Enumeration, Ranking and Generation of Binary Trees Based on Level-Order Traversal Using Catalan Cipher Vectors

Adrijan Božinovski[1], **Biljana Stojčevska**[2], **Veno Pačovski**[3]

[1]bozinovski@uacs.edu.mk, [2]stojcevska@uacs.edu.mk, [3]pachovski@uacs.edu.mk

Critical review

DOI: **10.7251/JIT1302078B** UDC: **579.253:577.2**

Abstract: In this paper, a new representation of a binary tree is introduced, called the Catalan Cipher Vector, which is a vector of n elements with certain properties. It can be ranked using a special form of the Catalan Triangle designed for this purpose. It is shown that the vector coincides with the level-order traversal of the binary tree and how it can be used to generate a binary tree from it. Streamlined algorithms for directly obtaining the rank from a binary tree and vice versa, using the Catalan Cipher Vector during the processes, are given. The algorithms are analyzed for time and space complexity and shown to be linear for both.

The Catalan Cipher Vector enables a straightforward determination of the position and linking for every node of the binary tree, since it contains information for both every node's ancestor and the direction of linking from the ancestor to that node. Thus, it is especially well suited for binary tree generation. Using another structure, called a canonical state-space tableau, the relationship between the Catalan Cipher Vector and the level-order traversal of the binary tree is explained.

Keywords: Enumeration, Rank, Generation, Binary tree, Level-order traversal, Catalan Cipher Vector, Canonical State-Space Tableau

Introduction

Enumeration of binary trees means that every binary tree is linked to a unique linear representation, usually in a form of a sequence of integers or characters. Since there are C_n different binary trees of n nodes, with C_n being the n -th Catalan number, there should be C_n different representations to uniquely identify the trees. Every representation can be given a rank, usually a single integer, which establishes a relation of strict order between the binary trees. The conversion from a representation to a rank is usually done by using a Catalan Triangle, in a form that best suits the given representation.

The research in enumeration of binary trees has produced results including enumeration using bit strings [4, 8, 13] and integer sequences [5, 10, 11, 12]. Enumeration using integer sequences has been further subdivided into enumeration by Codewords [9, 14], weights [7] and distance [6]. More recently, enumeration using Catalan combinations [3] has been introduced.

The ranking of a binary tree is done by obtaining a unique value from the binary tree or its enumeration, which gives it a certain rank (i.e. number in a sequence) among other binary trees of a given size. The generation of a binary tree from its enumeration or rank represents the actual formation of the binary tree from its representation, whether it is the enumeration or the rank.

This paper introduces a new way of enumeration of a binary tree, called a Catalan Cipher Vector, and a way to transform that enumeration into previous forms and vice versa. In particular, it will be shown how this enumeration relates to the level-order traversal of the binary tree. Streamlined algorithms will be presented, by which the rank of a binary tree is obtained from the generated binary tree and vice versa, during which the corresponding Catalan Cipher Vector elements will be obtained and directly utilized. The algorithms will be analyzed for time and space complexity.

Introducing the Catalan Cipher Vector

A *Catalan Cipher Vector* is the vector $v = [v_0\ v_1\ v_2\ \dots\ v_{n-1}]$ which satisfies the following properties:

1) $v_0 = 0$;

2) $v_{i-1} + 1 \le v_i \le 2i$, for $i = 1,2,3, \dots , n - 1$ and $v_i \in \mathbb{N}$.

Listing all distinct Catalan Cipher Vectors with lengths n shows that there are C_n such vectors. In Table 1, all Catalan Cipher Vectors are listed for $n = 4$, alongside the corresponding Codewords and Catalan combinations (the index of the first element in every representation is 0, and that element's value is also always 0). The relationships among the representations are also given at the bottom of the table. Unlike in the Codewords and the Catalan combinations, the elements in the Catalan Cipher Vectors are never equal to one another.

The Canonical State-Space Tableau and its Connection to the Catalan Cipher Vector

The usefulness of the Catalan Cipher Vector can be demonstrated by using a special structure, a state-space tableau. In it, the binary tree is represented by using a tableau with dimensions $n \times 3$, where the first

Table 1. List of Codewords, Catalan combinations and Catalan Cipher Vectors for n = 4

Rank	Codeword $\boldsymbol{d}$	Catalan combination $\boldsymbol{c}$	Catalan Cipher Vector $\boldsymbol{v}$
0	[0 0 0 0]	[0 0 0 0]	[0 1 2 3]
1	[0 0 0 1]	[0 0 0 1]	[0 1 2 4]
2	[0 0 0 2]	[0 0 0 2]	[0 1 2 5]
3	[0 0 0 3]	[0 0 0 3]	[0 1 2 6]
4	[0 0 1 0]	[0 0 1 1]	[0 1 3 4]
5	[0 0 1 1]	[0 0 1 2]	[0 1 3 5]
6	[0 0 1 2]	[0 0 1 3]	[0 1 3 6]
7	[0 0 2 0]	[0 0 2 2]	[0 1 4 5]
8	[0 0 2 1]	[0 0 2 3]	[0 1 4 6]
9	[0 1 0 0]	[0 1 1 1]	[0 2 3 4]
10	[0 1 0 1]	[0 1 1 2]	[0 2 3 5]
11	[0 1 0 2]	[0 1 1 3]	[0 2 3 6]
12	[0 1 1 0]	[0 1 2 2]	[0 2 4 5]
13	[0 1 1 1]	[0 1 2 3]	[0 2 4 6]
	$d_0 = 0$; $d_i = c_i - c_{i-1} = v_i - v_{i-1} - 1$: $1 \le i \le (n-1), i \in \mathbb{N}$	$c_o = 0$; $c_i = \sum_{j=0}^{i} d_j = v_i - i$: $1 \le i \le (n-1), i \in \mathbb{N}$	$v_0 = 0$; $v_i = \sum_{j=0}^{i} d_j + i = c_i + i$: $1 \le i \le (n-1), i \in \mathbb{N}$

(left-hand) column contains the symbols representing the values of the nodes of the tree, the second (middle) one contains the symbols rep, resenting the values of the nodes that the elements in the first column of the corresponding rows have as their left sub-nodes, and the third (right-hand) one contains the symbols representing the values of the nodes that the elements in the first column of the corresponding rows have as their right sub-nodes. In other words, the first column contains a node of the binary tree, the second column of that row contains that node's left sub-node (if the field is non-empty) and the third column of that row contains that node's right sub-node (if the field is non-empty). The root of the tree is the node of which the value is not found in the second or third column of the tableau (since the root does not have an ancestor node). Because a tree has at least one leaf, at least one of the rows in the tableau will have no values in the second and third column (since a leaf does not have any other nodes as sub-nodes). This is because a tree with n nodes has $n - 1$ edges: in a tableau of $3n$ fields, n are occupied in the first column and $n - 1$ in the second and third column, which leaves $n + 1$ empty fields; of those, at least two will be in the same row.

In the state-space tableau, it is irrelevant whether the rows are shuffled, since the structure of the binary tree is uniquely determined by the position relative to the columns, especially the second and third column, and not the rows. The row that has the elements in the second and third column as empty will be a leaf, and the element that is not found in the second and third column, but is present in the first one, will be the root. This is shown in Figure 1, where in all tableaux ***A*** is the root (it is present in the first, column, but not in the second and third column), while ***C*** and ***D*** are leaves (they have blank fields in the second and third column of their respective rows).

Since all state-space tableaux for a given binary tree are equivalent, it is necessary to choose one of them, to be worked with. The best choice is to select the tableau where the first row is the one that represents the root, and all other nodes are ordered in such a way that their vertical sequence in the first column follows the horizontal sequence in the second and third column. Such a state-space tableau is called the *canonical state-space tableau*. For example, Figure 1b is the canonical state-space tableau for the binary tree in Figure 1a.

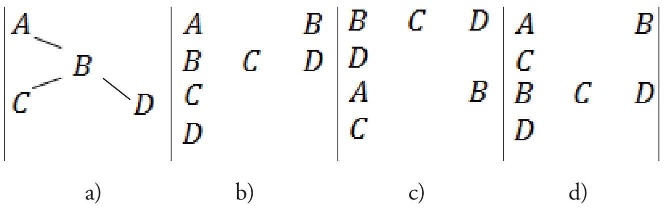
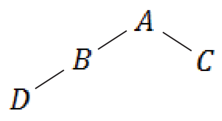
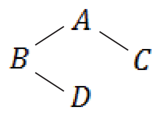
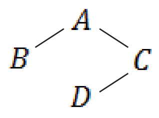
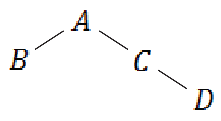
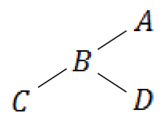
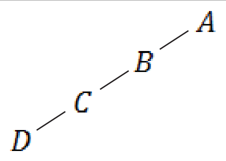
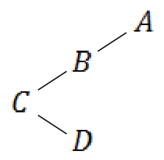
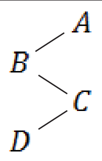
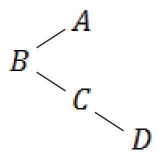
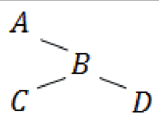
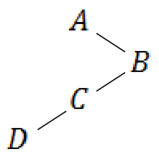
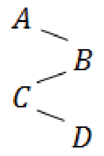
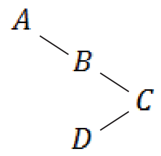
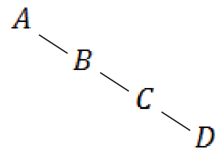


Figure 1. a) A binary tree; b) Its canonical state-space tableau; c), d) Other equivalent state-space tableaux

Table 2 contains all binary trees for $n = 4$ and their corresponding canonical state-space tableaux and Catalan Cipher Vectors. For clarity, the elements in the canonical state-space tableaux are indexed, both with subscript and superscript indices. The subscripted indices, in the first column, represent the indices of the values stored in the nodes of the tree, obtained by following some traversal of the binary tree. The superscripted indices, in the second and third column, are sequential, starting with 1 at the top row in the second column and moving to the right and down, and enumerating only the elements in those two columns. If the field with a given index in the second or third column is non-empty, it represents the value stored in the node, which the node in the given row has as a left sub-node (if the field is in the second column, i.e. has an odd index) or right sub-node (if the field is in the third column, i.e. has an even index). An index of 0 denotes the root, and $v_0 = 0$ for every Catalan Cipher Vector, since the root does not have an ancestor. The elements of the Catalan Cipher Vector are also indexed with subscripted indices, to demonstrate the connection with the indices of the corresponding elements of the first column of the canonical state-space tableau.

Table 2. All binary trees with their corresponding canonical state-space tableaux and Catalan Cipher Vectors for n = 4

Rank	Binary tree	Canonical state-space tableau	Catalan Cipher Vector $\boldsymbol{v}$
0		$\begin{array}{lll} A_0 & B^1 & C^2 \\ B_1 & D^3 & {}^4 \\ C_2 & {}^5 & {}^6 \\ D_3 & & \end{array}$	$[0_0 \quad 1_1 \quad 2_2 \quad 3_3]$
1		$\begin{array}{lll} A_0 & B^1 & C^2 \\ B_1 & {}^3 & D^4 \\ C_2 & {}^5 & {}^6 \\ D_3 & & \end{array}$	$[0_0 \quad 1_1 \quad 2_2 \quad 4_3]$
2		$\begin{array}{lll} A_0 & B^1 & C^2 \\ B_1 & {}^3 & {}^4 \\ C_2 & D^5 & {}^6 \\ D_3 & & \end{array}$	$[0_0 \quad 1_1 \quad 2_2 \quad 5_3]$
3		$\begin{array}{lll} A_0 & B^1 & C^2 \\ B_1 & {}^3 & {}^4 \\ C_2 & {}^5 & D^6 \\ D_3 & & \end{array}$	$[0_0 \quad 1_1 \quad 2_2 \quad 6_3]$
4		$\begin{array}{lll} A_0 & B^1 & {}^2 \\ B_1 & C^3 & D^4 \\ C_2 & {}^5 & {}^6 \\ D_3 & & \end{array}$	$[0_0 \quad 1_1 \quad 3_2 \quad 4_3]$
5		$\begin{array}{lll} A_0 & B^1 & {}^2 \\ B_1 & C^3 & {}^4 \\ C_2 & D^5 & {}^6 \\ D_3 & & \end{array}$	$[0_0 \quad 1_1 \quad 3_2 \quad 5_3]$
6		$\begin{array}{lll} A_0 & B^1 & {}^2 \\ B_1 & C^3 & {}^4 \\ C_2 & {}^5 & D^6 \\ D_3 & & \end{array}$	$[0_0 \quad 1_1 \quad 3_2 \quad 6_3]$
7		$\begin{array}{lll} A_0 & B^1 & {}^2 \\ B_1 & {}^3 & C^4 \\ C_2 & D^5 & {}^6 \\ D_3 & & \end{array}$	$[0_0 \quad 1_1 \quad 4_2 \quad 5_3]$
8		$\begin{array}{lll} A_0 & B^1 & {}^2 \\ B_1 & {}^3 & C^4 \\ C_2 & {}^5 & D^6 \\ D_3 & & \end{array}$	$[0_0 \quad 1_1 \quad 4_2 \quad 6_3]$
9		$\begin{array}{lll} A_0 & {}^1 & B^2 \\ B_1 & C^3 & D^4 \\ C_2 & {}^5 & {}^6 \\ D_3 & & \end{array}$	$[0_0 \quad 2_1 \quad 3_2 \quad 4_3]$

10		$\begin{array}{lll} A_0 & 1 & B^2 \\ B_1 & C^3 & 4 \\ C_2 & D^5 & 6 \\ D_3 & & \end{array}$	$[0_0\ \ 2_1\ \ 3_2\ \ 5_3]$
11		$\begin{array}{lll} A_0 & 1 & B^2 \\ B_1 & C^3 & 4 \\ C_2 & 5 & D^6 \\ D_3 & & \end{array}$	$[0_0\ \ 2_1\ \ 3_2\ \ 6_3]$
12		$\begin{array}{lll} A_0 & 1 & B^2 \\ B_1 & 3 & C^4 \\ C_2 & D^5 & 6 \\ D_3 & & \end{array}$	$[0_0\ \ 2_1\ \ 4_2\ \ 5_3]$
13		$\begin{array}{lll} A_0 & 1 & B^2 \\ B_1 & 3 & C^4 \\ C_2 & 5 & D^6 \\ D_3 & & \end{array}$	$[0_0\ \ 2_1\ \ 4_2\ \ 6_3]$

The benefit from the Catalan Cipher Vector is that it directly determines the topology of the binary tree, i.e. the connections between the nodes. The element with index 0 (i.e. the root) can have elements connected to it with values, in the Catalan Cipher Vector, of 1 and 2 only, which corresponds to its left and right sub-nodes, respectively, in the corresponding canonical state-space tableau. The element with index 1 can have elements connected to it with values, in the Catalan Cipher Vector, of 3 and 4 only etc. In other words, the element with index $\boldsymbol{i}$ can have connections to elements with values, in the Catalan Cipher Vector, of $2i + 1$ and $2i + 2$ only. This means that the index of the ancestor node for the node with index i (for $i \geq 1$) is directly obtainable from the corresponding value v_i of the Catalan Cipher Vector as $\left\lfloor \frac{v_i}{2} \right\rfloor$, if v_i is odd, or $\frac{v_i}{2} - 1$, if v_i is even. Furthermore, the direction of linking from the ancestor to the current node is also directly obtainable, since, following the canonical state-space tableau, the current node will be its ancestor's left sub-node if v_i is odd, or right sub-node if v_i is even.

As an example from Table 2, the tree with rank 12 has the Catalan Cipher Vector of [0 2 4 5]. Viewing the node with index 1 (in this example, with information B), the value of the corresponding element from the Catalan Cipher Vector is $v_1 = 2$. Since v_1 is even, the node is its ancestor's right sub-node and its ancestor is the node with the index $\frac{v_2}{2} - 1 = 0$ (i.e. the root, with information A). On the other hand, the value of the Catalan Cipher Vector for the node with index 3 (in this example, with information D) is odd ($v_3 = 5$), so the node is its ancestor's left sub-node and its ancestor is the node with the index $\left\lfloor \frac{v_3}{2} \right\rfloor = 2$ (i.e. the node with information C).

The Catalan Triangle

To establish a connection between the representation of the binary tree and its rank, several researchers [7, 11, 13] have utilized various forms of the Catalan Triangle. The form introduced in this paper is similar to the one used by [3], and is a transposed version of it. Figure 3 shows the Catalan Triangle that will be used in this paper, for $n = 4$. For clarity, the Catalan Triangle will be referred to as CT, and its elements will be given by their indices. For example, in Figure 2, $CT_{2,1} = 3$.

$i\downarrow$ $j\rightarrow$	0	1	2	3
0	14			
1	9	5		
2	4	3	2	
3	1	1	1	1

Figure 2. A Catalan Triangle CT for n = 4

A closer examination of the Catalan Triangle in Figure 2 reveals that its elements can be obtained iteratively. Figure 3 presents the relationships among the elements of the Catalan Triangle in Figure 2.

i↓ j→	0	1	2	3
0	$CT_{1,0} + CT_{1,1}$			
1	$CT_{2,0} + CT_{1,1}$	$CT_{2,1} + CT_{2,2}$		
2	$CT_{3,0} + CT_{2,1}$	$CT_{3,1} + CT_{2,2}$	$CT_{3,2} + CT_{3,3}$	
3	1	1	1	1

Figure 3. Relationships among the elements of the Catalan Triangle for n = 4

In this Catalan Triangle, and any other for $n \geq 1$, all elements in the bottom row are 1, every element on the diagonal is the sum of the element below it and the element below and right to it, and every other element is a sum of the element below it and the element to the right of it. Therefore, the elements of the Catalan Triangle can be obtained as

$$CT_{i,j} = \begin{cases} 1 & i = n-1 \\ CT_{i+1,j} + CT_{i+1,j+1} & i = j \\ CT_{i+1,j} + CT_{i,j+1} & \text{any other case} \end{cases}$$

An algorithm which initializes the Catalan Triangle using the aforementioned formula, for a given n, would be optimized for space and time, in a sense that the memory occupied would be only as much as needed, and that every element would be updated only once. Nevertheless, this means that there would be $n(n+1)/2$ memory units utilized and as many time units for updating them, which would make such an algorithm $\Theta(n^2)$ for space and time complexity. However, once the Catalan Triangle would be initialized, all other algorithms would utilize the information stored in it and their performances would be enhanced.

Algorithms for Converting between the Rank and the Binary Tree

In this paper, the algorithms will be presented in pseudocode that resembles the C++ programming language, where the keywords will be displayed in italic type. The keyword *ref* means that the value is passed by reference, i.e. that it will be modified within the algorithm and that its modified version will be available after the algorithm ends. The keyword *define* is used to define auxiliary variables or references within the algorithm. The keyword *new* means that memory will be assigned to the reference it affects. The keyword *array* means that the memory assigned to the reference will be an array with a specified size. The keyword *node* means that the memory assigned to the reference will be of a type of a binary tree node. The keyword *isEmpty* is an algorithm that returns true if a queue is empty and false otherwise. The keywords *enqueue* and *dequeue* are algorithms for enqueuing into or dequeuing from a queue, respectively. The keywords *leftSubNode* and *rightSubNode* refer to a left sub-node and a right sub-node of a given node, respectively. Other keywords, as well as special characters, retain their corresponding meanings and functions from the C++ language.

An efficient algorithm for obtaining a Catalan combination from a given rank is given in [3]. It can be slightly modified to produce a corresponding Catalan Cipher Vector as a result, because of the interchangeability between a Catalan combination and a Catalan Cipher Vector (Table 1). Since each element of the Catalan Cipher Vector determines the predecessor of the corresponding node in the binary tree, as well as whether it is its predecessor's left or right sub-node, the binary tree can be generated immediately after obtaining each of its elements. Since the root does not have a predecessor, it can be generated directly, without linking it to any other node.

Algorithm rank2tree (Figure 4a) shows how to obtain the binary tree for a given rank. First the Catalan Cipher Vector element is obtained and then a new node of the binary tree is generated, for which the index of its predecessor and the direction of linking from it is calculated, based on the value of the element of the vector. If the current node is the root, no linking takes place; else, the current node is linked to the predecessor node. All nodes are generated in an array, and only the first node of the array is returned, which is the root of the tree.

Algorithm val (Figure 4b) is an auxiliary algorithm that generates values of the information fields for the nodes of the trees based on their indices, which in this case is set to return the index itself as the information field of the node. Arbitrary logic can be used if it needs to return different information fields based on the indices (for example, the trees in Figures 1 and 2 and Table 1 have the letters of the alphabet stored in the information fields of the nodes).

```
rank2tree(rank, CT, n){
    define i, base, v, tn;
    i = 0;    base = 0;
    v = new array(n);
    tn = new array(n);
    tn[0] = new node(val(0));
    while(i < n){
        while((base < n) && (CT[i][base] <= rank)){
            rank = rank - CT[i][base];
            base++;
        }
        v[i] = base+i;
        tn[i] = new node(val(i));
        if(i != 0)
            if(v[i]%2 != 0)
                tn[v[i]/2].leftSubNode = tn[i];
            else
                tn[v[i]/2 - 1].rightSubNode = tn[i];
        i++;
    }
    return tn[0];
}
                        a)
```

```
val(ind){
    return ind;
}
        b)
```

Figure 4. a) An algorithm for obtaining the binary tree of a given rank; b) The auxiliary algorithm for obtaining the value of a node of the binary tree, based on its index

The time complexity analysis of rank2tree is concerned with the overhead from calculating the value of an individual Catalan Cipher Vector element, based on the rank, by traversing the Catalan Triangle. In the worst case [3] it takes $2n - 1$ time units and in the best case it takes $n - 1$ time units to obtain an individual element of the vector from the given rank. Since each of the n nodes of the tree will be generated once (and since the algorithm val is $\Theta(1)$), this means that the overall time complexity of rank2tree is $\Theta(n)$. For the space complexity, it can be seen that the only memory elements with variable sizes are the two arrays that represent the intermediate Catalan Cipher Vector and the nodes of the binary tree respectively. Therefore, the space complexity of this algorithm is also $\Theta(n)$.

To obtain the rank for a given binary tree, first the Catalan Cipher Vector would need to be obtained from the tree and then the total rank would be calculated by the contribution from each element of the vector. The Catalan Cipher Vector is linked with the canonical state-space tableau in which the left and right sub-nodes for every node are represented. To obtain each element of the Catalan Cipher Vector from each node of the tree, the tree would need to be traversed in a way so that, for a given root, first the left sub-node would be traversed and then the right sub-node, but without going into recursion, i.e. the traversal would be by depth. By definition [2], this is level-order traversal and it is therefore used as the traversal of choice for obtaining each element of the Catalan Cipher Vector for each node.

Algorithm tree2rank (Figure 5a) shows how each element of the Catalan Cipher Vector is obtained following the level-order traversal of the tree. When it is obtained, it is used to update the overall rank by using algorithm update (Figure 6b). Since tree2rank incorporates the level-order traversal of a tree, which is a $\Theta(n)$ algorithm, its time complexity analysis requires an analysis of algorithm update. update contains a loop which runs only if displacement occurs, of the column of the Catalan Triangle under consideration. This happens when there is a change in the value of an element of the Catalan combination ($c_i = v_i - i$) relative to the previous element ($c_{i-1} = v_{i-1} - (i - 1)$) and the loop will not run if those two elements are identical. The numbers of displacements from the previous to the next element in the Catalan combination (or Catalan Cipher Vector) are actually the elements of the corresponding Codeword, and the sum of all elements in the Codeword for a given n is at most $n - 1$ (examples for $n = 4$ are given in Table 1). Thus, when no displacement occurs, there are n movements through the Catalan Triangle, and this is the best case, i.e. the algorithm is $\Omega(n)$. The worst-case scenario occurs when there are $n - 1$ displacements in the Catalan Triangle, which means there are $n + (n - 1) = 2n - 1$ movements, i.e. the algorithm is $O(n)$. This means that the time complexity of update is $\Theta(n)$, so the overall time complexity of tree2rank is also $\Theta(n)$.

For the space complexity analysis, it can be seen that tree2rank requires an array v of n integers, which represents the resulting Catalan Cipher Vector, and a queue q of nodes, which will occupy various amounts of memory, depending on the tree being traversed. Assuming that each node occupies k units of memory, and the integers each take a unit of memory, the space complexity can be analyzed. In the best case, one of a degenerate tree, there will be only one node in the queue during the level-order traversal, which means that the total memory usage will be $n + k$ and thus the space complexity will be $\Omega(n)$. In the worst case, that of a complete binary tree, for every node dequeued two more nodes will be enqueued, so there will be at most $\frac{n+1}{2}$ nodes in the queue and thus the full memory usage will be $n + k\frac{n+1}{2} = n\left(1+\frac{k}{2}\right)+\frac{k}{2}$, i.e. the space complexity will be $O(n)$. Therefore, the space complexity will also be $\Theta(n)$.

```
tree2rank(t, CT, n){
   define q, v, i, row, rank, p;
   v = new array(n);
   i = 1;
   row = 0;
   rank = 0;
   q.enqueue(t);
   v[0] = 0;
   while(!q.isEmpty()){
      p = q.dequeue();
      if(p.leftSubNode != NULL){
         q.enqueue(p.leftSubNode);
         v[i] = 2*row + 1;
         update(rank, CT, v, i);
         i++;
      }
      if(p.rightSubNode != NULL)      {
   q.enqueue(p.rightSubNode);
         v[i] = 2*row + 2;
         update(rank, CT, v, i);
         i++;
      }
      row++;
   }
   return rank;
}
```

a)

```
update(ref rank, CT, v, i){
   define prev, next, j;
   prev = v[i-1]-(i-1);
   next = v[i]-i;
   for(j = prev; j < next; j++)
      rank = rank + CT[i][j];
}
```

b)

Figure 5. a) An algorithm for calculating the rank from a given binary tree. b) An auxiliary algorithm for updating the rank given the Catalan Cipher Vector and the index of the element which contributes to the overall rank

Special Forms of the Binary Trees Obtained from Certain Ranks

The goal of the enumeration of binary trees is to establish a 1-to-1 relation between a binary tree and its representation. In this paper, every binary tree has a unique Catalan Cipher Vector and a unique rank related with it. The advantage of the ranking system presented in this paper is that certain ranks always produce certain forms of the binary trees. It can be viewed in Table 2, for example.

The rank of 0 is equivalent to the initial Catalan Cipher Vector, where each element of the vector has the value of its index, or every $v_i = i$, for $0 \le i \le n-1$.In the canonical state-space tableau, this corresponds with a tableau where the second and third column would be filled up as follows: first the second column of the first row, then the third column of the first row, then the second column of the second row, then the third column of the second row and so on. In other words, every node will obtain first a left then a right sub-node, before the same thing happens for the next node in the traversal. Since the traversal is level-order, the levels of the tree will be filled from left to right and the nodes will have both left and right sub-nodes, or only a left sub-node (if there is an even number of nodes; there will be only one such node), or no sub-nodes (such will be the leaves). This means that all levels will be completely filled, except the last level, where all nodes will be positioned as far left as possible. This is the definition of a complete binary tree [1]. Therefore, it can be said that algorithm rank2tree on Figure 5 will create the complete binary tree of n nodes, if the given rank is 0. Equivalently, a tree of rank 0 is a complete binary tree for any number of nodes n, when using algorithm rank2tree. For $n = 2^k - 1, k \geq 1, k \in \mathbb{N}$, the tree with rank 0 is a full binary tree.

The rank of $C_n - 1$ yields a Catalan Cipher Vector where $v_i = 2i, \text{ for } 0 \leq i \leq n-1$. In the canonical state-space tableau, this corresponds with a tableau where only the third column is filled for every row except the last. In the generated tree, this means that every node (except the last) will have only a right sub-node, which is a feature of a degenerate tree. Therefore, it can be said that algorithm rank2tree on Figure 5 will create a degenerate tree of n nodes, if the given rank is $C_n - 1$. Equivalently, a tree of rank $C_n - 1$ is a degenerate tree for any number of nodes n, when using algorithm rank2tree.

Conclusion

A new way of enumerating binary trees, called the Catalan Cipher Vector, is introduced. Another representation, the canonical state-space tableau, helps to show how the Catalan Cipher Vector determines the entire topology of the binary tree. Algorithms are given which show how to obtain the rank from a given binary tree and vice versa, using the Catalan Cipher Vector within the algorithms themselves. It is shown how, following those algorithms, certain ranks always produce certain forms of the binary trees. Since the algorithms for conversion from a binary tree to its respective rank and vice versa are linear in both time and space complexity, while utilizing the Catalan Cipher Vector as an intermediate result, it is an efficient representation. To the best of the knowledge of the authors, the Catalan Cipher Vector is the first enumeration that utilizes level-order traversal to generate the binary tree from itself, and it is their belief that it is therefore also more intuitive and elegant.

Authorship statement

Author(s) confirms that the above named article is an original work, did not previously published or is currently under consideration for any other publication.

Conflicts of interest

We declare that we have no conflicts of interest.

References

[1] Black, P. E. "complete binary tree", in *Dictionary of Algorithms and Data Structures* [online], Paul E. Black, ed., U.S. National Institute of Standards and Technology. 26 May 2011. Available from: http://www.nist.gov/dads/HTML/completeBinaryTree.html. Accessed on July 4, 2013

[2] Black, P. E. "level-order traversal", in *Dictionary of Algorithms and Data Structures* [online], Paul E. Black, ed., U.S. National Institute of Standards and Technology. 26 May 2011. Available from: http://www.nist.gov/dads/HTML/levelOrderTraversal.html. Accessed on July 30, 2013

[3] Črepinšek, M. and Mernik, L. (2009). An Efficient Representation for Solving Catalan Number Related Problems. International Journal of Pure and Applied Mathematics, Vol. 56, No. 4, 589-604.

[4] M. C. Er. (1985). Enumerating Ordered Trees Lexicograpically. The Computer Journal, Vol. 28, No. 5, 538-542.

[5] Knott, G. D. (1977). A Numbering system for binary trees. Communications of the ACM, Vol. 20, No.2, 113-115.

[6] Mäkinen, E. (1987). Left Distance Binary Tree Representations. BIT Numerical Mathematics, Vol. 27, No. 2, 163-169.

[7] Pallo, J. M. (1986). Enumerating, Ranking and Unranking Binary Trees. The Computer Journal, Vol. 29, No. 2, 171-175.

[8] Proskurowski, A. (1980). On the generation for binary trees. Journal of the ACM Vol. 27, 1-2.

[9] Roelants van Baronaigen, D. (1991). A loopless algorithm for generating binary tree sequences. Information Processing Letters 39, 189-194.

[10] Rotem, D. and Varol, Y. L. (1978). Generation of binary trees from ballot sequences. Journal of the ACM, Vol. 25, No.3, 396-404.

[11] Ruskey, F. and Hu, T. C. (1977). Generating Binary Trees Lexicographically. SIAM Journal on Computing, Vol. 6, No. 4, 745-758.

[12] Xiang, L., Tang, C. and Ushijima, K. (1997). Grammar-Oriented Enumeration of Binary Trees. The Computer Journal, Vol. 40, No. 5, 278-291.

[13] Zaks, S. (1980). Lexicographic Generation of Binary Trees. Theoretical Computer Science, Vol. 10, 63-82.

[14] Zerling, D. (1985). Generating binary trees using rotations. Journal of the ACM, Vol. 27, 694-701.

Submitted: July 31, 2013.
Accepted: December 23, 2013.