mathematics

Article

Novel Static Multi-Layer Forest Approach and Its Applications

Ganesh Bhagwat [1], Shristi Kumari [2], Vaishnavi Patekar [3] and Adrian Marius Deaconu [4,*]

[1] Mercedes-Benz Research and Development India, Whitefield, Bangalore 560066, India; ganesh.bhagwat@daimler.com
[2] Robert Bosch Engineering & Business Solutions Pvt. Ltd., Adugodi, Bangalore 560004, India; shristi.kumari@in.bosch.com
[3] KPIT Technologies Pvt. Ltd., Bellandur, Bangalore 560103, India; vaishnavi.patekar@kpit.com
[4] Department of Mathematics and Computer Science, Faculty of Mathematics and Computer Science, Transilvania University of Brasov, 50003 Brasov, Romania
* Correspondence: a.deaconu@unitbv.ro

Abstract: The existing multi-layer tree is of dynamic linked list type which has many limitations and is complicated due to the pointer-node structure. Static array representation gives more flexibility in programming of algorithms and operations like insertion, deletion, and search. It also reduces the storage space. This paper presents a new method for representing multi-layer forest data structure in array format. It also explains various tree operations, unique data compression algorithm and migration algorithm between traditional approach and the proposed data structure. Most of the fundamental algorithms like those from artificial intelligence that employ decision trees are based on trees/forest data structure. The current paper brings a completely new idea in the representation of these data structures without employing recursion and targeting memory optimizations with reduced code complexities. The applications of forest data structures are many and span over various interdisciplinary areas of Engineering, Medicine, Aviation, Locomotive, Marine, etc. The proposed novel approach not just introduces a new method to look at the tree data structure but also provides the flexibility to adapt to the existing methods as per the user needs. A few such applications in Simulink debugging and the Forest visualisation have been discussed in detail in this paper.

Keywords: data structure; tree forest; tree traversal; tree visualization

check for updates

Citation: Bhagwat, G.; Kumari, S.; Patekar, V.; Deaconu, A.M. Novel Static Multi-Layer Forest Approach and Its Applications. *Mathematics* **2021**, *9*, 2650. https://doi.org/10.3390/math9212650

Academic Editor: Ezequiel López-Rubio

Received: 27 September 2021
Accepted: 15 October 2021
Published: 20 October 2021

1. Introduction

Storing information and being able to process it as per the requirement is the most essential feature of a database or computer science in general. When the database is discrete, it is pretty straightforward as the order of storage does not matter. In cases where each data is linked to another, we have to store it accordingly as a stack. However, when one data is dependent on multiple data, it has to be represented in a tree. The concept of tree originates from the need to store data hierarchically. Tree is the second most generic data structure next to graphs. A tree is an abstract data type represented as a link of nodes starting with a root. All the nodes are connected (between every two nodes there exists a path) and there are no cycles. There are various tree storage applications like folder structure, circuit information, Simulink modeling, etc.

Since a tree is a generic data structure, there are a lot of studies on its storage, operations, and applications. Some of the traditional approaches can be found in [1,2]. There are many types of trees available like binary-tree, AVL, Red-black tree, N-ary tree [2]. Usually, distinguishing the tree type based on the need and application eases the algorithm implementation as seen in one of the examples in [3] that uses binary trees implementation in FPGA for Huffman encoding. An advantage of using specific trees like the binary tree is the static storage of data like an array [2]. In static storage, it is easy to go back and forth or pinpoint data using an index which becomes complex with dynamic storage. We can also find another example in [4] which mentions non-recursive algorithms for traversal of binary trees.

Another operation or application of trees is visualization. In [5], the author describes binary tree visualization using post-order traversal. We have tried to modify this approach and present a more generic approach to visualize all kinds of trees and not just binary trees. A forest is a data structure that stores multiple trees. A data structure storing a forest rather than a tree is clearly a generalization. We propose an algorithm where we can intuitively imagine an empty information node connecting the roots of the trees of the forest. Thus, a whole forest can be represented as a tree. By doing this we would have a generalized way of storing multiple trees making this a forest data structure rather than a single tree data structure.

There exists many tree representations that can be chosen as per the application requirement [6]. Some of these are Balanced Parentheses Sequence(BP) [7], Level Order Unary Degree Sequence (LOUDS), and Depth First Unary Degree Sequence (DFUDS) [8]. A lot of work regarding their variation and comparison can be found in [6]. For the purpose of comparison, some of the applications in this paper can also be examined using these representations. There are many other works of literature that are using and enhancing these approaches. For example, the work by Hector Ferrada [9] is an improvisation on the Balanced Parentheses representation. We can also see the LOUDS++ representation proposed by O'Neil Delpratt in [10]. We propose a deviation to these representations considering the application with an aim to reduce time complexity.

In the traditional approach, the tree's nodes are accessed in a recursive manner [1]. Every node is linked with the next nodes (called children). Usually, the tree is stored using linked lists. In our approach, we represent this dynamic information as a static data structure. Rather than generalizing trees with a linked list [2] approach if we try different representations and choose the algorithm closest to the application needs it will be more efficient. In [11], various methods of representing a tree data structure are discussed. In this paper, we are introducing another novel multi-layer forest representation.

For proof of concept that the proposed method is generalized to all multi-layer trees and also to be able to use the traditional linked list approach when required, we have proposed a method for re-construction of the linked list data structure from the proposed tree storage algorithm. The reconstruction is a derived idea from the pre-order traversal method in [12] but traversal in the proposed algorithm was not done as first or second pre-order, rather we have used the parent-child relation to do the same. Section 4 talks about the details of this approach.

Usually, trees represent large quantities of data. Thus, it becomes important to look at storage optimization of the data structure as well. A lot of literature exists on the storage of the data structure. Some speak of a generic approach while some make it specific to the algorithmic needs by targeting structural compactness using encoding methods. One such technique of optimization can be seen in [13] in which the characteristic of the tree is changed using Lempel–Ziv–Welch (LZW) compression technique [14]. One can also find tree compression analysis based on its orientation in [15], while trying to introduce a new algorithm for the tree data structure in this paper that has operational ease, we have also placed importance on the storage compactness of the algorithm which is introduced in Section 7. The algorithm is not just used from a storage or data structure point of view but can also be used in algorithms like the decision tree, random forest, and other pruning applications. Traditionally, in all these applications recursion is innately used. However, in the case of recursion which works as depth-first, controlling the path of the tree is not easily possible. It can only be stopped. We can substitute our proposed method in these applications.

Considering a decision tree, most of its literature work is based on changing the scoring mechanism, subdivision splitting or changing the stopping criteria. A combination of all these parameters gives rise to a lot of different algorithms in decision trees. One of these algorithms is Rank Entropy-Based Decision Tree (REMT) [16] in which a new scoring method and stopping criteria is introduced. Qinghua Hu [16] has also compared it with the traditional scoring of the Gini CART algorithm. In both cases, the CART and the REMT, the underlying building path and tree growing algorithm is the same recursion based approach.

Static tree algorithms might give more ability to control the path and trace back to correct weight based on future node prediction. RVMEA/OL [17] is one of the algorithms in the decision tree where path transition probability is also used. This can be explored and researched more by controlling the path itself, by migrating to static tree architecture it would provide greater flexibility and possibilities to these approaches. The proposed algorithm reduces the complexity of recursion and also makes it easier to find and access a node since it is stored directly in a static array. More research would be required in this aspect, where the fundamental static tree can be manipulated as per the requirement.

Considering the fundamental nature of the tree and forest algorithm, there are many different domains where the traditional approach is immediately considered, however considering a different static tree approach for each of those possibilities will expand the scope of research in a lot of areas. Artificial intelligence (AI) is one of these more advanced areas in which if we can modify and optimize the decision tree, then the algorithms depending on it can also be explored in a deeper sense. Some of these algorithms would be random-forest [18], Recursive partitioning [19], Behavior tree [18], and many more. There would obviously be a trade-off and more modification of the algorithm required for each of these applications. Another additional feature would be the possibility of visualization (Section 6) of these trees in a graphical manner which would help in understanding the approach in a more intuitive way.

There exists minimal literature that addresses bug finding in Simulink and process automation in the testing phase of the V-model of software development. Representing the Simulink model as a forest data structure gives the flexibility for automation of debugging processes like root cause analysis for bug finding, impact analysis of dependent features etc. With the data structure proposed in this paper, this has been briefly explained in Section 6.2 with solutions to problems of subsystems and looping concepts. The idea of treating Simulink as an XML has been briefly described in [20]. Further work on Simulink XML for developing machine learning solutions for bug finding has been described in [21]. The approach for our proposed data structure derives its closeness to the Fault tree analysis method for MATLAB and Simulink [22].

The paper is organized as follows. In Section 2, we introduce the idea to approach a forest without recursion. We show that the minimal data needed to store a forest consists of the information of nodes and the number of children for each node. In the upcoming Sections 3 and 4, we describe the traditional operations performed on our data structure: traversal, node search, parent search, insertion, deletion, and how the new data structure can be converted to a traditional linked structure. The advantages of this algorithm are pointed out in Section 5 followed by some tree applications in Section 6. The compression algorithm for storage optimization is laid out in Section 7 followed by future scope in Section 8. Conclusions are drawn at the end of the paper in Section 9.

2. Presentation of the Newly Proposed Data Structure

In this paper, a new data structure is proposed to store a tree and even a forest in an unconventional way, without using recursion when accessing data. Usually, for storing a tree, at least three vectors are needed. The advantage of this new approach is that any forest can be easily represented with a combination of only two vectors, each having the length given by the number of nodes. For the explanation of the data structure, we take the example from Figure 1. For a given forest, we tried to recreate it using minimized saved information. This has been achieved by storing information of nodes and, separately, the number of trees and the number of children of each node. So, we have two arrays. The first one stores the information of the node (A, B, C, etc., as seen in Figure 1). The second array stores the number of trees in NoOfChildren[0] and then stores the number of children of the nodes. Because the 0th index of NoOfChildren is reserved for the number of trees, the lengths of Info array and NoOfChildren array differ by 1. As it can be seen in the example below, the number of trees is NoOfChildren[0] = 3. So, the 3 root nodes are A, B, and C. Now, A has 3 children (D, E, and F) and this number is stored in NoOfChildren[1]. After DEF are exhausted as children of A, we move to the children of B and store them after the sequence ABCDEF. B has 2 children (G and H) and this fact is stored in NoOfChildren[2] and so on. Thus, as can be seen, these two arrays are sufficient to reconstruct the forest. This idea works by storing the level information, i.e., breadth-first manner.

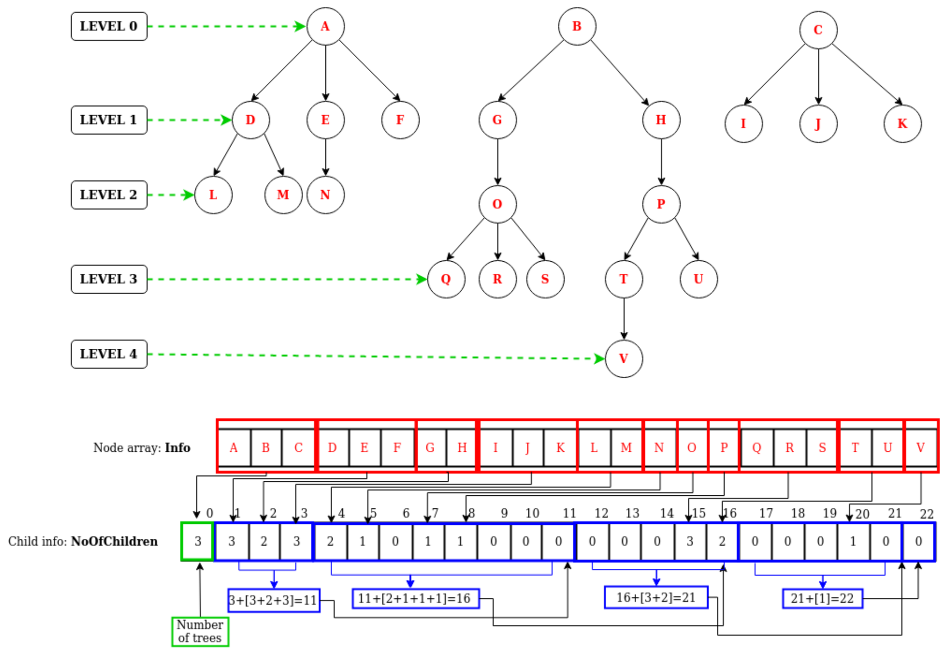

Figure 1. Forest Data Structure Representation.

3. Common Operations

In this section, we present the algorithms that can be used to perform general operations on the newly introduced forest data structure: forest traversal, search, insertion and deletion.

3.1. Traversal

With the traditional approach of linked lists and recursion, tree traversal is a depth first approach but for our data structure we move to breadth first idea which is more appropriate in certain applications. In this paper, we propose four different methods of traversal. The first method is the traversal of the forest on levels. The second and third methods are the traversal of the left and, respectively, the rightmost children. The fourth method is similar to the first method but here, we traverse only a tree or a subtree, i.e., pick a node and traverse the corresponding subtree (its children, grandchildren, and so on).

3.1.1. Traversal on Levels

In this method, the nodes are traversed level wise as seen in Figure 2. Since NoOfChildren[0] is the number of trees and the rest of the NoOfChildren array stores the number of children for each node, the sum of all elements is one less than the length of the array. This is the broader idea for the outer loop to run, i.e., the position of the current node is less than the length of the array. We also group the levels in NoOfChildren by summing up the number of children from the previous level. For instance, in Figure 2 we have NoOfChildren [0] = 3 roots (A, B, and C). We then have a total of 3 + 2 + 3 = 8 children

(DEFGHIJK) of these three root nodes. The sum of these values gives the index of the last element of the next level. The pseudo-code is Algorithm 1.

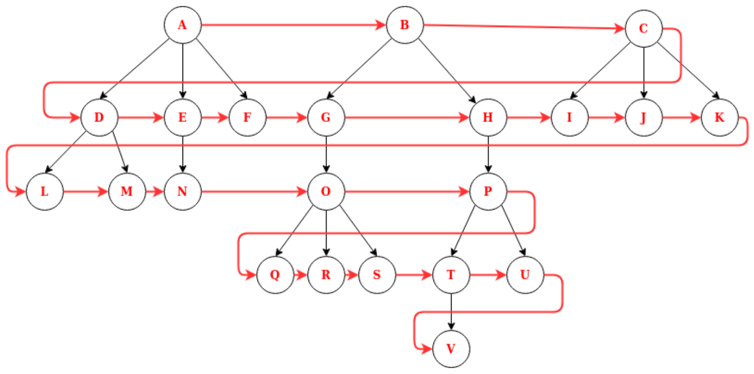

Figure 2. Traversal On Levels.

For an index m we have:

$$S_m = \sum_{i=1}^{m+1} NoOfChildren[i] \tag{1}$$

S_m = Index of last child of Info[m]
where, S_m = rightmost child index
S_{m-1} = 2nd child index
S_{m-2} = 3rd child index
....
S_m-NoOfChildren[m] = leftmost child index.

3.1.2. Left and Right Extremities Traversals

This method of traversal does not visit every node. It is similar to the traversal of only an end element or beginning element of that level of the forest. In this method, we either traverse only the leftmost nodes or the rightmost nodes of a given forest. Figure 3 depicts the traversal of only the leftmost nodes. This traversal does not use all the nodes present in a forest. This can be useful in some special cases as per the requirement and, the time complexity of this approach is significantly less. One of the applications of this traversal is to find the level of a given node. This will be mentioned in detail in the search section of the paper.

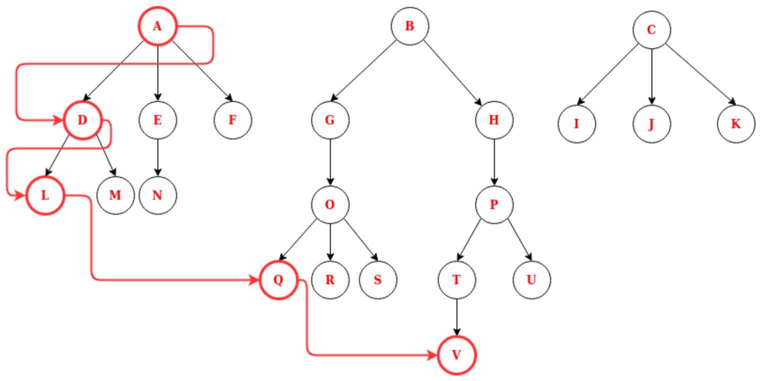

Figure 3. Leftmost Traversal.

3.1.3. Leftmost Traversal

We use Equation (1) as the reference to build the algorithm. Rather than iterating through all the nodes, we only visit "S_m − NoOfChildren[m + 1]" nodes. Thus, only the leftmost nodes are traversed.

Algorithm 1 Traversal on levels

```
1: Info
2: NoOfChildren
3: n = length(Info)                                   ▷ Total number of nodes
4: T = ""                                             ▷ T is the final output
5: Left = 0                                           ▷ Leftmost index of a given level
6: Right = 0                                          ▷ Rightmost index of a given level
7: PosInfo = 0                                        ▷ Position of a current node
8: while $PosInfo < n$ do
9:     TotalChild = 0
10:                      ▷ Iterate level from the leftmost to the rightmost child
11:    for $PosChild \leftarrow Left$ to $Right$ do
12:        for $i \leftarrow 0$ to $NoOfChildren[PosChild]$ do
13:            T = T + Info[PosInfo]
14:            PosInfo = PosInfo + 1
15:        end for
16:              ▷ When PosChild is equal to Right, it would mean end of a level
17:        if $PosChild < Right$ then
18:            T = T + "*"
19:        end if
20:        TotalChild =
                   TotalChild + NoOfChildren[PosChild]
21:    end for
22:                                                   ▷ Update the variables
23:    Left = Right + 1
24:    Right = Right + TotalChild
25:    Print the next level in a Newline
26: end while
```

3.1.4. Rightmost Traversal

This algorithm is similar to the leftmost traversal but here, instead of nodes at "S_m - NoOfChildren[m]" we visit only the nodes at "S_m". Thus, the pseudo-code for the rightmost traversal is similar to that of the leftmost.

3.1.5. Traversal of a (Sub)Tree

The idea here is to traverse only one component of the forest: a tree or a subtree. The information of a node is provided as the starting node to traverse through. Only the (sub)tree having that node as root will be traversed by this algorithm. Let us consider an example where all the nodes connected to 'P' have to be traversed, as shown in Figure 4. Referring to Equation (1), where in this case, m = P, S_P indicates the last child of P, i.e., node V. Thus, we can traverse through the nodes as follows.

$S_c - 0 \rightarrow T$
$S_c - 1$
... .
$S_c - NoOfChildren[c] \rightarrow U$

Once we have traversed all the children of C, we proceed to the grandchildren in the same manner. However, we have to know when to end this loop. For that, we are storing the values of the visited nodes from Info and NoOfChildren in two new arrays which contain only the information of the sub-tree. For each visited node, the information and the number of children of that node is pushed into the sub-arrays as seen in Figure 4.

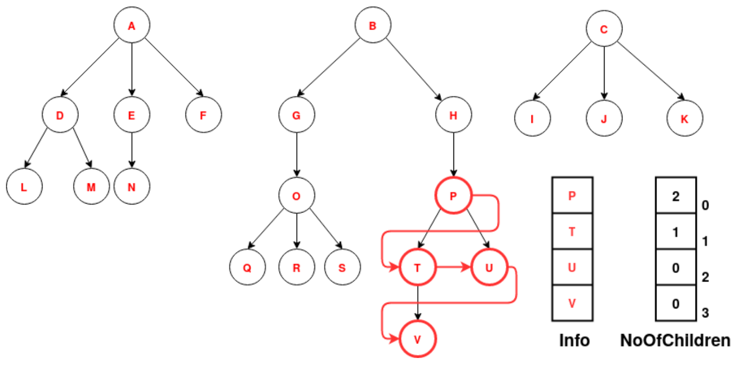

Figure 4. Sub-Tree Traversal.

3.2. Search

To find a given node in a forest the whole traversal is not required. We can search linearly in the Info array. However, to find other information like the parent, child, or the level of a node or search a sub-tree, the algorithm explained in this section would be more convenient. This section has been sub-categorized into three operations: finding the parent node, finding multiple occurrences of information with level information, and also finding the path from root to node.

3.2.1. Finding the Parent Node

The operation of finding the parent node has many applications. Operations like search, insertion, and deletion become easy with the information of the parent node. To find the index of the parent, we first proceed with a linear search for the index of the given node. From Equation (1), we know that

$$S_m = \sum_{i=1}^{m+1} NoOfChildren[i]$$

where, S_m = Index of the last child of Info[m]

In this equation, we already know the index of the child from a linear search. Equation (1) can thus be modified as below.

$$Ch_j = \sum_{i=0}^{m+1} NoOfChildren[i] - j \tag{2}$$

where, j ∈ $\{0, 1, 2, ..., NoOfChildren[m]\}$. Now we need to find m. Rearranging Equation (2), we get,

$$Ch_j + j = \sum_{i=0}^{m+1} NoOfChildren[i]$$

$$Ch_0 \leq \sum_{i=0}^{m+1} NoOfChildren[i] \leq Ch_0 + NoOfChildren[m]$$

The pseudo-code for the parent search is Algorithm 2.

3.2.2. Finding Level

For finding a level, we have to search the index and traverse the rightmost nodes since the rightmost values give the index of the last element in that level. Thus, Equation (1) is modified as below.

$$S_m = \sum_{i=1}^{m+1} NoOfChildren[i] = \text{Index of the last child of Info[m]}$$

Algorithm 2 Finding the parent node

```
1:  Info
2:  NoOfChildren
3:  Element
4:  i = 0
5:  Sum = 0
6:  k = [Find index where Element is present in Info] + 1
7:  while $Sum < k$ do
8:      Sum = Sum + NoOfChildren[i]
9:      i = i + 1
10: end while
11: Parent = i − 2
```

For m index of a parent, the index of the rightmost node is S_m. As we have already seen in the rightmost traversal algorithm, we are only visiting the rightmost (last) nodes at any level. The index of the node should lie between S_m and S_{S_m}.

$$S_m \leq Index \leq \sum_{i=0}^{S_m} NoOfChildren[i]$$

3.2.3. Finding Multiple Occurrences of the Information

In some applications, there could be a case of multiple occurrences of an information in a forest. In this case, we only need to make use of the previous algorithm to find the level and iterate through the tree till the end. This will store the index value and respective levels of all the occurrences of the node.

3.2.4. Finding Path

Given a forest, for some applications, it might be needed to know the path in the backward direction from the current node to the root node of the tree in the forest. For example, taking 'U' as the node, then the path for U is $U \rightarrow P \rightarrow H \rightarrow B$ as shown in Figure 5. This might be helpful especially in applications involving files and folder structure. So to implement it, the idea is to find the parent of the given node. Then we repeat the process as explained above to find the parent by feeding the nodes encountered. The process ends when the parent node is equal to a root node or a node in level 0 which would be represented as an $index > NoOfChildren[0]$.

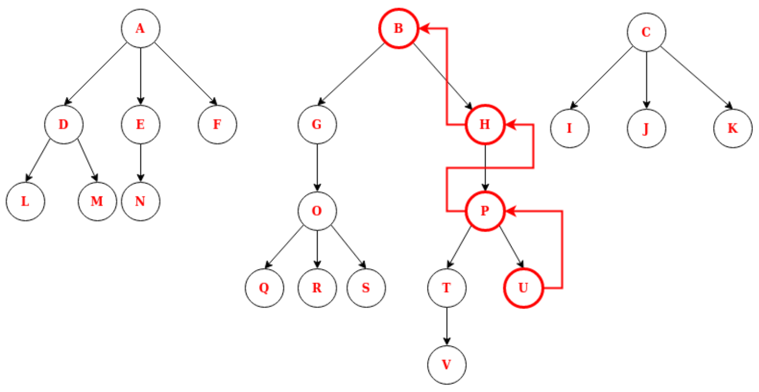

Figure 5. Finding Path.

3.3. Deletion

In the traditional approach, the deletion of a node is pretty straightforward. Here we link the parent to the children of the deleted node. However, in the data structure proposed in this paper, removing a single node will disrupt the whole array. So to remove a node, we need to take care of the sub-tree of the deleted node. We have given two methods for this purpose, first the deletion of an entire sub-tree and second, the deletion of a given node and linking its children with the parent of the deleted node. However, in both of the methods it is necessary to find the sub-tree. We can use the same algorithm that we used for the traversal of the sub-tree which gives two sub-arrays representing the nodes and the number of children in the sub-tree. Along with this, while storing the sub-tree we can also store the index values of the nodes in the original tree which makes the deletion easier. With this information, the two methods are explained below.

3.3.1. Deletion of Sub-Tree

Since we know the indices of the nodes to be deleted, we can easily pop them, i.e., remove the nodes from Info and NoOfChildren. However, the problem that arises here is that if we keep popping nodes from the start (the root of the sub-tree), the indices of the remaining elements to be deleted get changed. Thus, to overcome this, we remove elements from the end (leaf of the sub-tree) and finally reduce the children count of the parent of the deleted node by 1.

Let us consider the example of the deletion of node 'P' and its successors, then the forest after deletion should look like in Figure 6. For applications with such requirements, it is also possible to isolate a sub-tree from a base tree as the information regarding the sub-tree is already present in the sub-arrays.

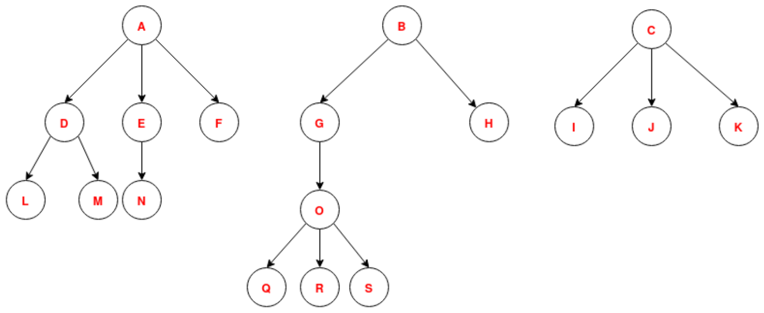

Figure 6. Deletion of Sub-tree.

3.3.2. Deletion of a Node and Linking Its Children with a Parent

This method is similar to the one mentioned above but here instead of popping all the elements we only delete a single node. Here, we have to reattach the children of the deleted node with its grandparent to establish the links in the tree. The idea is to go backward. Rather than removing the elements from the base array entirely, we pop the node and insert it at the index of its parent. Due to this insertion, the position of the next element to be popped changes to the index that was determined earlier. For this purpose, we store the parent information (index) of each node in a separate array called 'parent'. So, we know that the pop index and the insertion index would be exactly equal to the parent index.

Let us consider, deletion of node 'P' from the base forest. Refer the pseudo-code in Algorithm 3.

3.4. Insertion

Insertion is one of the operations in our algorithm that is similar to the traditional approach. As we have already presented an algorithm for finding the parent node, insertion operation becomes quite easy. The idea here is to get the information of the new node and its parent node. In the case the node to be inserted is the root, the parent node becomes 'NA'. For other than the case of insertion of the root node, it is necessary to find the index of the parent and increase its number of children by 1.

Algorithm 3 Deletion

```
1: procedure DELETE(Node)
2:     Get StackValue, StackNoofchildren and StackIndex
       from SubTree (Node)
3:     Get RootParent from parent SubTree function
4:     j = 0, Parent = [ ]
5:     Parent.append(RootParent)
6:     for $i \rightarrow 0$ to $length(StackValue)$ do
7:         for $k \rightarrow 0$ to $StackNoofchildren[i]$ do
8:             Parent.append(StackIndex[i])
9:         end for
10:    end for
11:                              ▷ Reconnecting the child tree to the grandparent node
12:    for $i \rightarrow length(StackValue)$ to 0 do
13:        k = Info.index(StackValue[i])
14:        del Info[k]
15:        del NoOfChildren[k+1]
16:        if $Parent[i] >= 0$ and $i! = 0$ then
17:            Insert Parent[i] + 1 in Info at StackValue[i]
18:            Insert Parent[i] + 2 in NoOfChildren at StackNoofchildren[i])
19:        end if
20:    end for
21:    NoOfChildren[RootParent + 1]=
   NoOfChildren[RootParent + 1] + StackNoofchildren[0]-1
22: end procedure
```

Referring to Equation (1), we can get information about the rightmost node of the parent. We can thus simply insert the new node in the rightmost position. We also have the choice of inserting the node in the leftmost position. To insert the child in the leftmost position we simply subtract the number of children in Equation (1) ($S_m -$ NoOfChildren[m + 1]). This gives the index for the leftmost insertion. Thus, in this approach, we present the idea for positional insertion, leftmost or rightmost. Refer the pseudo-code in Algorithm 4 for leftmost insertion.

3.5. Pre-Order Traversal

As our data structure represents a forest, we could be interested in traversing the tree by requesting the root node. To understand the algorithm, let us consider the traversal of the second tree represented via root B (Figure 1). From the knowledge of the pre-order traversal technique, we know that the sequence of traversal will be BGOQRSHPTVU. To obtain this sequence, we first traverse the leftmost nodes starting from B by the method demonstrated in Section 3.1.3 Leftmost Traversal. The traversal halts when the traversed node has no further children. So the sequence obtained until now is BGOQ, while traversing the nodes, we also simultaneously store the index of the current node and the number of children of the current node in separate new arrays, *index* and *traversalchildren*, respectively. Each time a child is traversed, the count of no. of children of its parent is decreased by 1. So the array *traversalchildren* represents the no. of children left to be traversed. To proceed further, we find the last element from array *traversalchildren* having a non-zero value which in this case happens to be O represented by 2. This means O still has two other children left to be traversed. To traverse the next child of O, we need to determine the child index which can be given by the below formula:

$$Newindex = index[i+1] + NoOfChildren[index[i]] - traversalchildren[i] \quad (3)$$

where, i is the index of the parent node (non-zero element). Index [i + 1] gives the index of the first child of O, i.e., Q in the original array Info. Thus, by adding $(3-2)$ 1 gives the index of the next child, i.e., R. With the index of the new node, we apply again the same leftmost node traversal technique which gives us R. *traversalchildren* for O becomes 1. Proceeding the same way gives us S. This continues until the sum of all the elements of *traversalchildren* becomes 0. The pseudo-code for the pre-order traversal is given in Algorithm 5.

Algorithm 4 Insertion

```
 1: Info
 2: NoOfChildren
 3: procedure INSERT(Node, Parent)
 4:                                  ▷ Check if the Node is the root and the first node
 5:     if ParentName==’N/A’ && length(NoOfChildren) then
 6:         Insert NodeName in Info at index 0
 7:         Insert 1 in NoOfChildren at index 0
 8:         Insert 0 in NoOfChildren at index 1
 9:         i = i + 1
10:     else
11:         if ParentName==’N/A’ then                ▷ If root Node but not first Node
12:             Insert NodeName in Info at index 0
13:             NoOfChildren[0] = NoOfChildren[0] + 1
14:             Insert 0 in NoOfChildren at 1 index
15:             i = i + 1
16:         else
17:             if ParentName in info then
18:                 PIndex = info.index(ParentName)
                       ▷ PIndex will get the index value where it lies in the Info array
19:             else
20:                 PIndex = −1
21:                 Print "Parent Node Not Exist"
22:             end if
23:             Sum = 0
24:             if PIndex >= 0 then
25:                 Sum = sum(noofchildren[0:PIndex + 1])
26:                 Insert NodeName in Info at Sum index
27:                 NoOfChildren[PIndex + 1]++
28:                 Insert 0 in NoOfChildren at Sum + 1 index
29:             end if
30:         end if
31:     end if
32: end procedure
```

3.6. Post-Order Traversal

The expected outcome of post-order traversal is QRSOGVTUPHB. To obtain this sequence, we start with an approach similar to that mentioned in the previous section. We start traversing the leftmost branch of the tree and store the nodes in another array called the path, its index in the index array and children to be traversed in traversal children. As soon as we reach the leftmost leaf, we pop the node from path and move it to an array named post. Thus, we get the first element of our sequence, i.e., Q. To decide on the next element, we check if the last element of array path correspondingly has non-traversed children in array traversal children.

Algorithm 5 Pre-order Traversal

```
Info
NoOfChildren
procedure LEFT(Element)                                    ▷ Go to the leftmost Node
    k = Info.index(Element) + 1
    s = sum(NoOfChildren[0:k])
    Path.append(Info[k-1])
    if $NoOfChildren[k] > 0$ then
        traversalchildren.append(NoOfChildren[k]-1)
    else
        traversalchildren.append(NoOfChildren[k])
    end if
    Index.append(k)
    while NoOfChildren[k]! = 0 do
        k1 = k
        k = s + 1
        s = s + sum(NoOfChildren[k1:k])
        Path.append(Info[k-1])
        if $NoOfChildren[k] > 0$ then
            traversalchildren.append(NoOfChildren[k]-1)
        else
            traversalchildren.append(NoOfChildren[k])
        end if
        Index.append(k)
    end while
end procedure
Path = [ ]                                                 ▷ Output of Pre-Order Sequence
traversalchildren = [ ]
Index = [ ]
Left('B')
while $sum(traversalchildren) > 0$ do
    i = len(traversalchildren)-1
    while $i \geq 0$ do
        i = i-1
        if traversalchildren[i]! = 0 then
            Break
        end if
    end while
    I1 = Index[i + 1] + NoOfChildren[Index[i]]-traversalchildren[i]
    traversalchildren[i] = traversalchildren[i]-1
    Left(Info[I1-1])
end while
```

If the value is 0, the node is popped from path and inserted in array post. On the other hand, if the value is non-zero, i.e., the node has non-traversed children, we move to the traversal of those. The last popped node will be the previous child of the current node. Adding 1 to the index of last node will give the index of the next child (Refer Equation (1)). Once we know the new index, we proceed again with leftmost traversal until all the children of the node are traversed and the parent is popped off the array path. The process is repeated and it ends when there is no element left in the array path. The pseudo-code for the post-order traversal is explained in Algorithm 6.

Algorithm 6 Post-order Traversal

```
1:  Info
2:  NoOfChildren
3:  procedure LEFT(Element)                                  ▷ Go to the leftmost Node
4:      k = Info.index(Element) + 1
5:      s = sum(NoOfChildren[0:k])
6:      Path.append(Info[k-1])
7:      NoChild.append(NoOfChildren[k])
8:      Index.append(k)
9:      while NoOfChildren[k]! = 0 do
10:         k1 = k
11:         k = s + 1
12:         s = s + sum(NoOfChildren[k1:k])
13:         Path.append(Info[k-1])
14:         NoChild.append(NoOfChildren[k])
15:         Index.append(k)
16:     end while
17: end procedure
18: Post = [ ]                                               ▷ Post Order Sequence
19: Path = [ ]
20: NoChild = [ ]
21: Index = [ ]
22: Left('B')
23: while $length(Path) > 0$ do
24:     Post.append(Path[-1])                                ▷ Adding last element in the stack
25:     NextIndex = Index[-1]
26:     NoChild.pop(-1)
27:     Path.pop(-1)
28:     Index.pop(-1)
29:     if $length(NoChild) > 0$ then
30:         NoChild[-1] = NoChild[-1]-1
31:     end if
32:     if NoChild[-1]! = 0 then
33:         Left(Info[NextIndex])
34:     end if
35: end while
```

4. Migration to Conventional Linked List Structure

To convert the traditional tree data structure to the proposed data structure, we can run the traditional level order traversal and store the node names and number of children in two separate arrays which will drive our data structure. However, to convert our storage to the traditional approach we would need another method as this cannot be done directly.

To construct the sequence, we start traversing the tree in a fashion such that we put the current node in the sequence and its children are pushed into a queue. Then we pop the first inserted child (first in, first out), insert it in the sequence, find its children and push it in the queue again. The process continues until the entire tree is constructed. To understand it better, let us take the example of the second tree from Figure 1. We first encounter B and insert it in the linked list. B has 2 children (G, H) that are pushed into a queue. Next, we pop G from the queue and insert it in the linked list which gives us B-G. G has 1 child (O). The queue now has H, O. H is popped. The linked list becomes B-G-H. H has a child (P). The queue becomes O, P. This process continues until we get the entire linked list B-G-H-O-P-Q-R-S-T-U-V. The pseudo-code for this method is given in Algorithm 7.

Algorithm 7 Migration

```
Info
NoOfChildren
struct Node
{ char key
vector<Node *>child;};
procedure NODE *NEWNODE(char key)                    ▷ Create a new tree node
    Node *temp = new Node;
    temp ->key = key;
    return temp;
end procedure
                                          ▷ Creating a generic tree from our tree
Node* Parent                             ▷ Temp Variable to store Parent Nodes
queue<Node *> q;                                             ▷ Create a queue
int k = 0;
int j = 0;
Node *root = newNode(char(Info[0]));
                                                         ▷ First Node Creation
q.push(root);
while !$q.empty()$ do
    Parent = q.front();
    q.pop();
    for $i \to 0$ to $NoOfChildren[j]$ do
        k = k + 1;
        $(Parent\text{-}{>}\ child).pushback($
                          $newNode(char(Info[k])));$
        $q.push(Parent\text{-}{>}\ child[i]);$
    end for
    j = j + 1;
end while
                          ▷ Root has the final linked-list output which is required
```

5. Advantages of the New Approach

Static data structures are more direct compared to dynamic ones and can pin-point to an exact location through the index. The biggest advantage of variable lengths seen in dynamic data structures has been extended to static data structures with advancements in programming languages like list in Python and vectors in C++. Thus, shifting our approach to static tree data structure offers the simplicity of code without restraining us from any advantage offered by a dynamic linked list.

The data structure proposed in this paper is a two array-based approach that offers us easy accessibility of a node especially in scenarios where the node data is important and not its link. We can search a node linearly in the whole array without the hassle of traversing the entire forest. We can group the levels by summing up the number of children from the previous level. The sum of the number of children from the previous level gives the index of the last element of the next level. This helps in direct leftmost or rightmost tree traversals. We can easily switch between different traversal techniques from any node at any point during traversal. The problem of stack overflow encountered with recursion is also completely eliminated and the biggest advantage of all is it lets us stop at any point during the traversal without any loss of data.

Many complex systems represented in trees or forest become easy to approach using the proposed data structure. We will consider a few such applications and explain them in detail in the next section.

6. Applications

From the existing literature, we can say in general that the tree and forest data structures are widely essential in many applications. In most of these applications, replacing the traditional approach with ours might make it easier, computationally less expensive and might also reduce memory consumption targeting the requirements of an application. However, our data structure can be used wherever tree or forest data structures are involved.

Among the various applications of tree algorithms, we can explore our idea fitted to that application, many of which we have discussed in the introduction. Some of the algorithms mentioned in the introduction can be exploited to an application in interdisciplinary research and analysis, mainly in the field of biology where a lot of variables are considered for a given diagnosis. One such application was to distinguish viral and bacterial meningitis as discussed in [23] using multiple parameters like glucose concentration, protein concentration, age, etc. The main idea was to use Chi-square with a decision tree to make a statistical-based model. It is possible here to explore our proposed method for both construction and visualization of the result. Other literatures like [24] have discussed marine container terminal's scheduling algorithm which is developed considering the berths, handling rate and the vessel. The scheduling is to minimize CO_2 emissions and the service cost of vessels. In this application, they have solved the Green Berth Scheduling Problem (GBSP) with a Hybrid Evolutionary Algorithm (HEA). The structure of the algorithm is to deploy local search heuristics. The operation of local search heuristics can be made in a static array manner. There are several other applications as well where modification of a tree data structure would give us different perspectives. Some of their applications are presented in [25,26].

This paper mainly elaborates on visualization techniques and Simulink applications due to the fact that they are purely based on tree data structures. Moreover, exploring these applications do not need to consider any other factor such as scoring function or any other domain-related constraint.

6.1. Visualization

Visualization of a data structure might be helpful while programming or handling a wide cluster of data, especially when a dense forest with multiple trees and branches is involved. There are also other multiple needs of graphical representation of data for the end-users, e.g., when block diagrams of some operations have to be visually represented [27]. There is limited literature on forest visualization and most of the tree visualization algorithms work only for binary trees.

The idea for the visualization algorithm is, for a given level, divide the canvas area by the total number of children based on the position of the parent node. So, for a given node A, we store its centre coordinates Cx, Cy and its number of children N. The space width for its children is designated from X[0] till X[1]. We calculate the centre coordinates of the 1st child as follows.

$$D_c = \text{Distance of one child} = \frac{X[1] - X[0]}{N}$$

$$Ch_{x1} = \text{Centre x-coordinate of first child} = X[0] + \frac{D_c}{2}$$

$$Ch_{y1} = \text{Centre y-coordinate of first child}$$

$$Ch_{y1} = C_y - \text{Radius} - \text{Gap}$$

where, Radius = user-defined radius of the circles
Gap = user-defined distance between two levels

Figure 7 shows the above mentioned parameter in the visualization.

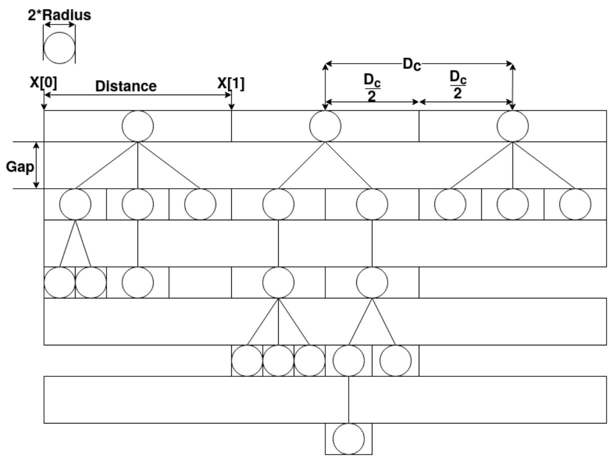

Figure 7. Parameter For Forest Visualization.

For the consecutive child, Chy will remain the same.

$$Ch_{y2} = Ch_{y1}$$

$$Ch_{x2} = Ch_{x1} + D_c$$

To draw the lines connecting the parent to the child, the coordinates of the parent as well as the child are known. Thus, applying the traversal technique already known, the entire graph can be constructed by computing the coordinates as mentioned above. The pseudocode for the visualisation is explained in Algorithm 8.

6.2. Simulink Application

Simulink is an application widely used for complex system design in mechanical or electrical systems as it has real time simulation possible with provisions for signal visualization and bug detection in the early stages of development. Many industries like the aerospace, automotive and industrial design trust and use Simulink. The Simulink models go through extensive phases of testing like unit testing, model-in-the-loop (MIL), and software-in-the-loop (SIL) testing. It might be helpful to visualize and check the dependencies of the variables in the model for bug detection. In some cases, logic replacements or other model tweaks become easier by understanding the dependency of affected variables and parameters. For these and many other automation requirements, traversing the Simulink model as a tree might be helpful.

In this section, we would discuss in detail about the extraction of the dependent tree for a variable which would be very helpful in testing. We can treat the Simulink model as a forest data structure and traverse it with any of the described traversal techniques. To convert the Simulink model to our data structure representation, we treat each block as a node. Using the 'get' command in Matlab, we can extract all the information associated with a block and given a block handle, we can access any block in the model. Thus, we do not need to separately store the node info in our data structure. We can use it simultaneously in this case. However, some workaround would be necessary in case a subsystem or looping is encountered which we have discussed below.

Algorithm 8 Forest Visualisation

```
InputTree                                        ▷ Input Tree Values, Number of Children Array
Gap = 30                                         ▷ Gap between one level and the next
                                                 ▷ Canvas Size
SizeX = 700
SizeY = 400
setup(SizeX,SizeY)
R = 10                                           ▷ Radius of the Node
                                                 ▷ Function draws the Nodes of the forest.
procedure DrawNode(X,Height,N,Cx,Cy,NotFirstNode)
    global XCordinate                            ▷ Store X[0],X[1]
    global j                                     ▷ Index Tracking of Child Nodes
    if N! = 0 then
        Distance = X[1]-X[0]
        BlockDistance = Distance/N
        XCenter = X[0] + (BlockDistance/2)
        XStart = X[0]
        XEnd = X[0] + BlockDistance
        for $i \rightarrow 0$ to $N$ do
            j = j + 1
            draw('Line',Cx,Cy,XCenter,Height + Radius);
            XCordinate[j] =
                [[XStart,XEnd],XCenter,Height-Radius]
            XStart = XEnd
            XEnd = XEnd + BlockDistance
            XCenter = XCenter + BlockDistance
            draw('Circle',XCenter,Height-R,R)
            if NotFirstNode then
                draw('Line',Cx,Cy,XCenter,Height + R);
                                                 ▷ Line between parent and child
            end if
        end for
    end if
                                                 ▷ Drawing the Root Nodes
    Height = SizeY/2
    XCordinate[0] = [[SizeX/2*-1,SizeX/2],0,SizeY/2]
    j = 0
    N = InputTree[i]
    DrawNode(XCordinate[0][0],XCordinate[0][2]-Radius,N,XCordinate[0][1],XCordinate[0][2],0)
                                                 ▷ Iterate through all the nodes to draw their children
    for $i \rightarrow 0$ to $length(InputTree)$ do
        N = InputTree[i]
        DrawNode(XCordinate[i][0],XCordinate[i][2]-Radius-
Gap,N,XCordinate[i][1],XCordinate[i][2],1)
    end for
end procedure
```

6.2.1. Subsystem

For a subsystem, we can maintain a separate array which contains the information of the blocks inside the subsystem. When we encounter the subsystem while traversal, we treat the logic inside the subsystem to be at the same Simulink hierarchical level as that of the parent subsystem. To understand this let us consider Figure 8. Let us treat output1 as the root. Output1 is connected to a subsystem. So, we go inside the subsystem and check which block is connected to Out1 (in turn connected to Output1). The next node thus becomes the AND block which is stored in the Info array. Additionally, we store the subsystem level in another array. Now, we treat the AND block as a parent node, and again go inside the subsystem by retrieving the level information from the subsystem array.

It is left to the user to decide if they want the logic of the subsystem to be included or treat the entire subsystem as a node. We can represent a Simulink model including a subsystem as a tree as shown in Figure 8.

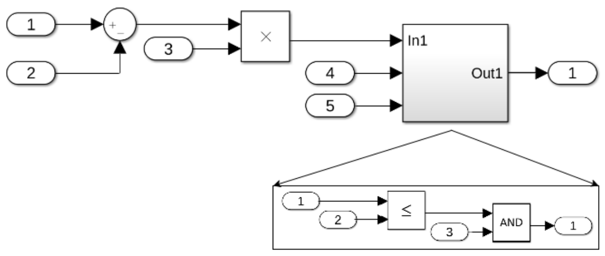

Figure 8. Simulink SubSystem.

6.2.2. Looping

The presence of a loop, makes the tree a cyclic graph which will abort further traversal. There are two ways to resolve this. First, we need to understand the fact that if looping happens in Simulink then a delay must be present. The first method involves checking for the presence of the preceding node in the existing array with delay and breaking the loop on occurrence. The second option is to check for the occurrence of the current node already in the data structure while traversing. If it exists, we can break the loop by terminating the node and assigning it a negative index in the NoOfChildren array as shown in Figure 9. This will help us to retain the link.

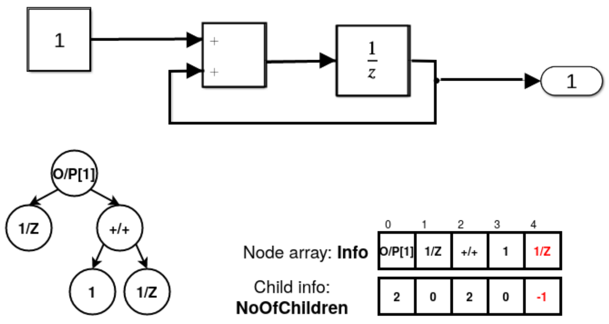

Figure 9. Simulink Looping.

The user can decide the type of structure (i.e., with/without subsystem information) they want to use for their application. For models with multiple inputs and outputs, if we are trying to analyse the errors due to specific blocks (like interpolation or filter), we can use this approach that will help us visualize the entire dependency chart and bring us a lot of information for testing techniques like MIL or SIL.

6.3. *Xml, Html*

Languages like XML, HTML operate on tags. Many softwares and applications like Polyspace, Simulink, Reactis store data in the backend in XML format. Data retrieval and information gathering from these XMLs can be easily achieved by treating it as the proposed tree data structure and traversing it using any of the methods described above. Using this approach, a particular branch or parent condition can be located easily and much faster. Similar approach can also be extended to traverse HTML files. We would not be indulging in the details of the traversal but storing it in our data structure format will give the ease of operation and also provide for removal of conditional nodes by linearly targeting it in an array.

7. Compression and Memory Optimisation

The proposed data structure has quite less memory consumption compared to the traditional linked list approach that stores the data in each node along with the node links as pointers. Even though our storage space is less, we can still optimize the space. For the first array (Info), we can use standard compression techniques like LZW [14] or Huffman Coding [28]. For the second array (NoOfChildren), we can have a different compression algorithm as there would be a lot of zeros in this array. The proposed compression technique is described below. Let us consider the example shown in Figure 1.

$$NoOfChildren = [3,3,2,3,2,1,0,1,1,0,0,0,0,0,0,3,2,0,0,0,1,0,0]$$

If we define the array to be of data type int or uint8, then each element in the array wastes a lot of bits to store 1 or 2 bit values. As there are 23 elements, it would require 184 (23 $\times$ 8) bits = 23 bytes, whereas if we consider 2 bit storage, 46 (23 $\times$ 2) bits are sufficient. The value of the number of children for a few nodes can be considerably large. To make the algorithm generic, we first represent the numbers in binary with a minimum number of bits. Thus,

$$NoOfChildren = [11,11,10,11,10,1,0,1,1,0,0,0,0,0,0,11,10,0,0,0,1,0,0]$$

Now, we add 0 or 1 between each bit to indicate continue reading the next bit or stop reading, respectively, to derive the value for a node. For example, 3 $\rightarrow$ 11 is now represented as 1011. The sequence that we obtain for the array NoOfChildren is 1011 1011 1001 1011 1001 11 01 11 11 01 01 01 01 01 01 1011 1001 01 01 01 11 01 01. The total number of bits in the sequence is 60, which can be easily stored in 8 bytes. For the initial storage, we needed 23 bytes. So, the compression ratio is 8/23, i.e., 34.78%.

8. Scope and Possible Improvement

With all the benefits of the proposed data structure, there exists scope for improvement. By slightly modifying the storage of the proposed data structure, we lay the ground for future work.

The idea here is to store the trees in arr1 and arr2 sequentially instead of level-wise (current implementation of forest data structure). In arr2, we are first storing the number of trees followed by the length of each tree. Then as usual, we proceed with the no. of nodes of the trees consecutively as shown in Figure 10. This approach can have a reduction in computation, especially in cases where only exclusive trees from a forest are to be accessed or in operations like insertion, and deletion by only visiting the affected tree.

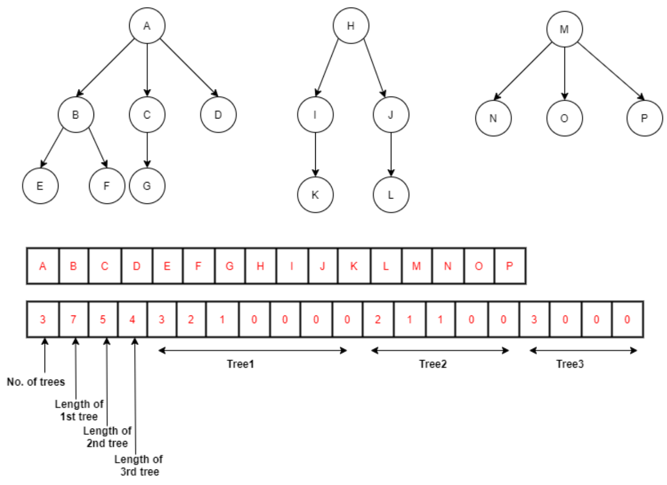

Figure 10. New Forest Representation.

9. Conclusions

The new proposed data structure has a large number of applications. With ever-evolving advancements, we can look towards modifications to the traditional data storage approach based on the needs of the application rather than modifying the entire algorithm. Many times tweaking the existing storage makes it easier and more robust than designing the algorithm all over again.

As the paper aims to solve a fundamental problem, many other algorithms based on it can also use, modify and be tweaked according to the use cases. The use cases can be cross domain, for example, decision trees or an AI algorithm (random forest) that expands its scope to all the applications that proceed it. Furthermore, Section 8 which mentions future improvement in the storage structure itself can also be explored in much detail. As the paper introduces novel data structure and algorithms, a lot more variations of this can open up and lead to more research work on data-structures and other subsequent domains.

Author Contributions: Conceptualization, G.B., S.K. and V.P.; Data curation, A.M.D.; Formal analysis, G.B., S.K., V.P. and A.M.D.; Funding acquisition, A.M.D.; Investigation, G.B., S.K. and V.P.; Methodology, S.K.; Project administration, A.M.D.; Software, G.B. and V.P.; Supervision, A.M.D.; Validation, S.K. and A.M.D.; Visualization, G.B.; Writing—original draft, G.B., S.K. and V.P.; Writing—review & editing, G.B., S.K., V.P. and A.M.D. All authors have read and agreed to the published version of the manuscript.

Funding: This research was partially funded by University Transilvania of Brașov.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Aho, A.V.; Hopcroft, J.E. *The Design and Analysis of Computer Algorithms*, 1st ed.; Addison-Wesley Longman Publishing Co., Inc.: Reading, MA, USA, 1974.
2. Cormen, T.H.; Leiserson, C.E.; Rivest, R.L.; Stein, C. *Introduction to Algorithms*, 3rd ed.; The MIT Press: Cambridge, MA, USA, 2009.
3. Beak, S.; Van Hieu, B.; Park, G.; Lee, K.; Jeong, T. A new binary tree algorithm implementation with Huffman decoder on FPGA. In Proceedings of the 2010 Digest of Technical Papers International Conference on Consumer Electronics (ICCE), Las Vegas, NV, USA, 9–13 January 2010; pp. 437–438. [CrossRef]
4. Al-Rawi, A.; Lansari, A.; Bouslama, F. A new non-recursive algorithm for binary search tree traversal. In Proceedings of the 10th IEEE International Conference on Electronics, Circuits and Systems, Sharjah, UAE, 14–17 December 2003; Volume 2, pp. 770–773. [CrossRef]
5. Borovskiy, V.; Müller, J.; Schapranow, M.; Zeier, A. Binary search tree visualization algorithm. In Proceedings of the 2009 16th International Conference on Industrial Engineering and Engineering Management, Beijing, China, 21–23 October 2009; pp. 108–112.
6. Navarro, G.; Sadakane, K. Fully-Functional Static and Dynamic Succinct Trees. *ACM Trans. Algorithms* **2009**, *10*, 1–39. [CrossRef]
7. Geary, R.F.; Rahman, N.; Raman, R.; Raman, V. A Simple Optimal Representation for Balanced Parentheses. In *Combinatorial Pattern Matching*; Sahinalp, S.C., Muthukrishnan, S., Dogrusoz, U., Eds.; Springer: Berlin/Heidelberg, Germany, 2004; pp. 159–172.
8. Benoit, D.; Demaine, E.; Munro, J.; Raman, R.; Raman, V.; Satti, S.R. Representing Trees of Higher Degree. *Algorithmica* **2005**, *43*, 275–292. [CrossRef]
9. Ferrada, H.; Navarro, G. Improved Range Minimum Queries. In Proceedings of the 2016 Data Compression Conference (DCC), Snowbird, UT, USA, 30 March–1 April 2016; pp. 516–525.
10. Delpratt, O.; Rahman, N.; Raman, R. Engineering the LOUDS Succinct Tree Representation. In *Experimental Algorithms*; Àlvarez, C., Serna, M., Eds.; Springer: Berlin/Heidelberg, Germany, 2006; pp. 134–145.
11. Berztiss, A.T. *Data Structures: Theory and Practice*, 2nd ed.; Academic Press, Inc.: New York, NY, USA, 1975.
12. Deaconu, A. Iterative Algorithm for Construction of a Tree from its Pre-order and Post-order Traversals in Linear Time and Space. *Sci. Ann. Cuza Univ.* **2004**, *14*, 69–80.
13. Shenfeng, C.; Reif, J.H. Efficient lossless compression of trees and graphs. In Proceedings of the Data Compression Conference, Snowbird, UT, USA, 31 March–3 April 1996; p. 428. [CrossRef]
14. Welch. A Technique for High-Performance Data Compression. *Computer* **1984**, *17*, 8–19. [CrossRef]

15. Andrusky, K.; Curial, S.; Amaral, J.N. Tree-Traversal Orientation Analysis. In *Languages and Compilers for Parallel Computing*; Almási, G., Caşcaval, C., Wu, P., Eds.; Springer: Berlin/Heidelberg, Germany, 2007; pp. 220–234.
16. Hu, Q.; Che, X.; Zhang, L.; Zhang, D.; Guo, M.; Yu, D. Rank Entropy-Based Decision Trees for Monotonic Classification. *IEEE Trans. Knowl. Data Eng.* **2012**, *24*, 2052–2064. [CrossRef]
17. Zhao, H.; Zhang, C. An online-learning-based evolutionary many-objective algorithm. *Inf. Sci.* **2020**, *509*, 1–21. [CrossRef]
18. Breiman, L. Random Forests. *Mach. Learn.* **2004**, *45*, 5–32. [CrossRef]
19. Landau, S.; Barthel, S. Recursive Partitioning. In *International Encyclopedia of Education*, 3rd ed.; Peterson, P., Baker, E., McGaw, B., Eds.; Elsevier: Oxford, UK, 2010; pp. 383–389. [CrossRef]
20. Binkley, D.; Gold, N.; Islam, S.; Krinke, J.; Yoo, S. A comparison of tree- and line-oriented observational slicing. *Empir. Softw. Eng.* **2019**, *24*, 3077–3113. [CrossRef]
21. Shrestha, S.L. Automatic Generation of Simulink Models to Find Bugs in a Cyber-Physical System Tool Chain using Deep Learning. In Proceedings of the 2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), Seoul, Korea, 5–11 October 2020; pp. 110–112.
22. Tajarrod, F.; Latif-Shabgahi, G. A Novel Methodology for Synthesis of Fault Trees from MATLAB-Simulink Model. *World Acad. Sci. Eng. Technol. Int. J. Comput. Electr. Autom. Control. Inf. Eng.* **2008**, *2*, 1756–1762.
23. D'Angelo, G.; Pilla, R.; Tascini, C.; Rampone, S. A proposal for distinguishing between bacterial and viral meningitis using genetic programming and decision trees. *Soft Comput.* **2019**, *23*, 11775–11791. [CrossRef]
24. Dulebenets, M.A.; Moses, R.; Ozguven, E.E.; Vanli, A. Minimizing Carbon Dioxide Emissions Due to Container Handling at Marine Container Terminals via Hybrid Evolutionary Algorithms. *IEEE Access* **2017**, *5*, 8131–8147. [CrossRef]
25. Kaufmann, K.; Vecchio, K.S. Searching for high entropy alloys: A machine learning approach. *Acta Mater.* **2020**, *198*, 178–222. [CrossRef]
26. Azad, M.; Chikalov, I.; Hussain, S.; Moshkov, M. Entropy-Based Greedy Algorithm for Decision Trees Using Hypotheses. *Entropy* **2021**, *23*, 808. [CrossRef] [PubMed]
27. Vlase, S.; Marin, M.; Deaconu, O. Vibration Properties of a Concrete Structure with Symmetries Used in Civil Engineering. *Symmetry* **2021**, *13*, 656. [CrossRef]
28. Huffman, D. A method for the construction of minimum-redundancy codes. *Resonance* **2006**, *11*, 91–99. [CrossRef]