Space Complexity Analysis of the Binary Tree Roll Algorithm

Adrijan Božinovski[1], George Tanev[1], Biljana Stojčevska[1], Veno Pačovski[1], Nevena Ackovska[2]

[1]*School of Computer Science and Information Technology, University American College Skopje, Macedonia*

[2]*Faculty of Computer Science and Engineering, University "Sv. Kiril i Metodij", Skopje, Macedonia*

[1]*bozinovski@uacs.edu.mk*, [1]*george.tanev@uacs.edu.mk*, [1]*stojcevska@uacs.edu.mk*, [1]*pachovski@uacs.edu.mk*, [2]*nevena.ackovska@finki.ukim.mk*

Contribution to the state of the art

DOI: **10.7251/JIT1701009B** UDC: **004.42:510.5**

Abstract: This paper presents the space complexity analysis of the Binary Tree Roll algorithm. The space complexity is analyzed theoretically and the results are then confirmed empirically. The theoretical analysis consists of determining the amount of memory occupied during the execution of the algorithm and deriving functions of it, in terms of the number of nodes of the tree *n*, for the worst - and best-case scenarios. The empirical analysis of the space complexity consists of measuring the maximum and minimum amounts of memory occupied during the execution of the algorithm, for all binary tree topologies with the given number of nodes. The space complexity is shown, both theoretically and empirically, to be logarithmic in the best case and linear in the worst case, whereas its average case is shown to be dominantly logarithmic.

Keywords: Binary Tree Roll Algorithm, space complexity, theoretical analysis, empirical analysis.

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 [19]. Trees are one of the basic data structures used in combinatorial algorithms [13], search techniques (e.g., [8, 4]), and game playing [17]. This paper also points out the use of binary trees for generating integer sequences, which are important in information forensics [20], cryptography [9], and security [12]. Binary trees have been shown to be very useful in mathematics and computer science and as such have been extensively studied. Several variations of the binary tree structure have been conceived, such as binary search trees, red-black trees [10], AVL trees [1], B-trees [3], and so on. Binary trees are often used as auxiliary data structures in other research endeavors, both practical (e.g., [18, 15]) and theoretical (e.g., [16, 14]), but occasionally are the subject of the research itself (e.g., [2]).

Binary Tree Roll is an operation by which all of the nodes of a binary tree are rearranged in such a way, so that two of the depth-first traversals of the newly obtained binary tree yield the same results as other two traversals of the original binary tree. The graphical representation of the newly obtained binary tree is that it appears to be rolled at a 90 degree angle (either counterclockwise or clockwise, depending on the direction of the applied roll operation) relative to the original binary tree; hence the name "Binary Tree Roll".

This operation was introduced and defined in [5]. There are two variants of the Binary Tree Roll Operation: a counterclockwise (CCW) and a clockwise (CW) roll. The counterclockwise roll of a binary tree, abbreviated as CCW(), is defined as follows. Given two binary trees T_1 and T_2, as well as their respective preorder(), inorder() and postorder() traversal functions, operation CCW() is defined as in Definition 1:

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

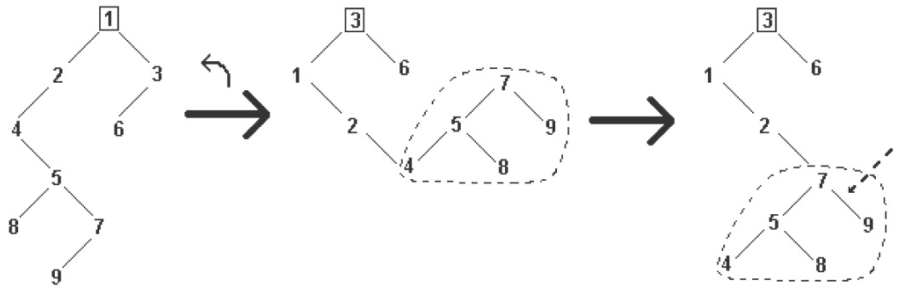
In other words, upon CCW(), the preorder traversal of the original tree is identical to the inorder traversal of the tree obtained by the counterclockwise roll, and the inorder traversal of the original tree is identical to the postorder traversal of the tree obtained by the counterclockwise roll.

Likewise, the clockwise roll of a binary tree, abbreviated as CW(), is defined as in Definition 2:

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

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.

A graphical explanation was given in [5], showing how the resulting binary tree is obtained visually, so as to comply with definition (1) or (2), depending on the direction of the roll. The *downshift* visual operation, illustrated in Figure 1, was also presented. It was shown that CCW() and CW() are inverses of each other, and algorithms for CCW() and CW() were given, which did not require obtaining the traversals of the input tree in order to generate the rolled tree.



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 1. *Graphical explanation of the CCW() algorithm, and an example of a downshift [5]*

Structurally, the algorithm presented in [5] contains a trivial case, two basic cases, and a third, more complex one. The pseudocode for both the CCW() and CW() variations of the algorithm are shown in Figure 2.

The algorithm takes two input parameters, which represent two binary tree nodes: the root of the tree to be processed, and its predecessor. The predecessor's initial value is always NULL, since the root of the input tree never has a predecessor node. However, the value of the predecessor parameter changes as further recursive calls to the algorithm are being invoked from inside the function itself. Moreover, the values of both the root and the predecessor nodes are guaranteed to change within subsequent recursive function calls, since the entire structure of the binary tree is rearranged after the roll operation executes fully.

The motivation for this paper was the fact that the binary tree roll algorithm, in either its CCW() or CW() variant, has so far been analyzed for time complexity [7] but not for space complexity. The latter is the goal of this paper and it will be done as follows, focusing on the CCW() variant. First, a theoretical analysis of the space complexity will be given, treating all cases of the algorithm execution. The principle of space com-

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

a)

```
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);
  }
 }
 }
}
```

b)

Figure 2. *The algorithms for a) CCW() and b) CW()[5]*

plexity analysis will be outlined, resulting in the worst- and best-case scenarios for the algorithm, stated in the form of presenting them as functions of the number of nodes in the tree *n*. Afterwards, it will be shown how those results are tested empirically, addressing the analytical results for the space complexities of the worst case, best case and average case of the algorithm. The paper will end with a conclusion about the material presented herein.

Space Complexity – Analytical Approach

The analysis will be done upon the CCW() version of the algorithm, i.e. it will concern Figure 2a. As stated in [1], the CW() algorithm is an inverse of CCW()— substituting "left" for "right" and vice versa, as well as CCW() for CW() (for the recursive calls) will transform the CCW() algorithm into the CW() algorithm, so the following analysis can thus be used for the CW() algorithm as well. The number of nodes of the tree will be denoted by *n*. The line numbers will refer to the algorithm in Figure 2.

The space complexity is concerned with the memory which gets occupied during the execution of the algorithm. The occupied memory consists of the memory of the system stack given to every recursive call and to auxiliary variables which are needed for certain cases of the algorithm.

Let *S(n)* denote the space complexity function of the binary tree roll. The trivial and the non-trivial cases of the CCW() algorithm will be analyzed for space complexity, since they all cause that a memory stack frame be used and placed on the call stack. In this analysis, the size of the stack frame reserved by the (recursive) function calls will be denoted by s ($s > 0$), whereas the size of the auxiliary variables introduced by a certain invocation case (more specifically, the third case of the algorithm, specified by lines 25-38 in Figure 2) will be denoted as a ($a > 0$).

The space complexity analysis cannot be performed using simple addition of function calls on the call stack, since, during the execution of the algorithm, some function calls get completed and leave the call stack, whereas others take their place. Therefore, it is the *depth* of *active function calls on the call stack* which is the measure of the space complexity [11], and a way to determine it needs to be devised. An example of such an approach is by drawing a *call stack tree* of the recursive function calls placed on the stack, which will be used in this paper. In the following analysis, such trees will be displayed for both the trivial and the non-trivial cases of the algorithm.

As will be shown in the following sections, the topology of the tree is a determining factor in the space complexity analysis of the algorithm. Since any topology of a tree with a given number of nodes *n* is equally likely to be passed to the roll algorithm, it is unfeasible to derive an equation S(*n)* as a function solely of the number of nodes *n*. However, it is possible to derive the worst- and best-case scenarios for the space complexity which are dependent on *n* and such approach will be shown for the space complexity analysis. To do so, the concept of a *terminal situation* will be introduced, which is a situation in which a case of the algorithm is invoked after which there are no more recursive calls, except to the trivial case only. This approach will be used to determine the space efficiency of the binary tree roll algorithm and thus express the space complexity as a function of the number of nodes in the binary tree.

The trivial case - line 3

The trivial case gets invoked every time an empty tree is given to the CCW() algorithm for processing, i.e., when the test in line 3 of Figure 1 yields false. This case simply generates a function call in the call stack, which momentarily occupies space s and then leaves the stack. It can be displayed as in Figure 3.

Since the CCW(0) case equals to a constant space complexity, its conversion to the constant s will not be displayed in subsequent figures and will be assumed to happen instantaneously. Therefore, the trivial case will be assumed to be a part of the terminal situations of the other, non-trivial, cases.

$$\begin{tikzcd} CCW(0) \arrow[d] \\ s \end{tikzcd}$$

Figure 3. *A call stack tree for the trivial case*

First case - lines 5-10

This case is invoked when the root has no right sub-tree (line 5 in Figure 2). The root's left sub-tree will be placed as its right sub-tree and a recursive call will be made upon the new right sub-tree. Figure 4 presents this case visually.

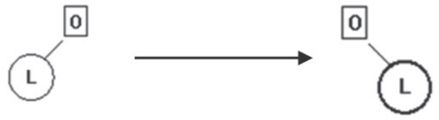


Figure 4. *The first basic case in the CCW() algorithm [1]*

In the first case, there are no auxiliary variables; just a function call to the sub-tree which is rolled from left to right (as shown in Figure 4). The sub-tree has one node less than the initial tree, so this can be represented as in Figure 5.

$$\begin{tikzcd} CCW(n_I) \arrow[dr] & \\ & CCW(n-1) \end{tikzcd}$$

Figure 5. *A call stack tree for the first case*

Note that the function call was denoted as $CCW(n_1)$. This is to stress that it is not only the amount of nodes in the tree that makes the difference, but also its *topology*. In other words, the first case will be invoked upon a tree with n nodes (which may also be a sub-tree of the original tree) only if its topology is such that the root of that tree does not have a right sub-tree. If the topology is such that the root has a right sub-tree having no right sub-tree of its own, the second case will be invoked, whereas if the root has a right sub-tree having a right sub-tree of its own, the third case will be invoked.

A terminal situation in the first case happens when CCW() is invoked on a (sub-)tree with just a single node (i.e., a leaf of the tree). This situation shows when the first case will no longer be invoked and it can be represented as in Figure 6.

$$\begin{tikzcd} CCW(1) \arrow[dr] & \\ & CCW(0) \end{tikzcd}$$

Figure 6. *A call stack tree for the terminal situation of the first case (a tree with just one node)*

<u>*Second case - lines 11-24*</u>

This case is activated when the root of the tree has a right sub-tree, which in turn does not have a right sub-tree of its own (line 13 in Figure 2). It is shown visually in Figure 7.

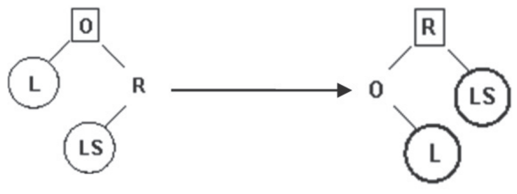


Figure 7. *The second basic case in the CCW() algorithm [1]*

This case of the algorithm also does not introduce auxiliary variables, whereas it produces two recursive function calls on sub-trees which have a total of nodes. In other words, two of the nodes get handled by the second case, whereas the remainder of the nodes is processed by the subsequent recursive calls. This can be represented as in Figure 8:

$$\begin{tikzcd} & CCW(n_{II}) \arrow[dl] \arrow[dr] & \\ CCW(t_1) & & CCW(t_2) \end{tikzcd}$$

Figure 8. *A call stack tree for the second case*

where $t_1 + t_2 = n - 2$.

A terminal situation in the second case happens when CCW() is invoked on a (sub-)tree containing two nodes, arranged as a topology of a root and its right child node. In such a situation, both of the nodes are handled by the second case and the two recursive calls are made on empty trees. This can be shown as in Figure 9.

$$\begin{tikzcd} & CCW(2_{II}) \arrow[dl] \arrow[dr] & \\ CCW(0) & & CCW(0) \end{tikzcd}$$

Figure 9. *A call stack tree for the terminal situation of the second case (a tree containing just a root and its right child node)*

<u>*Third case - lines 25-38*</u>

This case gets invoked when there is a stem of two or more right child nodes to the root (i.e., when the root has a right sub-tree, which has a right sub-tree of its own and so on; line 25 in Figure 2). As stated in [1], this case deals with the downshift of stems of right child nodes and transforming them into stems of left child nodes, according to the principle of CCW binary tree roll. The algorithm first creates a recursive call upon the right child node of the root and it continues to do so until a basic case is reached (i.e., until a sub-tree with at most one right child node is reached, following the stem of right child nodes from the root towards its rightmost child node). When such case is handled by the algorithm, the remainder of the third case relocates the former root of the tree to be the leftmost child node in the newly rolled tree, and the procedure is then recursively invoked again on the former root (and its entire left sub-tree), now placed as the leftmost node in the sub-tree handled by the third case. Figures 10 and 11 show the third case visually.

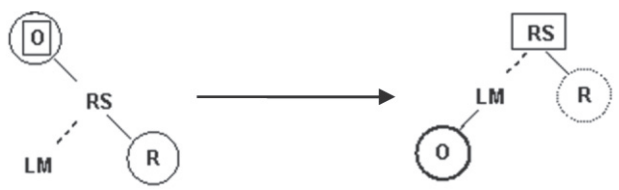


Figure 10. *The third and most complex case in the CCW() algorithm [1]*

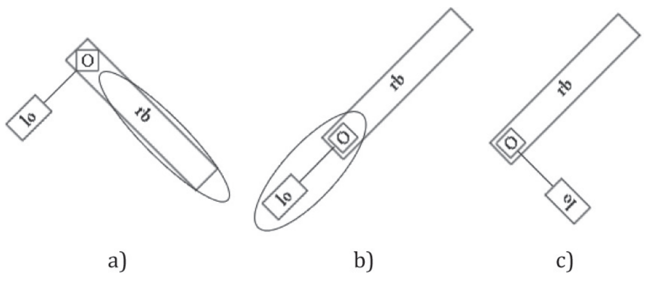


Figure 11. *The third case of the CCW() algorithm: a) the head recursion (ellipse) of the third case deals with the stem of right child nodes () and transforms it into a stem of left child nodes via downshift; b) the root () is linked as the leftmost in the stem of left child nodes and the tail recursion (ellipse) of the third case is invoked upon it; c) since the former root does not have a right child node of its own, the tail recursion will invoke the first case, and the left sub-tree of the former root () will become its right sub-tree*

Concerning the space complexity, the third case of the algorithm includes a head recursion on the nodes not containing the root and its left sub-tree, two auxiliary variables and a tail recursion on the root and its left sub-tree. This can be displayed as in Figure 12, from where it can be seen that the third case does not perform the actual roll, but only the downshift of the tree (since the amount of processed nodes does not decrease in the subsequent recursive calls).

$$\begin{tikzcd}[column sep=small] & CCW(n_{III}) \arrow[dl] \arrow[dr] & \\ CCW(n-1-lo) & & 2a+CCW(1+lo) \end{tikzcd}$$

Figure 12. *A call stack tree for the third case; is the number of nodes in the left sub-tree of the root*

A terminal situation for the third case is when CCW() is invoked on a (sub-)tree with three nodes which form a stem of right child nodes. In such a situation, the head recursion is invoked upon the bottom two nodes, which are handled by the second case, and the tail recursion is invoked upon the root, which is handled by the first case. This is graphically represented in Figure 13.

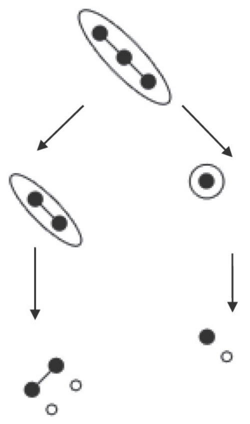


Figure 13. *A terminal situation for the third case. The head recursion is invoked on the bottom two nodes of the stem (left-hand side) which get CCW rolled and induce two recursive calls on empty sub-trees (small circles to the lower right of each node). The tail recursion gets invoked on the root of the stem, after the downshift process (right-hand side), which finishes with a recursive call on an empty sub-tree (small circle)*

The call stack tree of the terminal situation can therefore be shown as in Figure 14.

$$\begin{tikzcd}[column sep=small] & & CCW(3_{III}) \arrow[dl] \arrow[dr] & & \\ & CCW(2_{II}) \arrow[dl] \arrow[dr] & & 2a+CCW(1) \arrow[dr] & \\ CCW(0) & & CCW(0) & & CCW(0) \end{tikzcd}$$

Figure 14. *A call stack tree for the terminal situation of the third case*

<u>*The space complexity analysis: the space efficiency parameter*</u>

As initially assumed in the space complexity analysis, every recursive call (CCW (n_I), (CCW (n_{II}), (CCW (n_{III}), (CCW ($n-1$) etc.) occupies the same amount of memory space. The maximum depth of the call stack tree for a given roll operation therefore shows the maximum amount of memory space occupied by the recursive calls during the algorithm execution, i.e., represents the upper bound of the space complexity of the algorithm for a given tree with nodes. This can be used to determine the extreme scenarios, i.e., the worst- and best-case scenarios, for the space complexity of the CCW() algorithm.

To better quantify this, it is useful to introduce the *space efficiency* of the algorithm, as in Equation 3:

$$se_{case} = \frac{number\ of\ nodes\ processed\ per\ terminal\ situation\ of\ a\ given\ case}{amount\ of\ memory\ occupied\ by\ the\ terminal\ situation\ of\ the\ given\ case} \quad (3)$$

As can be inferred from Figures 9, 12 and 14, $se_I = \frac{1}{2s}$ (in the first case, processing 1 node takes 2 frames of recursive calls), $se_{II} = \frac{2}{2s} = \frac{1}{s}$ (in the second case, processing 2 nodes takes 2 frames of recursive calls), and $se_{III} = \frac{3}{3s+2a}$ (in the third case, processing 3 nodes takes 3 frames of recursive calls, plus the memory occupied by the auxiliary variables), respectively. Since every recursive call allocates space for two formal parameters (root and predecessor) of the same type as the auxiliary variables introduced in the third case of the algorithm (leftmost and newroot in lines 28 and 32 of Figure 1, respectively), it can be stated that the amount of space occupied by the auxiliary variables is, at best, equal to the space occupied by the recursive call, i.e., $2a \leq s$. This yields that $\frac{1}{s} = \frac{3}{3s} > se_{III} = \frac{3}{3s+2a} \geq \frac{3}{4s}$, and, since $\frac{1}{s} > \frac{3}{4s} > \frac{1}{2s}$, it follows that $se_{II} > se_{III} > se_I$. In other words, as far as space complexity of the algorithm is concerned, the most efficient is the second case and the least efficient is the first case. Therefore, the best-case scenario happens when a tree is processed solely by the second case of the algorithm and the worst-case scenario happens when a tree is processed solely by the first case of the algorithm. Figure 15 shows trees with $n = 6$ which will exhibit worst-case and best-case space complexities when rolled by the CCW() algorithm.

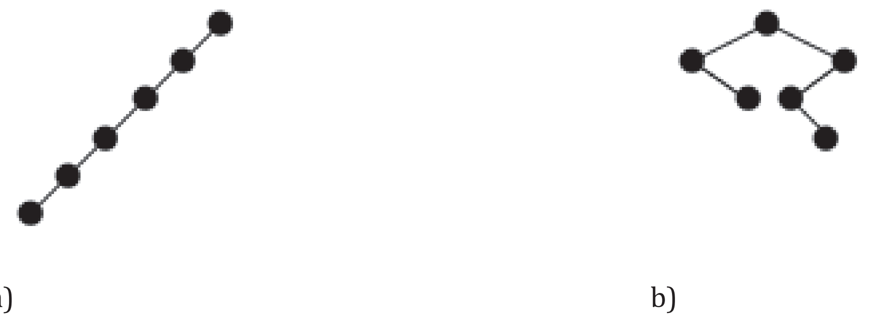


Figure 15. *Trees with which, when rolled with the CCW() algorithm, will exhibit a) worst-case time complexity; and b) best-case time complexity*

In the worst case, there will be as many recursive calls as there are nodes in the tree, plus one for the trivial case, which means that the equation for the worst-case scenario is represented by Equation 4:

$$S_O(n) = n + 1 \quad (4)$$

It is straightforward to notice that this equation has a linear time complexity, which means that the worst-case scenario for the space complexity of the algorithm is linear, or, as stated in Equation 5:

$$S(n) = O(n) \quad (5)$$

To analyze the best case, it is important to understand the regularity by which a tree is formed, which will be processed solely by the second case of the algorithm. The smallest such tree has $n = 2$ and consists of a root and its right child node. The next "best-case tree" will be formed by connecting two such sub-trees, consisting of a root and its right child node, as left child nodes of both the original root and the right child node of the root, which yields a tree with $n = 6$ (as shown in Figure 15b). The next such "best-case tree" will be formed by connecting sub-trees consisting of a root and its right child node to all nodes not having left child nodes, which are four in the "best-case tree" for $n = 6$, which will result in a new "best-case tree" with $n = 14$ (as shown in Figure 16). In such tree, there are eight nodes not having left child nodes, so connecting sub-trees consisting of a root and its right child node to all eight nodes will yield a new "best-case tree" with $n = 30$ and so on.

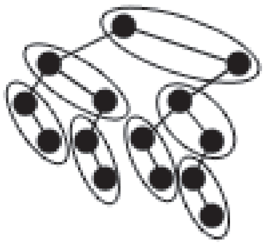


Figure 16. *Tree with which, when rolled with the CCW() algorithm, will exhibit the best-case time complexity. It is obtained by linking every node, which does not have a left sub-node, from the previous such tree (shown in Figure 15b), with a sub-tree consisting of a root and its right sub-node. Such sub-trees are indicated by ellipses around them*

From the following, it can be concluded that the initial "best-case tree" has $n_0 = 2$ nodes, and each successive "best-case tree" has $n = \sum_{i=1}^{j} 2^i = 2^{j+2} - 2$ nodes, where j is the length of the stem of left sub-nodes starting from the root (or from its right child node), which is the longest such stem (note that this stem does *not* include the root). In that case, $j + 2$ is the number of stack frames used for the recursive calls: j for the calls upon sub-trees with roots in the stem (there are j such nodes in the stem, excluding the root), 1 for the call on the initial root of the tree and 1 for the final call, on an empty tree from the leftmost node in the stem. Thus, the equation for it represents the relation for the best-case space complexity, expressed in terms of n, and it can be stated as in Equation 6:

$$S_\Omega(n) = s \cdot \lg(n + 2) \tag{6}$$

where s is the amount of memory used by every invocation of the recursive function in the system stack. Equation 6 indicates that the best-case space complexity is tightly logarithmic. This can be stated as in Equation 7:

$$S_\Omega(n) = \Theta(\lg n) \tag{7}$$

An equivalent statement is that the space complexity of the CCW() algorithm is logarithmic in the best case, as stated in Equation 8:

$$S(n) = \Omega(\lg n) \tag{8}$$

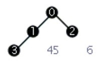
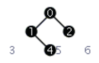
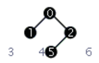
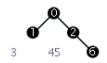
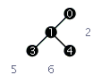
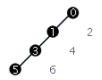
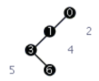
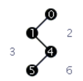
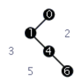
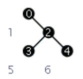
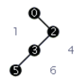
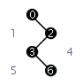
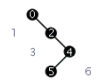
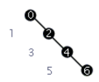
It is thus shown that the space complexity of the binary tree roll algorithm has a linear worst-case space complexity (Equation 5) and a logarithmic best-case space complexity (Equation 8).

Space Complexity – Empirical Approach

In order to be certain about how much space is needed to perform CCW roll on a tree with nodes, an exhaustive analysis needs to be performed. This includes obtaining all topologies of binary trees with nodes and performing CCW roll on them, while obtaining the stack depth for each tree while it is being CCW rolled. For this, it is necessary to first generate all topologies of binary trees for a given and then execute CCW roll on all of them, while measuring the stack depths during the executions of the CCW roll. The smallest value of the stack depth while CCW rolling a tree with nodes will represent the best case for that , whereas the largest value of such a stack depth will represent the worst case for that . As following from the theoretical analysis, the best cases for increasing values of are expected to grow logarithmically, and the worst cases are expected to grow linearly. It is also appealing to know whether the space complexity of the algorithm would be more dominantly logarithmic or linear, which is why an average time complexity would also need to be extracted, as an average of the space complexities for all topologies of binary trees for a given number of nodes.

In order to obtain all topologies of binary trees with a given number of nodes, the Catalan Cipher Vector approach is used in this paper. A Catalan Cipher Vector [6] is a vector which uniquely determines a binary tree's topology. For a tree with nodes, there will be (n-th Catalan number) topologies of binary trees and thus Catalan Cipher Vectors. Table 1 shows all the ranks, their corresponding Catalan Cipher Vectors, and the appropriate binary trees, for n = 4 nodes.

Table 1. *Ranks and enumerations of the binary trees with nodes using the Catalan Cipher Vector approach*

Rank	Catalan Cipher Vector	Binary Tree
0	[0 1 2 3]	
1	[0 1 2 4]	
2	[0 1 2 5]	
3	[0 1 2 6]	
4	[0 1 3 4]	
5	[0 1 3 5]	
6	[0 1 3 6]	
7	[0 1 4 5]	
8	[0 1 4 6]	
9	[0 2 3 4]	
10	[0 2 3 5]	
11	[0 2 3 6]	
12	[0 2 4 5]	
13	[0 2 4 6]	

Since the initial Catalan Cipher Vector for a tree with nodes is always [6], it is possible to generate the corresponding binary tree for it, and determine the stack depth occupied during the execution of CCW() on it. Then, the subsequent Catalan Cipher Vector can be obtained, the corresponding binary tree can be generated from it, have CCW() executed on it and determine the stack depth needed and so on, until all binary tree topologies get processed this way. The obtained minimum depth represents the best case, the maximum depth represents the worst case, whereas the average case is calculated as the sum of all depths divided by the number of possible trees with nodes.

The results for such an analysis have been performed and the results are given in Table 2.

Table 2. *Stack depths necessary to perform CCW() on all topologies of binary trees with given numbers of nodes*

n	*C*(*n*)	*Min*	*Max*	*Avg*	*Total*
2	2	2	3	3	5
3	5	3	4	3	16
4	14	3	5	4	54
5	42	3	6	4	187
6	132	3	7	5	664
7	429	4	8	6	2.393
8	1.430	4	9	6	8.719
9	4.862	4	10	7	32.073
10	16.796	4	11	7	118.848
11	58.786	4	12	8	443.081
12	208.012	4	13	8	1.660.503
13	742.900	4	14	8	6.250.670
14	2.674.440	4	15	9	23.620.379
15	9.694.845	5	16	9	89.560.477
16	35.357.670	5	17	10	340.599.877
17	129.644.790	5	18	10	1.298.763.168
18	477.638.700	5	19	10	4.964.255.082

The results are interpreted as follows. In the first data row, for a tree with = 2 nodes (first column), there are = 2 (second column) total topologies of binary trees. Executing CCW() on all of them yields a (sixth, i.e. last column) of 5 levels of stack depth, leading to an (average – fifth column) of 3 levels of stack depth per binary tree topology. Of all topologies, the (minimum – third column) levels of stack depth necessary to complete CCW() on a binary tree topology with 2 nodes is 2, and the (maximum – fourth column) number of such levels is 3. This interpretation follows all rows of the table, up to and including binary tree topologies for = 18 nodes.

Plotting the results obtained from Table 3 results in a chart like in Figure 17.

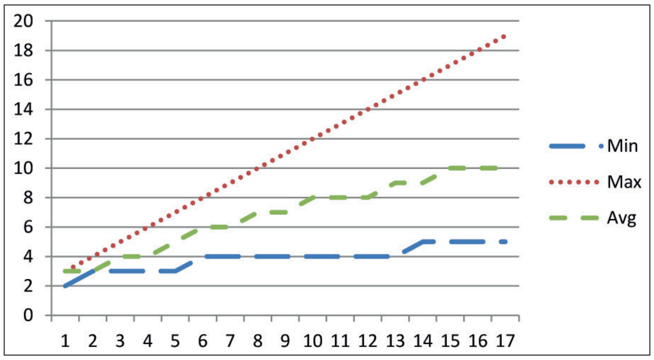


Figure 17. *A plot of the results given in Table 3*

Interpolating the equations for the worst and best cases yields the equations and respectively, confirming the conclusions obtained in the analysis of the space complexity that the worst-case space complexity of the CCW() algorithm is linear, and its best-case space complexity is logarithmic. Interpolating for the average case results in the equation , meaning that the average space complexity of the CCW() roll algorithm is dominantly logarithmic. Therefore, it can be stated that, on average, the space complexity of the CCW() algorithm is logarithmic.

CONCLUSION

This paper presented an analysis of the space complexity of the binary tree roll algorithm, specifically its counterclockwise (CCW()) variant, with the note that the analysis for its clockwise (CW()) variant is analogous. For it, the theoretical analysis was derived in such way as to show the amount of memory occupied during the execution of the algorithm, shown primarily through the number of frames occupied in the call stack by the recursive function calls. The analysis of the space efficiency of each of the three cases of the algorithm, as well as the trivial case, showed that trees which are processed solely by the first case of the algorithm (invoked upon a node which has no right sub-tree) yield the worst space complexity, whereas trees which are processed solely by the second case of the algorithm (invoked upon a node which has a right sub-tree, which in turn has no right sub-tree of its own) yield the best space complexity. The equations were derived to be linear for the worst case and logarithmic for the best case. An exhaustive empirical analysis was performed, counting the minimum and maximum stack depths for all trees with given numbers of nodes . The equations in the theoretical analysis were confirmed by the empirical results, that the CCW() algorithm has linear space complexity for the worst case and logarithmic space complexity for the best case. Moreover, the average case analysis showed that the CCW() algorithm has a dominantly logarithmic space complexity, so it can be stated that its average space complexity is logarithmic.

REFERENCES

[1] Adel'son-Vel'skii G. M. and Landis E. M. (1962) "An algorithm for the organization of information," *Soviet Math. Doklady*, 3: 1259-1263

[2] Amani M., Lai K. A. and Tarjan R. E. (2016) "Amortized rotation cost in AVL trees," *Information Processing Letters*, 98(5): 327-330

[3] Bayer R. and McCreight E. (1972) "Organization and Maintenance or Large Oriented Indexes," *Acta Informatica*, 3(3): 173-189

[4] Bozinovski A. and Bozinovski S. (2004) N-queens Pattern Generation: An Insight Into Space Complexity of a Backtracking Algorithm, Proceedings of the Third International Symposium on Information and Communication Technologies, Las Vegas, NV, USA (pp. 281-286). ACM

[5] Božinovski, A. and Ackovska, N. (2012) The Binary Tree Roll Operation: Definition, Explanation and Algorithm, International Journal of Computer Applications, 46(8):40-47

[6] Božinovski, A., Stojčevska, B. and Pačovski, V. (2013) Enumeration, Ranking and Generation of Binary Trees Based on Level-Order Traversal Using Catalan Cipher Vectors, Journal of Information Technology and Applications, 3(2):78-86

[7] Božinovski, A., Tanev, G., Stojčevska, B., Pačovski, V. and Ackovska, N. (2016) Time Complexity Analysis of the Binary Tree Roll Algorithm, International Journal of Computer Applications, 6(2):53-62

[8] Brassard G. and Bratley P. (2002) *Fundamentals of Algorithmics*. Prentice Hall of India

[9] Buchmann J., Dahmen E., Klintsevich E., Okeya K. and Vuillaume C. (2007) "Merkle signatures with virtually unlimited signature capacity," In *Proc. 5th Int. Conf. Applied Cryptography and Network Security*, Zhuhai, China (pp. 31-45). LCNS Springer

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

[11] Goodrich, M. T. and Tamassia, R. (2008) *Data Structures and Algorithms in Java, Third Edition*, Wiley India Pvt.

[12] Katz J. (2003) "Binary Tree Encryption: Constructions and Applications," In *Information Security and Cryptology (ICISC 2003)*, 2971: 1-11, LCNS Springer

[13] Kreher D. L. and Stinson D. R. (1998) *Combinatorial Algorithms: Generation, Enumeration, and Search*. Discrete Mathematics and its Applications (Book 7), CRC Press, 1st Edition

[14] Lee Y. and Lee J. (2015) "Binary tree optimization using genetic algorithm for multiclass support vector machine," *Expert Systems with Applications*, 42(8): 3843-3851

[15] Li Y., Xu M., Zhao H. and Huang W. (2016) "Hierarchical fuzzy entropy and improved support vector machine based binary tree approach for rolling bearing fault diagnosis," *Mechanism and Machine Theory*, 98: 114-132

[16] Liu B., Shen Y., Chen X., Chen Y. and Wang X. (2014) "A partial binary tree DEA-DA cyclic classification model for decision makers in complex multi-attribute large-group interval-valued intuitionistic fuzzy decision-making problems," *Information Fusion*, 18: 119-130

[17] Rich E. (1983) *Artificial Intelligence*. McGraw-Hill series in artificial intellilgence, McGraw-Hill Inc.

[18] Roch S. and Steel M. (2014) "Likelihood-based tree reconstruction on a concatenation of alignments can be statistically inconsistent, " *Theoretical Population Biology,* 100: 56-62

[19] Sedgewick R. (1998) *Algorithms in C++, Parts 1-4: Fundamentals, Data Structures, Sorting, Searching, 3rd ed,* Addison-Wesley

[20] Suh I. and Headrick T. C. (2010) "A comparative analysis of the bootstrap versus traditional statistical procedures applied to digital analysis based on Benford's Law," *Journal of Forensic and Investigative Accounting*, 2(2): 144-175.

Submitted: December 1, 2016.
Accepted: December 5, 2016.

About the Authors

Adrijan Božinovski works as an Associate Professor at the School of Computer Science and Information Technology at University American College Skopje, where he is currently the Dean. He obtained his BSc from University "St. Cyril and Methodius" in Skopje, Macedonia, and his MSc and PhD from University of Zagreb, Croatia.

Biljana Stojčevska works as an Associate Professor at the UACS School of Computer Science and Information Technology. She received her BSc, MSc and PhD degrees in Computer Science at the Institute of Informatics, Faculty of Natural Sciences and Mathematics, at "Sts. Cyril and Methodius University" in Skopje, Macedonia.

Veno Pachovski (1965) graduated, completed MSc and got his PhD from Faculty of Natural Sciences and Mathematics, University "Sts. Cyril And Methodius", Skopje, Macedonia. Since 2009, he teaches a variety of courses at the University American College – Skopje, mainly within the School of Computer Sciences and Information technology (SCSIT).

George Tanev is an MSc graduate student of the School of Computer Science and Information Technology at University American College Skopje, Macedonia, where he acquired his BSc in Computer Science. Also works as a software developer in Skopje, Macedonia.

Nevena Ackovska is Associate Professor at the Faculty of Computer Science and Engineering at "St. Cyril and Methodius" University in Skopje, Macedonia. She holds B.Sc. in Computer Engineering, Informatics and Automation from Electrical Engineering Faculty (2000), M.Sc. in Bioinformatics (2003) and a Ph.D. in Bioinformatics (2008) from Faculty of Natural Sciences and Mathematics at "St. Cyril and Methodius University" in Skopje, Macedonia.