The Binary Tree Roll Operation: Definition, Explanation and Algorithm

Article *in* International Journal of Computer Applications · April 2012

DOI: 10.5120/6931-9425

2 authors:

Адријан Божиновски
University American College Skopje

Nevena Ackovska
Ss. Cyril and Methodius University in Skopje

International Journal of Computer Applications (0975 – 8887)
Volume 46– No.8, May 2012

The Binary Tree Roll Operation: Definition, Explanation and Algorithm

Adrijan Božinovski
School of Computer Science and Information Technology
University American College Skopje
Treta Makedonska Brigada 60
1000 Skopje, Macedonia

Nevena Ackovska
Faculty of Computer Science and Engineering
University "St. Cyril and Methodius"
Rugjer Boškoviḱ 16
1000 Skopje, Macedonia

ABSTRACT
The paper introduces an operation on a binary tree, called binary tree roll, or roll of a binary tree. Two versions of the binary tree roll, counterclockwise and clockwise, are presented. The operations are mathematically defined and graphically presented. It is explained how the binary tree roll actually coincides with the process of turning the entire tree 90 degrees counterclockwise or clockwise. To visually explain and perform the roll operation, the concepts of a wedge node, true ancestor, illusory ancestor, illusory root and illusory ancestral stem of nodes are introduced, as well as the visual operations of turning and downshift. Both roll operations are implemented using programming algorithms. The algorithms are explained, and all the situations that might be encountered during processing the roll operation are examined and resolved. Thus, the paper gives a mathematical introduction of both binary tree roll operations, gives their visual explanations and offers algorithms for their implementations using a computer.

General Terms
Algorithm, binary tree

Keywords
binary tree, roll, operation, turning, downshift, algorithm

1. INTRODUCTION
Trees are fundamental concepts in computer science, and are frequently used to keep track of ancestors or descendants, sports tournaments, organizational charts of large corporations and so on [1]. Trees arise naturally as means to describe state spaces during the course of recursive algorithm processing, such as backtracking [2, 3], and also games (like tic-tac-toe, chess etc). Trees are composed of nodes and edges, the latter serving as links between nodes. The term "tree" implicitly carries an assumption that the tree is rooted, meaning that a special node is designated as the root of the tree, and all nodes that are connected to it are its sub-nodes (also called children or descendants) – the root is assigned the highest order in the hierarchy of the tree.

Each (rooted) tree has levels of hierarchy, i.e. a set of nodes having the same number of predecessors, or ancestors, from the root. Commonly, the predecessor nodes are displayed higher in the hierarchy, whereas the descendant nodes are displayed lower. The root is displayed the highest, and it is the only node that has no predecessor of its own.

Some trees are ordered inside a level. In ordered trees, the order in which the sub-nodes of a certain node (called the sub-nodes' parent or ancestor) are arranged is significant – each sub-node has an assigned position order relative to other sub-nodes having the same parent node. Binary trees are special kinds of trees, where each node (including the root) has at most two sub-nodes. Thus, in a binary ordered tree, the notions of a left and right sub-node are commonly used.

Tree traversal means processing every node by performing a certain function on it exactly once (a basic function would be to print out the node's information, for example). Since trees can be viewed as acyclic undirected graphs [4], the basic graph traversal techniques, namely depth-first and breadth-first traversals, are also applicable to trees. Adding the additional constraints of a root and at most two sub-nodes to every node, so as to obtain a binary tree, yields three depth-first traversal techniques, which are preorder, inorder and postorder, and one breadth-first traversal technique, which is level-order traversal [1].

The notions of "function" and "operation" on a binary tree are commonly used as synonyms (e.g. [5, 6, 7]). However, in this paper, a distinction will be made. A function on a binary tree is a procedure that does not change the internal structure of the binary tree. Thus, all traversals of the tree would be functions, as well as finding the height of the tree, the width of the tree, the number of nodes in the tree etc. On the other hand, an operation on a binary tree is a procedure that changes the internal structure of the binary tree. In other words, insertion of a node in the tree, deletion of a node in the tree, tree rotation [8] etc would all be operations on a binary tree. Because the structure of the binary tree changes when an operation is performed on it, promotion and demotion of nodes necessarily takes place, in a sense that nodes get positioned higher or lower in the hierarchy (i.e. at a shorter or longer distance from the root, respectively) as a result of the operation.

This paper introduces the operation named binary tree roll or roll of a binary tree. It can be applied on any binary tree. Two versions of the roll operations, the counterclockwise and the clockwise roll, will be defined and explained, both mathematically and visually. It will be shown that the two operations are unambiguous and also inverse to one another. Further in the paper, algorithms for their implementations will be presented. Since research in the field of binary trees is ongoing, both theoretically [9, 10, 11] and practically [12, 13, 14], this paper provides a new insight into the structure and possibilities for working with binary trees.

2. THE DEFINITIONS
The basis for both the counterclockwise and clockwise roll operation is the well known fact that a binary tree can unambiguously be generated given its preorder and inorder traversals, or its inorder and postorder traversals. In other

words, given a binary tree's inorder traversal, along with either one of its preorder or postorder traversals, the original binary tree can be reconstructed [15], and this reconstruction is unique, whereas for given preorder and postorder traversals only, the reconstruction is not unique [16]. It is important to mention that this is true if all nodes of the binary tree are different, i.e. can be differentiated from one another. The easiest way to differentiate the nodes of a binary tree is using their data fields, so therefore, throughout this paper, it will be assumed that all the elements of every binary tree in question will be different from one another.

The counterclockwise roll of a binary tree, abbreviated CCW() henceforth, can now be defined. Given binary trees T_1 and T_2, as well as the preorder(), inorder() and postorder() functions, which yield the respective traversals, operation CCW() is defined as

$$CCW(T_1) = T_2 \Leftrightarrow (preorder(T_1) = inorder(T_2) \wedge inorder(T_1) = postorder(T_2)) \tag{1}$$

In other words, upon CCW(), the preorder traversal of the original tree is identical to the inorder traversal of the tree obtained by the counterclockwise roll, and the inorder traversal of the original tree is identical to the postorder traversal of the tree obtained by the counterclockwise roll.

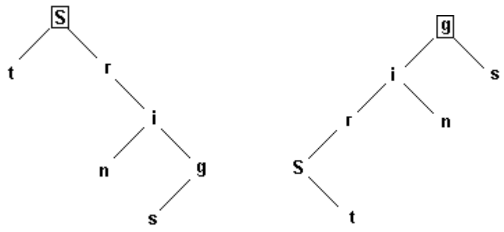
Likewise, the clockwise roll of a binary tree, abbreviated CW() henceforth, can be defined as

$$CW(T_1) = T_2 \Leftrightarrow (inorder(T_1) = preorder(T_2) \wedge postorder(T_1) = inorder(T_2)) \tag{2}$$

Similarly, upon CW(), the inorder traversal of the original tree is identical to the preorder traversal of the tree obtained by the clockwise roll, and the postorder traversal of the original tree is identical to the inorder traversal of the tree obtained by the clockwise roll.

3. EXAMPLES

Figure 1 shows an example of these processes and their results. The preorder and inorder traversals of T_1 are equal to the respective inorder and postorder traversals of T_2. In other words, using the preorder traversal of T_1 as the inorder traversal of some new tree and the inorder traversal of T_1 as the postorder traversal of the new tree, and thus unambiguously reconstructing the new tree, will yield tree T_2 as a result. Conversely, taking the inorder traversal of T_2 and using it as the preorder traversal of a new tree and the postorder traversal of T_2 and using it as the inorder traversal of the new tree, unambiguously reconstructing such a tree will yield tree T_1 as a result. Upon any roll operation, the root of the newly obtained binary tree is different from the root of the original tree, which is a consequence of definitions (1) and (2).



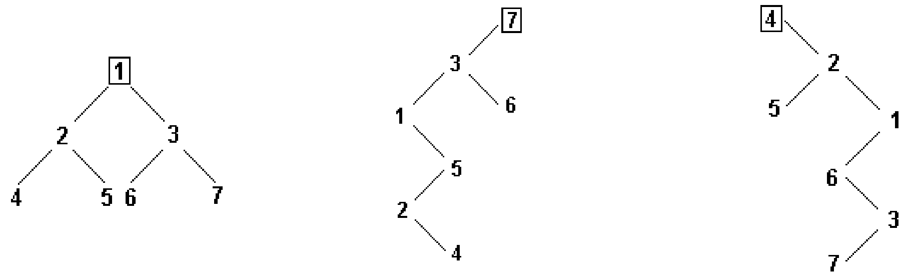
T_1 *T_2*

Preorder:	S,t,r,i,n,g,s	Preorder:	g,i,r,S,t,n,s
Inorder:	t,S,r,n,i,s,g	Inorder:	S,t,r,i,n,g,s
Postorder:	t,n,s,g,i,r,S	Postorder:	t,S,r,n,i,s,g

Figure 1: Example of a binary tree roll. CCW(T_1) = T_2. CW(T_2) = T_1. The root of every tree is marked with a rectangle.

Figure 2 shows another example. More specifically, Figure 2a shows a balanced tree, Figure 2b shows the result obtained by counterclockwise roll of the tree in Figure 2a and Figure 2c shows the result obtained by clockwise roll of the tree in Figure 2a.

Viewing trees T_1 and T_2 in Figure 1 leads to the thought that the two trees are turned 90 degrees respective to one another. Repeated testing has shown that this is indeed the case, and this has led to the roll operation originally being called "rotation of a binary tree" [17]. However, this is not always the case, as presented in Figure 2. The trees in Figure 2b and Figure 2c are clearly not obtained by simply rotating the tree in Figure 2a for 90 degrees counterclockwise and clockwise, respectively, even though, when viewing their respective traversals, it can be seen that they conform to definitions (1) and (2) for counterclockwise and clockwise roll, respectively. In the next section it will be explained why the term "binary tree roll" (or "roll of a binary tree") is more appropriate for the given operation and how the "rotation of a binary tree" can be thought of as a special case of the roll of a binary tree. Moreover, the term "roll" will be used to denote the operation itself, rather than the result of it (as previously implied in [17]). Also, in all figures, the root of every displayed tree will be marked with a rectangle, for clarity.



Preorder:	1, 2, 4, 5, 3, 6, 7	Preorder:	7, 3, 1, 5, 2, 4, 6	Preorder:	**4, 2, 5, 1, 6, 3, 7**
Inorder:	**4, 2, 5, 1, 6, 3, 7**	Inorder:	1, 2, 4, 5, 3, 6, 7	Inorder:	*4, 5, 2, 6, 7, 3, 1*
Postorder:	*4, 5, 2, 6, 7, 3, 1*	Postorder:	**4, 2, 5, 1, 6, 3, 7**	Postorder:	5, 7, 3, 6, 1, 2, 4
	a)		b)		c)

Figure 2: Roll of a balanced binary tree. a) The original tree, along with its preorder (underline), inorder (bold) and postorder (italic) traversals, which are used to build the b) counterclockwise and c) clockwise rolled tree

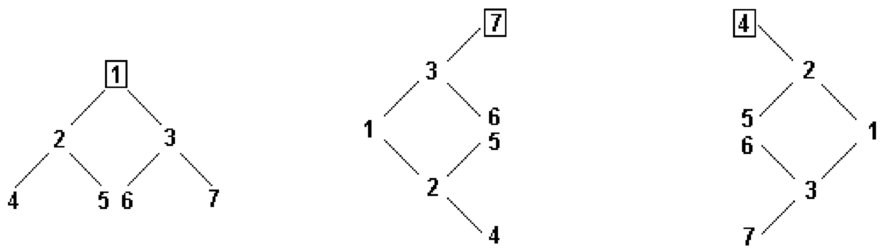
4. VISUAL EXPLANATION

In order to link the roll operation with the notion of a tree being turned for 90 degrees in some direction, the *turning* of a binary tree is defined as a visual operation, in which the tree is turned 90 degrees in some direction, so that the rightmost or leftmost node in the original tree becomes the root of the new tree, i.e. is placed the highest in the hierarchy. In the first case, a *counterclockwise turn* takes place, and the second case a *clockwise turn* takes place. Since this operation is purely visual, it can be stated that only the links get modified so that the new tree is obtained, whereas the node elements remain in their original, i.e. "unturned", positions.

Consider the balanced tree in Figure 3a (it is identical to the tree in Figure 2a but is repeated here for clarity). Visually, every ancestor node in the tree should be displayed higher than its descendant nodes, but, upon turning (Figure 3b and Figure 3c), some ancestor nodes are displayed *lower* than their descendant nodes. Specifically, in Figure 3b node '2' has one of its descendants (node '5') displayed higher in the picture, and in Figure 3c the same applies to node '3' (one of its descendants, node '6', is displayed higher in the picture). If the level of hierarchy is defined as the number of edges that a node is farther from the root, then it is clear that some internal (i.e. non-root) nodes are represented visually higher than their ancestor nodes. Such nodes form an *illusory ancestral stem* of nodes, containing nodes that are displayed higher than a certain node, but are essentially lower in the hierarchy (i.e. further from the root of the tree) than it. A node that has an illusory ancestral stem of nodes is called a *wedge node*, and it contains both a *true ancestor* (which is closer to the root of the tree) and an *illusory ancestor*, which is the first node connected to the wedge node along the illusory ancestral stem.

It is possible that the illusory ancestor be the only node of the illusory ancestral stem (as depicted in Figure 3), but it's also possible that the illusory ancestral stem contain more than one node. In that case, the *illusory root* is important, which represents the node that is as displayed visually the highest along the illusory ancestral stem of nodes. If the illusory ancestral stem contains only one node, i.e. only the illusory ancestor, then it is identical to the illusory root.

To maintain the visual representation that ancestors are displayed higher than the descendants, another visual operation needs to be performed, that would result with another tree, which does not have any illusory ancestral stems. In other words, all parts of the tree containing illusory ancestral stems would have to be repositioned, so that all ancestor nodes would be displayed higher in the hierarchy than their descendants.



a) b) c)

Figure 3: Turning of binary trees. a) The original fully balanced tree. b) The tree from a) is turned 90 degrees counterclockwise, with the new root of the tree set to '7'. The wedge node is '2', its true ancestor is '1' and its illusory ancestor and illusory root is '5', because the illusory ancestral stem of nodes consists only of node '5'. c) The tree from a) is turned 90 degrees clockwise and the new root of the tree is set to '4'. The wedge node is '3', its true ancestor is '1' and its illusory ancestor and illusory root is '6', because the illusory ancestral stem of nodes consists only of node '6'.

Such an operation is named *downshift*. When a wedge node appears upon turning, its *true ancestor needs to be linked to the illusory root, instead of with the wedge node, in the same direction of linking that the wedge node was linked to*. Essentially, that would mean visually pushing downwards the entire illusory ancestral stem; hence the term “downshift”. In counterclockwise roll, the downshift is to the left, and in clockwise roll, the downshift is to the right. *Left downshift* happens upon CCW() and *right downshift* happens upon CW(), when necessary (there are cases, which will be explained further in this chapter, when a downshift isn’t necessary to complete the roll operation).

Figure 4 demonstrates the process of obtaining a roll of a binary tree using turning and downshift, after which the results are the same as in Figure 2b and Figure 2c. Figure 5 displays CCW() using turning and downshift in a tree in which the illusory ancestral stem contains more than one node, and therefore the illusory ancestor and the illusory root are different. Again, the linking of the true ancestor with the illusory root is the key to performing the downshift.

Depending on the internal structure (i.e. topology) of the tree, performing a roll operation using turning and downshift might take several steps to complete. Figure 6 presents a case where, in order to complete a CW() operation, a clockwise turn and two consecutive right downshifts are needed. This is because two right sub-nodes (namely ‘3’ and ‘7’) contain left sub-trees of their own, and they all cause illusory ancestral stems after a clockwise turn, which must be handled by two consecutive downshifts. It is worth mentioning that, no matter the sequence of downshifts, as long as all illusory ancestral stems of nodes are downshifted, the resulting tree will comply with the appropriate definition for the corresponding roll operation (in Figure 6, the ending tree complies with definition (2), for clockwise roll).

In most cases, the downshift is necessary if the roll operation gets performed visually using turning. However, there are cases when the downshift is not necessary, and the roll yields the same result as the turn, both visually and theoretically. Thus, CCW() is equal to a counterclockwise turn *if no left sub-node in the tree has a right sub-node of its own* (tree T_1 in Figure 1 is such an example). Likewise, CW() is equal to a clockwise turn *if no right sub-node in the tree has a left sub-node of its own* (tree T_2 in Figure 1 is such an example).

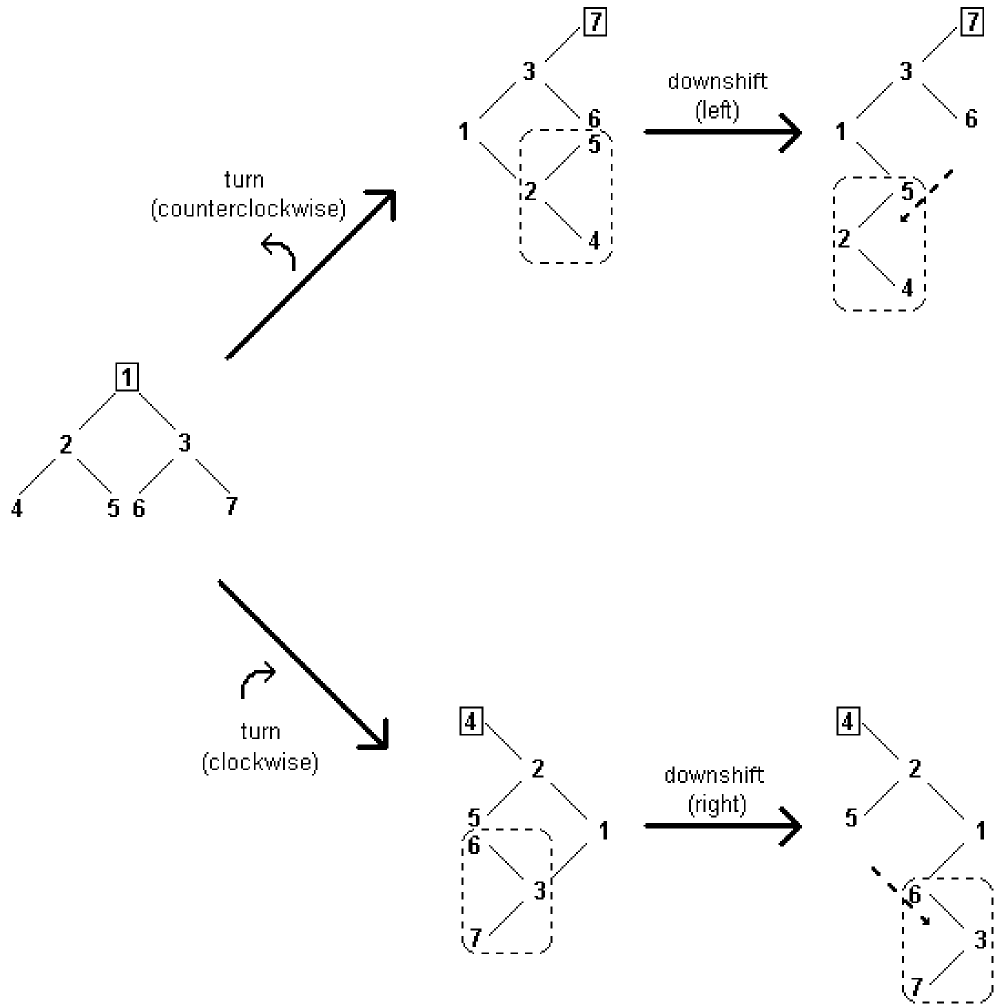
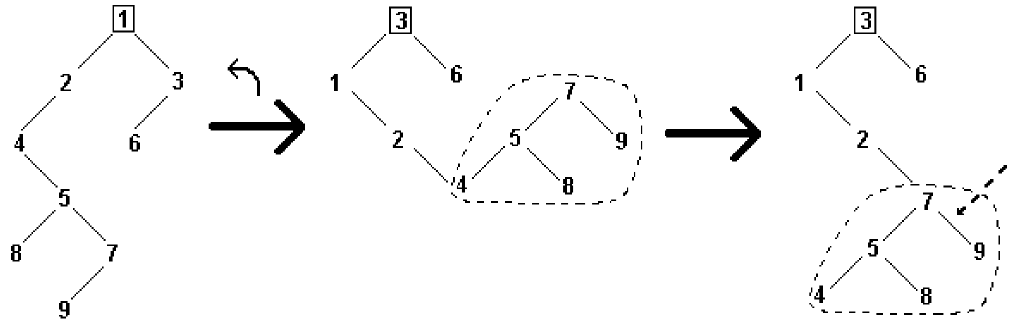


Figure 4: Achieving binary tree roll using turning and downshift



Preorder: 1, 2, 4, 5, 8, 7, 9, 3, 6
Inorder: **4, 8, 5, 9, 7, 2, 1, 6, 3**
Postorder: *8, 9, 7, 5, 4, 2, 6, 3, 1*

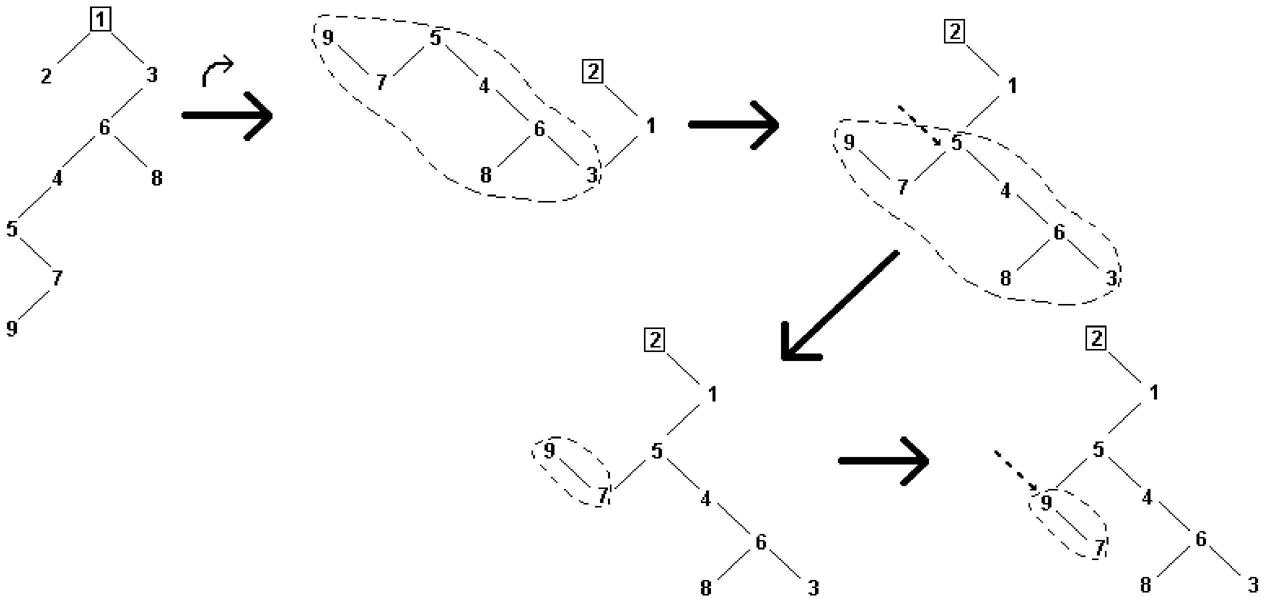
Preorder: 3, 1, 2, 7, 5, 4, 8, 9, 6
Inorder: 1, 2, 4, 5, 8, 7, 9, 3, 6
Postorder: **4, 8, 5, 9, 7, 2, 1, 6, 3**

Figure 5: CCW() of a tree using turning and downshift. After the turn, the new root of the tree is '3', the wedge node is '4', its true ancestor is '2', its illusory ancestor is '5' and its illusory root is '7', since the illusory ancestral stem of nodes consists of nodes '5' and '7'. The downshift is performed by linking the true ancestor ('2') with the illusory root ('7'), instead of the wedge node ('4'), in the same direction of linking – the illusory root becomes the right sub-node of the true ancestor, just like the wedge node used to be. The traversals of the starting tree (bottom left) and the ending tree (bottom right) are given for clarity

It is worth mentioning that the turning and the downshift operations, as well as the notions of a wedge node, true ancestor, illusory stem of ancestors, illusory ancestor and illusory root, are *visual* only and don't actually take place or appear when performing the roll operations defined by definitions (1) and (2). However, they are useful for envisioning the roll processes and obtaining the rolled trees without first obtaining the traversals needed to generate them.

5. MUTUAL INVERSENESS OF THE ROLL OPERATIONS

From the definitions of CCW() and CW(), i.e. definitions (1) and (2), it follows immediately that both operations are inverse respective to one another (trees T_1 and T_2 in Figure 1 give a good visual demonstration of this property). As another example, if T_1 is the tree represented in Figure 3a and T_2 is the tree represented in Figure 3b, CCW(T_1) = T_2 and CW(T_2) =



Preorder: 1, 2, 3, 6, 4, 5, 7, 9, 8
Inorder: **2, 1, 5, 9, 7, 4, 6, 8, 3**
Postorder: 2*, 9, 7, 5, 4, 8, 6, 3, 1*

Preorder: **2, 1, 5, 9, 7, 4, 6, 8, 3**
Inorder: 2*, 9, 7, 5, 4, 8, 6, 3, 1*
Postorder: 7, 9, 8, 3, 6, 4, 5, 1, 2

Figure 6: CW() of a tree using turning and consecutive downshifts. In the first downshift, the wedge node is '3', its true ancestor is '1', its illusory ancestor is '6' and its illusory root is '5', since the illusory ancestral stem of nodes consists of nodes '6', '4' and '5'. In the second downshift, the wedge node is '7', its true ancestor is '5' and its illusory ancestor and root is '9', since the illusory ancestral stem of nodes consists only of node '9'. When there are no more illusory ancestral stems, the roll operation is complete, which can be verified by the traversals of the original tree (bottom left) and the final tree (bottom right)

T_1. This can also be verified for the trees in Figure 3c (if considered as T_1) and Figure 3a (if considered as T_2). Testing the operations for any binary tree gives the same result. Therefore, it can be represented mathematically that, for any given binary tree T, it holds true that

$$CW\big(CCW(T)\big) = T;\ CW^{-1}(T) = CCW(T) \tag{3}$$

and

$$CCW\big(CW(T)\big) = T;\ CCW^{-1}(T) = CW(T) \tag{4}$$

6. ALGORITHMS FOR THE ROLL OPERATIONS

As a result of a roll operation on a given binary tree, a new binary tree gets produced, which contains all the elements of the original binary tree, but rearranged in a different hierarchical structure, so as to comply with definition (1) or (2), depending on the direction of roll. The most straightforward way to accomplish the roll in the desired direction is to obtain the traversals of the original tree and then reconstruct the new tree using the appropriate traversal combinations according to definition (1) or (2). However, since the tree obtained as the result of the roll always contains the same elements as the original tree, it is possible to generate an algorithm that will rearrange the elements so as to obtain the rolled tree without the need to obtain the traversals first. An algorithm for the roll operation will now be presented and explained.

Figure 7 shows the algorithm for CCW(). The algorithm is given in pseudocode, which most closely resembles the C++ programming language, and the lines are numbered, for a more detailed explanation. The algorithm is recursive and employs auxiliary variables defined within it, which are of the data type of a binary tree node. Normally, a node of a binary tree contains an information field, as well as links to the left sub-node and right sub-node of the same node type, called `lSn` and `rSn` respectively in the pseudocode. The information field can be of any data type, and does not need to be comparable (i.e. does not have to have relation operators, such as >, <, == or !=, defined for it). The `define` directive will be used to define (i.e. declare and initialize) auxiliary variables. It is assumed that the node element type can be compared for equality and inequality to `NULL` (the null element, containing no information or value) and any other node element data type. The assignment operator (=) is also assumed to be defined for this node element data type. For clarity, the function header and the subsequent recursive calls within the algorithm are presented with bold letters.

Line 1 defines the function header. The function return type is not listed, as the algorithm itself does not produce a return value (if a return type would have to be listed, `void` would be a good choice). The parameters that are requested are the root of the tree (it is also used as the root of sub-trees in the subsequent recursive calls) and the root's predecessor, i.e. the node that had the root of the (sub-)tree as its sub-node. Initially, the `predecessor` node is `NULL` (and needs to be passed as such upon initial function calls), since the root of the original tree has no ancestor node. The values of both the `root` and the `predecessor` nodes are guaranteed to change within the function calls (since the roll operation requires that the root and the internal structure of the tree change so as to obtain the resulting tree) and that's why they are called by reference, as indicated by the ampersand (`&`) preceding them, indicating that the changes, that they'll be subjected to, would need to be carried over as the algorithm processes further.

```
1.  CCW(&root, &predecessor)
2.  {
3.   if(root != NULL)
4.   {
5.    if(root.rSn == NULL)
6.    {
7.     root.rSn = root.lSn;
8.     root.lSn = NULL;
9.     CCW(root.rSn, root);
10.   }
11.   else
12.   {
13.    if(root.rSn.rSn == NULL)
14.    {
15.     root.rSn.rSn = root.rSn.lSn;
16.     root.rSn.lSn = root;
17.     root = root.rSn;
18.     root.lSn.rSn = root.lSn.lSn;
19.     root.lSn.lSn = NULL;
20.     if(predecessor != NULL)
21.         predecessor.rSn = root;
22.     CCW(root.lSn.rSn, root.lSn);
23.     CCW(root.rSn, root);
24.    }
25.    else
26.    {
27.     CCW(root.rSn, root);
28.     define leftmost = root.rSn;
29.     while(leftmost.lSn != NULL)
30.         leftmost = leftmost.lSn;
31.     leftmost.lSn = root;
32.     define newroot = root.rSn;
33.     root.rSn = NULL;
34.     root = newroot;
35.     if(predecessor != NULL)
36.         predecessor.rSn = root;
37.     CCW(leftmost.lSn, leftmost);
38.    }
39.   }
40.  }
41. }
```

Figure 7: The algorithm for the counterclockwise roll (i.e. the CCW() operation)

Line 3 gives the first test. The algorithm is designed to go through all nodes of the tree, so that the entire tree would be (recursively) rolled. Following some of the links (e.g. sub-nodes of leaves of the tree) will lead to `NULL` values, so the first test is for whether the algorithm will need to proceed at all. Should a `NULL` value be encountered, the algorithm (i.e. the current recursive call) ends immediately, as this first `if` test does not contain an `else` clause. This is actually the only trivial case in the algorithm and is thus used as the termination condition for the recursive calls.

Line 5 tests whether the root of the (sub-)tree contains a null right sub-node (i.e. does not contain a right sub-tree). If that is the case (presented visually in Figure 8), whatever is given as a left sub-node (which may be `NULL` as well) is placed as the right sub-node (Line 7), the left sub-node is set to `NULL` (Line 8) and the now right sub-tree is recursively rolled (Line 9).

The idea is that, upon CCW(), left sub-trees of the nodes become right sub-trees of the nodes. This is the first basic case, when there is no right sub-tree that needs to be processed, so the left sub-tree is put in its place and processed recursively.

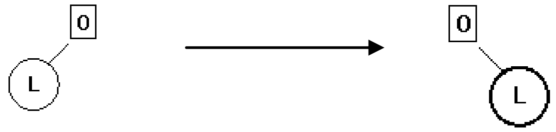


Figure 8: The first basic case: if the root (O) contains just a left sub-tree (L), upon CCW() it becomes the right sub-tree, while the root's left sub-tree is set to NULL. The now right sub-tree of the root is recursively processed using CCW() (bolded circle)

Lines 11 and 13 test for whether there is not more than one right sub-node. If so, this is the second basic case (presented visually in Figure 9). The left sub-node of the root's right sub-node will become the right sub-node of the root's right sub-node (Line 15), whereas the left sub-node of the root's right sub-node will point to the root (Line 16). The root's right sub-node will become the new root (Line 17). Since the former root is now the left sub-node of the new root, whatever was present as its left sub-node (it may be `NULL` as well) is placed as its right sub-node (Line 18), and its left sub-node is set to `NULL` (Line 19). Since the root changes when this case is reached, the predecessor's right sub-node should also be changed to point to the new root, if the currently processed node (i.e. root) does have a predecessor (Lines 20 and 21). Finally, the new root's left sub-node's right sub-node can be processed recursively (Line 22), as well as the new root's right sub-node (Line 23), minding to include their respective predecessor nodes as parameters as well.

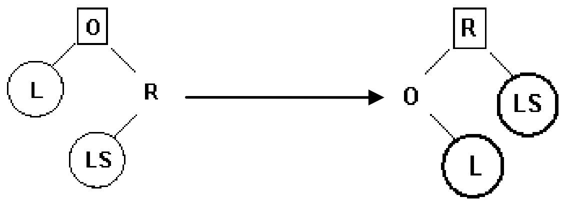


Figure 9: The second basic case: if the root (O) contains a left sub-tree (L), just a single right sub-node (R), which in turn contains just a left sub-tree of its own (LS), then, after the transformation, CCW() is recursively invoked upon L and LS in their new respective positions (bolded circles)

Line 25 introduces the most complex case, when the right sub-node has also a right sub-tree of its own (presented visually in Figure 10). To handle it, the algorithm first invokes a recursive call to the root's right sub-node (Line 27), so the function would reach one of the basic cases and handle them. Afterwards, the algorithm finds the leftmost sub-node of the root's right sub-tree (the "leftmost" node means a node containing no left sub-node of its own – Lines 28 to 31), so that the current root can be linked to it as its left sub-node. The original root's right sub-node will become the new root (Lines 32 and 34), after the original root's right sub-node gets set to `NULL` (Line 33). If a predecessor exists, its right sub-node is updated to point to the new root (Lines 35 and 36). Afterwards, CCW() is recursively invoked upon the former root in its new position (Line 37). Again, since the root changes when this case is encountered, the predecessor node needs to be taken into account and its right sub-node updated.

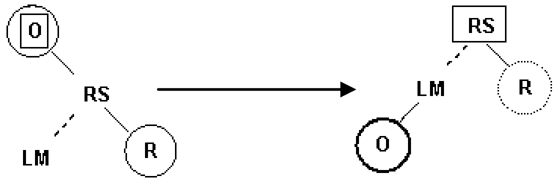


Figure 10: If the root (O) contains a right sub-node (RS) with a right sub-tree of its own (R), first CCW() is (recursively) invoked upon R, so it would be handled by a basic case, or this same case (if R contains a right sub-tree of its own). Then, it is necessary to find the leftmost sub-node (LM) of the root's right sub-node, i.e. of RS. In that case, O becomes the left sub-node of LM, whereas the right sub-node of O is set to NULL (so as not to point to RS anymore), and RS becomes the new root. CCW() is then recursively invoked upon O in its new position (bolded circle)

It may not be immediately apparent how the last case is connected to the roll operation. However, a closer look will reveal that it actually deals with the left downshift when CCW() is invoked upon stems of right sub-nodes (i.e. several nodes linked as right sub-nodes to one another). This case takes the root of the (sub-)tree and places it as the leftmost node of the root's right sub-node (which becomes the new root), but only *after* recursion calls have been invoked upon right sub-nodes progressively further down the stem, until one of the two basic cases, presented in Figure 6 and Figure 7, gets reached and handled. In this manner, a stem of right sub-nodes will progressively become a reversed stem of left sub-nodes, connected to the leftmost node of the sub-tree obtained as a result of handling one of the basic cases. Thus, left downshift will be achieved after a counterclockwise turn, progressively and recursively, one root at a time.

The algorithm for clockwise roll (i.e. CW()), presented in Figure 11, is a complete "mirror image" of the algorithm for CCW(). In fact, if everything "left" is replaced with "right" and vice versa in the CCW() algorithm, the CW() algorithm will be obtained (it is also necessary to replace "CCW" with "CW"). The procedure is completely analogous with the one for CCW().

7. CONCLUSION

A new operation, called binary tree roll, has been proposed. The counterclockwise and clockwise roll of a binary tree have been presented and defined. It has been shown that they are proper operations, in a sense that their results are unambiguous, and that they change the internal structure of the tree. The concepts of turning and downshift have been introduced as a means of simplifying and visualizing the concept of the binary tree roll. It has been shown that both roll operations are inverses to one another. The algorithm and code for the counterclockwise roll operation has been presented and explained, as well as the algorithm for clockwise roll operation. It is believed that the new operation will be useful in dynamical hierarchical structures, which need to introduce changes (e.g. new supervisor promotion) and yet preserve some properties of the original structure, such as hierarchical structure traversal.

```
CW(&root, &predecessor)
{
 if(root != NULL)
 {
  if(root.lSn == NULL)
  {
   root.lSn = root.rSn;
   root.rSn = NULL;
   CW(root.lSn, root);
  }
  else
  {
   if(root.lSn.lSn == NULL)
   {
    root.lSn.lSn = root.lSn.rSn;
    root.lSn.rSn = root;
    root = root.lSn;
    root.rSn.lSn = root.rSn.rSn;
    root.rSn.rSn = NULL;
    if(predecessor != NULL)
       predecessor.lSn = root;
    CW(root.rSn.lSn, root.rSn);
    CW(root.lSn, root);
   }
   else
   {
    CW(root.lSn, root);
    define rightmost = root.lSn;
    while(rightmost.rSn != NULL)
       rightmost = rightmost.rSn;
    rightmost.rSn = root;
    define newroot = root.lSn;
    root.lSn = NULL;
    root = newroot;
    if(predecessor != NULL)
       predecessor.lSn = root;
    CW(rightmost.rSn, rightmost);
   }
  }
 }
}
```

Figure 11: The algorithm for the clockwise roll (i.e. the CW() operation)

8. REFERENCES

[1] Sedgewick, R. 1998. Algorithms in C++, Parts 1-4: Fundamentals, Data Structures, Sorting, Searching. Addison-Wesley.

[2] Brassard, G. and Bratley, P. 2002. Fundamentals of Algorithmics. Prentice Hall of India

[3] Bozinovski, A. and Bozinovski, S. 2004. N-queens pattern generation: An insight into space complexity of a backtracking algorithm. In Proceedings of the 3rd International Symposium on Information and Communication Technologies. Las Vegas, Nevada, USA, 281-286

[4] Sedgewick, R. 2002. Algorithms in C++, Part 5: Graph Algorithms. Pearson

[5] Freedman, R. Binary Tree Operations. Notes on CS340: Data Structures and Algorithm Analysis. Department of Computer Science, Northern Illinois University. http://faculty.cs.niu.edu/~freedman/340/340notes/340btreeop.htm. Accessed 10 January 2012

[6] Mostafa, H. 2005. Fast Binary Tree Operations. The Code Project. http://www.codeproject.com/KB/recipes/BinaryTree.aspx. Accessed 10 January 2012

[7] Kruse, G. W. 2007. Binary Tree Operations. CS240: Computer Science II. Department of Information Technology and Computer Science. Juniata College. http://jcsites.juniata.edu/faculty/kruse/cs240/bintree.htm. Accessed 10 January 2012

[8] Sleator, D. D., Tarjan, R. E., and Thurston, W. P. 1988. Rotation distance, triangulations, and hyperbolic geometry. Journal of the American Mathematical Society. Vol 1. No 3. 647-681

[9] Chen, W. Y. C. and Yang, L. L. M. 2008. On Postnikov's hook length formula for binary trees. European Journal of Combinatorics. Doi: 10.1016/j.ejc.2007.11.025

[10] Doshi, N., Sureja, T., Akbari, B., Savaliya, H., and Daxini, V. 2010. Width of a Binary Tree. International Journal of Computer Applications. Vol 9. No 2. 41-43

[11] Arora, N., Tamta, V. K., and Kumar, S. 2012. Modified Non-Recursive Algorithm for Reconstructing a Binary Tree. International Journal of Computer Applications. Vol 43. No 10. 25-28

[12] Zhou, A., Huang, S., and Wang, X. 2007. BITS: A Binary Tree Based Web Service Composition System. International Journal of Web Services Research. Vol 4. No 1. 40-58

[13] Duarte, E. P. Jr, Pires, K., and Tavares, R. A. E. 2010. An efficient strategy for storing and searching binary trees in WORM external memory. Journal of Information Science. Vol 36. No 6. 751-762

[14] Wang, D., Zheng, J., and Zhou, Y. 2011. Binary tree of posterior probability support vector machines. Journal of Zhejiang University – Science C. Vol 12. No 2. 83-87

[15] Burgdorff, H. A., Jajodia, S., Springsteel, F. N., and Zalcstein, Y. 1987. Alternative methods for the reconstruction of trees from their traversals. BIT Numerical Mathematics. Vol 27. No 2. 133-140

[16] Narahari, Y. 4.2 Binary Trees. Data Structures and Algorithms. Computer Science and Automation. Indian Institute of Science. http://lcm.csa.iisc.ernet.in/dsa/node87.html. Accessed 10 January 2012

[17] Božinovski, A. 2011. Linear-Time Binary Tree Generation Algorithms for Arbitrary Input String Reproduction upon Depth-First Traversal and Rotation of a Binary Tree. International Conference on Innovative Technologies IN-TECH. Bratislava, Slovakia. 58-61