should reconstruct. In the rest of the algorithm there are two **if** conditions, such that each of them along with their **else** parts initiate the corresponding attributes based on the position of the current node, its child, variables $Left$ and $Right$. In fact, in each iteration one of these four parts is performed.

Algorithm 1 Reconstructing a binary tree from its inorder and postorder traversals.

```
InPos(In, Pos)
 1  stack.top = NIL
 2  CurrentNode = Pos[Pos.length]
 3  count_in = In.length
 4  for i = Pos.length downto 1
 5      Pos[i].type = 0
 6  if Pos[Pos.length] == In[In.length]
 7      Left = 1
 8      Right = 0
 9  else Left = 0
10      Right = 1
11      Push(stack, Pos.length)
12  for count_pos = Pos.length − 2 downto 0
13      while Pos[stack.top] == In[count_in] and stack.top ≠ NIL
14          CurrentNode = stack.top
15          Pop(stack)
16          Left = 1
17          Right = 0
18          count_in = count_in − 1
19      if Pos[count_pos + 1] ≠ In[count_in]
20          Push(stack, count_pos + 1)
21          if Left == 1
22              Pos[CurrentNode].left = Count_pos + 1
23              Pos[count_pos + 1].parent = CurrentNode
24              Pos[CurrentNode].type = Pos[CurrentNode].type + 1
25          else Pos[CurrentNode].right = Count_pos + 1
26              Pos[count_pos + 1].parent = CurrentNode
27              Pos[CurrentNode].type = Pos[CurrentNode].type + 1
28          Left = 0
29          Right = 1
30      else if Left == 1
31              Pos[CurrentNode].left = Count_pos + 1
32              Pos[count_pos + 1].parent = CurrentNode
33              Pos[CurrentNode].type = Pos[CurrentNode].type + 1
34          else Pos[CurrentNode].right = Count_pos + 1
35              Pos[count_pos + 1].parent = CurrentNode
36              Pos[CurrentNode].type = Pos[CurrentNode].type + 1
37          Left = 1
38          Right = 0
39          count_in = count_in − 1
40      CurrentNode = count_pos + 1
```

	1	2	3	4	5	6	7	8
In	b	e	j	a	f	c	g	n

	1	2	3	4	5	6	7	8
Pos	j	e	b	f	n	g	c	a

	1	2	3	4	5	6	7	8
$parent$	/	/	/	/	/	/	/	/
$left$	/	/	/	/	/	/	/	/
$right$	/	/	/	/	/	/	/	/
$type$	0	0	0	0	0	0	0	0

Figure 4: Initial step, consider four attributes per node associated with postorder traversal. all the three attributes of $parent$, $left$, and $right$ are equal to NIL , and the value of the attribute $type$ is equivalent to 0.

	1	2	3	4	5	6	7	8
In	b	e	j	a	f	c	g	n

	1	2	3	4	5	6	7	8
Pos	j	e	b	f	n	g	c	a

	1	2	3	4	5	6	7	8
$parent$	/	/	/	/	/	/	8	/
$left$	/	/	/	/	/	/	/	/
$right$	/	/	/	/	/	/	/	7
$type$	0	0	0	0	0	0	0	1

Figure 5: First iteration of the **for** loop. The element c is defined as right child of the element a. The corresponding attributes are initiated.

3.3 Example of the InPos algorithm

Here we exemplify how our algorithm InPos works. Once again consider the inorder and postorder traversals for the binary tree illustrated in Figure 2. Each figure shows one iteration of the **for** loop. For convenience, in each figure the initiated attributes as well as their associated indices are colored red.

Figure 4 shows the initial steps of the algorithm, in which all the attributes except the $type$ are equal to NIL, and the initial value of the $type$ for all the elements is 0. The index of the root (i.e. 8) is pushed into the stack. Figure 5 shows the first iteration of the **for** loop of the algorithm. The index of the element c (i.e. 7) is pushed into the stack. The right child of the root node (i.e. a) is defined which is the element c. The attributes $right$ and $type$ for the element a, as well as the attribute $parent$ for the element c are initiated. Figure 6 shows the second iteration of the **for** loop of the algorithm. The index of the elements g (i.e. 6) is pushed into the stack. The right child of the $CurrentNode$ (i.e. c) is defined which is the element g.

	1	2	3	4	5	6	7	8
In	b	e	j	a	f	c	g	n

	1	2	3	4	5	6	7	8
Pos	j	e	b	f	n	g	c	a

	1	2	3	4	5	6	7	8
$parent$	/	/	/	/	/	7	8	/
$left$	/	/	/	/	/	/	/	/
$right$	/	/	/	/	/	/	6	7
$type$	0	0	0	0	0	0	1	1

Figure 6: Second iteration of the **for** loop. The element g is defined as right child of the element c. The corresponding attributes are initiated.

	1	2	3	4	5	6	7	8
In	b	e	j	a	f	c	g	n

	1	2	3	4	5	6	7	8
Pos	j	e	b	f	n	g	c	a

	1	2	3	4	5	6	7	8
$parent$	/	/	/	/	6	7	8	/
$left$	/	/	/	/	/	/	/	/
$right$	/	/	/	/	/	5	6	7
$type$	0	0	0	0	0	1	1	1

Figure 7: Third iteration of the **for** loop. The element n is defined as right child of the element g. The corresponding attributes are initiated.

	1	2	3	4	5	6	7	8
In	b	e	j	a	f	c	g	n

	1	2	3	4	5	6	7	8
Pos	j	e	b	f	n	g	c	a

	1	2	3	4	5	6	7	8
$parent$	/	/	/	7	6	7	8	/
$left$	/	/	/	/	/	/	4	/
$right$	/	/	/	/	/	5	6	7
$type$	0	0	0	0	0	1	2	1

Figure 8: Next iteration of the **for** loop. The element f is defined as left child of the element c. The corresponding attributes are initiated.

	1	2	3	4	5	6	7	8
In	b	e	j	a	f	c	g	n

	1	2	3	4	5	6	7	8
Pos	j	e	b	f	n	g	c	a

	1	2	3	4	5	6	7	8
$parent$	/	/	8	7	6	7	8	/
$left$	/	/	/	/	/	/	4	3
$right$	/	/	/	/	/	5	6	7
$type$	0	0	0	0	0	1	2	2

Figure 9: Next iteration of the **for** loop. The element b is defined as left child of the element a. The corresponding attributes are initiated.

	1	2	3	4	5	6	7	8
In	b	e	j	a	f	c	g	n

	1	2	3	4	5	6	7	8
Pos	j	e	b	f	n	g	c	a

	1	2	3	4	5	6	7	8
$parent$	/	3	8	7	6	7	8	/
$left$	/	/	/	/	/	/	4	3
$right$	/	/	2	/	/	5	6	7
$type$	0	0	1	0	0	1	2	2

Figure 10: Next iteration of the **for** loop. The element e is defined as right child of the element b. The corresponding attributes are initiated.

	1	2	3	4	5	6	7	8
In	b	e	j	a	f	c	g	n

	1	2	3	4	5	6	7	8
Pos	j	e	b	f	n	g	c	a

	1	2	3	4	5	6	7	8
$parent$	2	3	8	7	6	7	8	/
$left$	/	1	/	/	/	/	4	3
$right$	/	/	2	/	/	5	6	7
$type$	0	1	1	0	0	1	2	2

Figure 11: Next iteration of the **for** loop. The element b is defined as left child of the element a. The corresponding attributes are initiated.

The attributes $right$ and $type$ for the element c, as well as the attribute $parent$ for the element g are initiated. Figure 7 shows the third iteration of the **for** loop of the algorithm. No elements are pushed or popped up into or from the stack. The right child of the $CurrentNode$ (i.e. g) is defined which is the element n. The attributes $right$ and $type$ for the element g, as well as the attribute $parent$ for the element n are initiated. Figure 8 shows the next iteration of the **for** loop of the algorithm. The index of the elements g and c (i.e. 6 and 7) are popped up from the stack in order. The left child of the $CurrentNode$ (i.e. c) is defined which is the element f. The attributes $left$ and $type$ for the element c, as well as the attribute $parent$ for the element f are initiated. Figure 9 shows the next iteration of the **for** loop of the algorithm. The index of the element a (i.e. 8) is popped up from the stack, and then the index of the element b (i.e. 3) is pushed into the stack. The left child of the $CurrentNode$ (i.e. a) is defined which is the element b. The attributes $left$ and $type$ for the element a, as well as the attribute $parent$ for the element b are initiated. Figure 10 shows the next iteration of the **for** loop of the algorithm. No elements are popped up or pushed from or into the stack. The right child of the $CurrentNode$ (i.e. b) is defined which is the element e. The attributes $right$ and $type$ for the element b, as well as the attribute $parent$ for the element e are initiated. Figure 11 shows the next iteration of the **for** loop of the algorithm. No elements are popped or pushed from or into the stack. The left child of the $CurrentNode$ (i.e. e) is defined which is the element j. The attributes $left$ and $type$ for the element e, as well as the attribute $parent$ for the element j are initiated. In brief, Figure 4 trough Figure 11 show the process of the reconstruction of the binary tree illustrated in Figure 2 from its inorder and postorder traversal by the algorithm InPos.

4 Results

In this section, we investigate time and space complexity of the Algorithm 1. The number of the elements in either inorder or postorder traversal is equal to n, and $T(n)$ as well as $S(n)$ denote to time and space complexity respectively, We probe the complexity of each of them separately.

4.1 Time complexity of InPos

Here, we look for the time complexity of the algorithm InPos. Except the **for** and **while** loops in lines 4,12, and 13 other commands take constant time, so they can be discarded. The **for** loop in line 10 takes $\theta(n)$. The **for** loop in line 12, runs $n-2$ times, which is an order of the $O(n)$. The **while** loop in line 13, repeats until its conditions do not hold. It compares the element at the top of the stack with the element in inorder traversal using the variable $count_{in}$. On the other hand, as was said before, the index of the nodes that have a right subtree are pushed into the stack. Therefore, the height of the stack is at most equal to the number of the nodes that have right subtree. Shape of a binary tree with n nodes that has the maximum number of the nodes with right subtrees is a right chain. Figure 12 shows an example of right chain binary tree. In this case the inorder and postorder traversals are reverse of each other Theorem 4 reflects it.

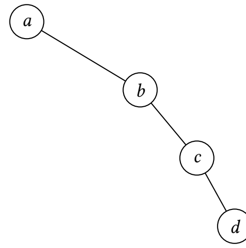


Figure 12: A right chain binary tree. The inorder traversal is *a,b,c,d*, the postorder traversal is *d,c,b,a*

.

Theorem 4. *If a binary tree has shape of right chain, then the inorder and postorder traversals are reverse of each other.*

Proof. According to inorder traversal, since each node does not have any left subtree, at first the root will be visited and then its left child. Thus, nodes will be visited in term of their level of the tree respectively. In other words, at first the root will be visited, then the node at the level 2 and so on. Whereas, based on postorder traversal, since each node does not have any left subtree, the right subtree will be visited first and then the root. Since each node in the right subtree (except the leaf node) has a parent node, the parent is also placed in the right subtree of its parent, these nodes will be visited from the leaf to root level by level.

□

If the binary tree is a right chain, the condition of the **while** loop, in line 13, never holds until the last repeation of the **for** loop in line 13. Hence, the number of iterations of the **while** loop can be discarded. The time complexity of the algorithm InPo depends on the number of iterations of the **for** loops in line 4 and 12, which is

$$\begin{aligned} T(n) &= \theta(n) + \mathrm{O}(n) \\ &= \theta(n). \end{aligned} \tag{1}$$

Thus, from Equation 1, the total time complexity of the algorithm InPos is $\theta(n)$. In other words, algorithm InPos considers each node according to the type. It considers elements that have two children twice, once for building their right subtrees, and once for their left subtrees. Likewise, it considers the elements that have one child, only once, for building either their right or left subtree, and the algorithm never considers the leaf node. That is to say, the variable $CurrentNode$ never initiates for leaf nodes. Therefore, from this point of view the total time complexity of the algorithm is at most $\theta(2n)$ which is $\theta(n)$.

4.2 Space complexity of InPos

Here, we investigate the space complexity of the algorithm InPos. Since the algorithm regards four attribute per node *n* the matrix-based structure that it returns has four rows and *n* columns. We let $M(n)$ denote to the space occupied by the matrix, which is equal to

$$M(n) = \theta(4n) \\ = \theta(n). \tag{2}$$

From Equation 2, the space occupied by the returning matrix is $\theta(n)$. On the other hand, in the InPos algorithm all commands take constant space except the dynamic space occupied by the stack, and so they can be discarded. The space that the stack takes is at most equal to the number of the nodes with one child, since only these nodes push into the stack. As was discussed above, when the resulting binary tree has a right chain shape, all the one child nodes push into the stack and the height of the stack would be $\theta(n)$ in this condition. In other situations it would be less, so the stack takes the $O(n)$. Hence, the total space complexity of the algorithm InPos is equal to the space occupied by the return matrix and the stack, which is

$$S(n) = \theta(n) + O(n) \\ = \theta(n). \tag{3}$$

Therefore, from Equation 3 the total space complexity of the algorithm InPos is linear.

5 Discussion and future work

Here we discus our result incuding camparison to the previously mentioned studies. As we proved in section 4, the algorithm InPos takes linear time and space, while the other study that uses inorder and postorder traversals to reconstruct a binary tree, takes $O(n^2)$. So, the algorithm InPos works much effectively. Furthermore, the time and space that this algorithm takes are optimal according to Theorem 5 and Theorem 6.

Theorem 5. *The reconstruction of a binary tree from its traversals, takes at least $\theta(n)$ time, where n is the number of elements in either traversal.*

Proof. In the reconstruction process of a binary tree from its traversals, since every node should be visited at least once, this process takes at least $\theta(n)$, where n is the number of elements in either traversal.

□

Theorem 6. *The reconstruction of a binary tree from its traversals, takes at least $\theta(n)$ space, where n is the number of elements in either traversal.*

Proof. In the reconstruction process of a binary tree from its traversals, since every node should be stored, this process takes at least $\theta(n)$ space, where n is the number of elements in either traversal.

□

Algorithm	Using Traversals	Time Complexity	Space Complexity
[4]	inorder and preorder	$O(n^2)$	not mentioned
[4]	inorder and postorder	$O(n^2)$	not mentioned
[6, 7]	inorder and preorder	$O(nlogn)$	$O(n)$
[6, 7]	inorder and preorder	$O(n)$	$O(n^2)$
[20]	inorder and preorder	$O(n)$	$O(n)$
[5]	inorder and preorder	$O(n)$	$O(h)$
[10]	inorder and preorder	$O(n)$	$O(nlogn)$
[3]	inorder and preorder	$O(n)$	$O(nlogn)$
[11]	preorder and reversed preorder	$O(max(n, 2^m))$	$O(n)$
InPos	inorder and postorder	$\theta(n)$	$\theta(n)$

Table 1: Proposed algorithms to reconstruct a binary tree from its traversals. n, h, and m respectively denote to number of elements in one of the traversal, height of the tree, and the number of one-child nodes.

The algorithm InPos takes linear time and space which are optimal. As is seen from the Table 1, across the presented algorithms to reconstruct a binary tree from its inorder and preorder traversals, only the algorithm proposed by [20] has linear time and space complexity, which are optimal. The authors of the algorithm presented in [5] have claimed that their algorithm takes $O(h)$ space where h is the height of the reconstructed binary tree which may vary from $logn$ to n. This is seemingly optimal, since its maximum will be n. But not the height is at stake during the reconstruction process, the algorithm takes $O(n)$ space, in the end, since every node should be stored (Theorem 6), where n is the number of the elements in one of the traversals. However, these two algorithms were proposed to reconstruct a binary tree only from its inorder and preorder traversals. Among the algorithms in Table 1, there is one study focusing on both inorder and postorder traversals that takes $O(n^2)$ time, but its space complexity is not mentioned . Here, our algorithm works more effective than it. In fact, our algorithm works in optimal linear time and space. Besides, during the reconstruction of a binary tree from its inorder and postorder traversals, it defines the type each tree. This will make it more convenient to work up a reconstruction tree. For many reasons the type of a node is important. The external nodes in both Huffman Coding [15] as well as decision trees [1, 19] represent the symbols and classes that can be found in linear time and space by this algorithm. At the end, it returns a matrix-based structure accessible to any structural information of the tree in linear time by any tree traversals, like BFS (Beardth-first search) or DFS (Depth-first search) [9]. Although the most common representation of a binary tree is a linked list [9] many programming languages, such as MATLAB, do not support it. Thus, the representation of a binary tree with linked lists may not be embeddable in many cases, while because of matrix-based structure of the binary tree beside the postorder traversal, can be used by many programming languages, and one can access any structural information of the resulting tree in linear time. It also does not bind the users only to some tree traversals. This representation enables any traversal of binary trees. This representation is also suggested by [9].

Furthermore, by applying both Theorem 2, Theorem 3, and Theorem 7 one can rebuilt a binary tree from inorder and postorder traversals from a different view point.

Theorem 7. *A two-child node x has a left child y and a right child z, such that both y are z are leaves if and only if y, x, z and y, z, x are in inorder and postorder traversals respectively.*

Proof. ($\Rightarrow$): According to inorder traversal, since the node y and z are a leaf, the node the nodes y,x,z will be visited respectively. Based on postorder traversal, the nodes y,z,x will be visited respectively.
($\Leftarrow$): On the contrary, let x is not a two-child node with the two leaf left child y and the leaf right child z. We have the following cases:

1. The node x is leaf. Let its parent is w. Without loss of generality, we let x is the left child of the node w. We have the following cases:

    (a) The nodes y and z are placed in right subtree of the node w. According to postorder traversal, in general, we have either $\ldots$,x,$\ldots$,y,$\ldots$,z,$\ldots$,w, $\ldots$ or $\ldots$,x,$\ldots$,z,$\ldots$,y,$\ldots$,w, $\ldots$. It is evident that the node x is always precedes the nodes y and z, which is contradiction.

    (b) The nodes y and z are in a subtree rooted from a node except w. According to inorder traversal, in general, we have $\ldots$,x,w,$\ldots$. It is evident the node x always precedes the node w, thus it is contradiction.

2. The node x is one child. Let its child is w. Without loss of generality, we let the node w be the left child of the node x. We have the following cases:

    (a) The both nodes y and z are placed in left subtree of the node w. According to inorder traversal, in general, we have either $\ldots$,y,$\ldots$,z,$\ldots$,w,x,$\ldots$ or $\ldots$,z,$\ldots$,y,$\ldots$,w,x,$\ldots$. It is evident that the node w always precedes the node x, thus it is contradiction.

    (b) The nodes y and z are places in right subtree of the node w. According to inorder traversal, in general, we have either $\ldots$,w,$\ldots$,y,$\ldots$,z,$\ldots$,x,$\ldots$ or $\ldots$,w,$\ldots$,z,$\ldots$,y,$\ldots$,x,$\ldots$. It is evident that the node x is always followed by the nodes y and z, thus it is contradiction.

    (c) The nodes y and z are placed elsewhere in the tree rather than the subtrees of the node w. According to inorder traversal, in general, we have one of the following pattern; $\ldots$,y,$\ldots$,z,$\ldots$,w,$\ldots$,x,$\ldots$ or $\ldots$,z,$\ldots$,y,$\ldots$,w,$\ldots$,x,$\ldots$ or $\ldots$,y,$\ldots$,w,$\ldots$,x,$\ldots$,z or $\ldots$,z,$\ldots$,w,$\ldots$,x,$\ldots$,y,$\ldots$. It is evident that in none of the conditions the three consecutive elements of x,y,z never happen, regardless of the gaps between sequences, thus it is contradiction.

3. The node x is a node with two children u and v such that the nodes u and v are the left and right children of the x respectively, and they are leaves. The nodes and are placed elsewhere. According to postorder traversal, in general we have $\ldots$,u,v,x. It is evident that the node u always precedes the node x and the node v always followed by the node x, thus it is contradiction. It should be noted that in this case by the assumption, we can consider that the nodes u and v are leaves.

Therefore, the two-child node x has a left child y and a right child z, such that both y are z are leaves

□

Based on Theorem 2 and Theorem 3 the one-nodes child can be determined, and depending on its child position and considering two distinct notations which are not contained in the these traversal values, these one-child with their only node can be merged into a new element with new value containing either notions, as mentioned. Similarly, based on Theorem 7, two-child nodes can be defined. Then each two-child node with their children can be merges them into new element regarding two new distinct notations from whom said for Theorem 2 and Theorem 3 inorder to distinguish between parent, left child, and right child values. Therefore, by using the three theorem- Theorem 2, Theorem 3, and Theorem 7, for each node its position in the reconstruction binary tree as well as their attributes- $parent$, $leftchild$, and $rightchild$ can be defined, and the binary tree can be reconstructed uniquely.

6 Conclusion

Binary trees [9, 22] are essential structures in computer science [3]. Tree traversals are therefore are across the most paramount topics in CS. The process of reconstructing a binary tree from its traversals first was described by Knuth [17] and later became its application omniscient. In Huffman Coding [15], for instance, inorder to increase the efficiency instead of transmission of the complete tree, its traversals are sent either inorder and preorder or inorder postorder traversals, and the tree will be reconstructed in the receiver [3]. Now, it is well-known, given inorder traversal along with preorder or postorder traversal of a binary tree, the binary can be rebuilt uniquely. Many algorithms have been presented to reconstruct a binary tree from inorder and preorder traversals, among them is the algorithm proposed by [20]. It has optimal time and space complexity. However, there is one study [4] focusing on reconstructing a binary tree from its inorder and postorder traversals, such that this takes $O(n^2)$ time. Here, was proposed InPos, an improved algorithm to reconstruct a binary tree from its inorder and postorder traversals. We proved that the algorithm takes optimal time and space which both are linear. Finally InPos returns a matrix-based structure enabling use by every programming language and all structural information of a binary tree can be accessed by InPos in linear time and space by every tree traversal. What is more, during the reconstruction process it determines the type of each node: with two children, with one child, or with no child. So this algorithm also could be applied in linear time and space to optimize decision trees in the applications to decision support, machines learning and pattern recognition systems [1, 19].

References

[1] Alpaydin, E. *Introduction to Machine Learning*, 2nd ed. The MIT Press, 2010.

[2] Andersson, A., and Carlsson, S. Construction of a tree from its traversals in optimal time and space. *Information Processing Letters 34*, 1 (1990), 21–25.

[3] Arora, N., Kaushik, P. K., and Kumar, S. Iterative method for recreating a binary tree from its traversals. *International Journal of Computer Applications 57*, 11 (November 2012), 6–13.

[4] Burgdorff, H., Jajodia, S., Springsteel, F. N., and Zalcstein, Y. Alternative methods for the reconstruction of trees from their traversals. *BIT Numerical Mathematics 27*, 2 (1987), 133–140.

[5] Cameron, R., Bhattacharya, B., and Merks, E. Efficient reconstruction of binary trees from their traversals. *Applied Mathematics Letters 2*, 1 (1989), 79–82.

[6] Chen, G., Yu, M., and Liu, L. Nonrecursive algorithms for reconstructing a binary tree from its traversals. In *Computer Software and Applications Conference, 1988. COMPSAC 88. Proceedings., Twelfth International* (1988), IEEE, pp. 490–492.

[7] Chen, G.-H., Yu, M., and Liu, L.-T. Two algorithms for constructing a binary tree from its traversals. *Information processing letters 28*, 6 (1988), 297–299.

[8] Chen, W., and Udding, J. T. Program inversion: More than fun! *Science of Computer Programing 15*, 1 (Nov. 1990), 1–13.

[9] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to Algorithms*, 3rd ed. The MIT Press, 2009.

[10] Das, V. V. A new non-recursive algorithm for reconstructing a binary tree from its traversals. In *Advances in Recent Technologies in Communication and Computing (ARTCom), 2010 International Conference on* (2010), IEEE, pp. 261–263.

[11] Deaconu, A. Optimal time and space complexity algorithm for reconstruction of all binary trees from pre-order and post-order traversals. In *7th conference on Operational research Conference, 2005. Proceedings.* (2006).

[12] Engelfriet, J. The time complexity of typechecking tree-walking tree transducers. *Acta Informatica 46*, 2 (Mar. 2009), 139–154.

[13] Habermehl, P., Iosif, R., and Vojnar, T. Automata-based verification of programs with tree updates. *Acta Informatica 47*, 1 (2010), 1–31.

[14] Hikita, T. Listing and counting subtrees of equal size of a binary tree. *Information processing letters 17*, 4 (1983), 225–229.

[15] Huffman, D. A. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE 40*, 9 (1952), 1098–1101.

[16] Knott, G. D. A numbering system for binary trees. *Communications of the ACM 20*, 2 (1977), 113–115.

[17] Knuth, D. E. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms.* Addison Wesley Longman Publishing Co., Inc., 1997.

[18] Knuth, D. E. *The Art of Computer Programming, Volume 3 (2nd Ed.): Sorting and Searching.* Addison Wesley Longman Publishing Co., Inc., 1998.

[19] Kuncheva, L. I. *Combining Pattern Classifiers: Methods and Algorithms.* Wiley-Interscience, 2004.

[20] Mäkinen, E. Constructing a binary tree from its traversals. *BIT Numerical Mathematics 29*, 3 (1989), 572–575.

[21] Reingold, E. M., and Hansen, W. J. *Data Structures in Pascal.* Little, Brown & Co. Inc., 1986.

[22] Skiena, S. S. *The Algorithm Design Manual*, 2nd ed. Springer Publ. Inc., 2008.

[23] Slough, W., and Efe, K. Efficient algorithms for tree reconstruction. *BIT Numerical Mathematics 29*, 2 (1989), 361–363.

[24] Takaoka, T. O(1) time algorithms for combinatorial generation by tree traversal. *The Computer Journal 42*, 5 (1999), 400–408.