Journal of Physical Sciences, Vol. 12, 2008, 253-264

A New Algorithm to Represent a Given *k*-ary Tree into Its Equivalent Binary Tree Structure

Sumit Kumar Ghosh *, ***Joydeb Ghosh*** [@] and ***Rajat Kumar Pal*** [#]

* Yahoo! Software Development India Pvt. Ltd., "Torrey Pines," Embassy Golf Links Business Park, Off Indirangar – Koramangala, Intermediate Ring Road, Bangalore - 560 071, India

[@] Department of Mathematics, Greater Calcutta College of Engineering and Management,
Vill.: Dudhnai, Ramnagar-2, P.O.: Baruipur, Dist.: 24 Parganas (South), West Bengal, India

[#] Department of Computer Science and Engineering, University of Calcutta, 92, Acharyya Prafulla Chandra Road, Kolkata – 700 009, India

Received November 20, 2008; accepted December 10, 2008

ABSTRACT

In this paper we have developed an algorithm that converts a given *k*-ary tree, for any $k \geq 3$, to its equivalent binary tree structure. The binary tree is generated in $O(n)$ time, for a *k*-ary tree with a total of *n* nodes. The algorithm is designed aiming at reducing the height of the constructed binary tree. The constructed tree does not contain any free links in the non-leaf nodes. That means the constructed tree is like a complete binary tree, where only leaves have no children, and nodes other than leaf nodes contain child (children) and some other valid information of the given *k*-ary tree.

Keywords: *Tree, Binary tree, k-ary tree, Tree conversion, Algorithm.*

1. Introduction

A binary tree is the simplest non-linear data structure. It gives us the same time complexity $O(\log_2 n)$ that is given by other *k*-ary trees, $O(\log_k n)$, for any $k \geq 3$. For a binary tree $k = 2$. For other trees *k* varies. However, $O(\log_2 n) \equiv O(\log_k n)$ for any constant $k \geq 3$! By the Big-Oh notation the trees are equivalent. So the performance of any *k*-ary tree with respect to Big-Oh is the same. Hence we like to develop an algorithm that converts a given *k*-ary tree to its equivalent binary tree structure.

From a different perspective of representing a tree structure using a computer, we can say that the binary tree representation is closer to the machine representation. This is best explained when we implement a binary tree using arrays. Let us assume that, in the array representation of a binary tree, we have the left child at $2*i$ position and the right child at $2*i+1$ position, where *i* is the address of the current node. Thus to lead us to a child we require a left shift followed by an addition, if necessary. Thus we can say that we can implement a binary tree in primary memory without much effort.

At the same time, if we represent a *k*-ary tree using an array, then it leads us to a sparse array, as in general, most of the nodes contain much less than *k* children. So, representation of any *k*-ary tree is usually a costly process as a huge memory space is left unused. However, the algorithm proposed in this paper is much more efficient in utilizing the computer memory for the representation of the given *k*-ary tree.

In this paper we have developed a new algorithm that converts a given *k*-ary tree (or a forest) to its equivalent binary tree such that any information in the given tree is not lost. We can say that property preservation of the existing tree is mandatory when it is converted to a binary tree. It is generated in such a way that neither any new restrictions are imposed on the tree nor any new undesirable property is added to it.

The rest of the paper is organized as follows. In Section 2, we make a brief survey on the existing literature. The proposed algorithm is included in Section 3, along with the necessary data structures. We briefly discuss the performance of the proposed algorithm in Section 4, and conclude the paper with a few remarks in Section 5.

2. Literature Survey

Natural correspondence between forests (or *k*-ary trees) and binary trees: As it is defined in [4], there is a natural way to represent any forest as a binary tree. The binary tree obtained here has a one-to-one correspondence with the original tree. However, after converting a single tree (other than a binary tree) to its equivalent binary tree structure, the computed binary tree's root node has no right subtree. The method of conversion is as follows [3, 4].

- Link the child nodes for a parent node that are at the same level.
- Then from the links of the original tree, the link from the parent to the first (or the leftmost) child is preserved and the subsequent links to the children are discarded.
- Then keeping the root as the center, the tree is rotated by 45º clockwise. The tree obtained is the desired binary tree.

To illustrate the method stated above, let us consider a *k*-ary tree as shown in Figure 1.

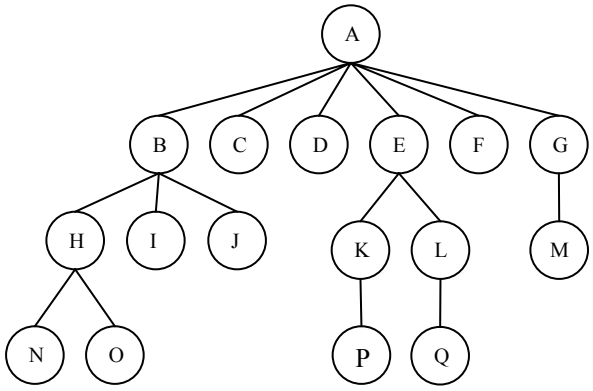


Figure 1. A sample *k*-ary tree, where k = 6.

In the first step we connect all the child nodes for a parent node that are in the same level as shown in Figure 2. The children of each family are linked together. As for example, B, C, D, E, F, and G are the children to A; hence, they belong to the same family. Similarly as H, I, and J belong to one family, and K and L to another. However, P and Q do not belong to the same family.

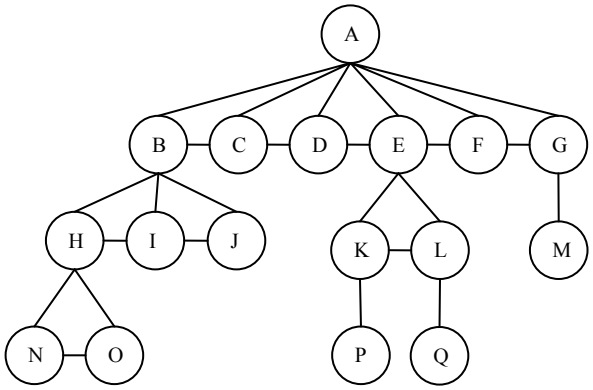


Figure 2. Connect all the child nodes at the same level for the same parent.

Now, from this representation shown in Figure 2, we remove the links from the vertical links except the ones that connect the parent to the first child. For example, B is the first child for A. Similarly H is the first child for B, and so on. The next obtained tree is shown in Figure 3.

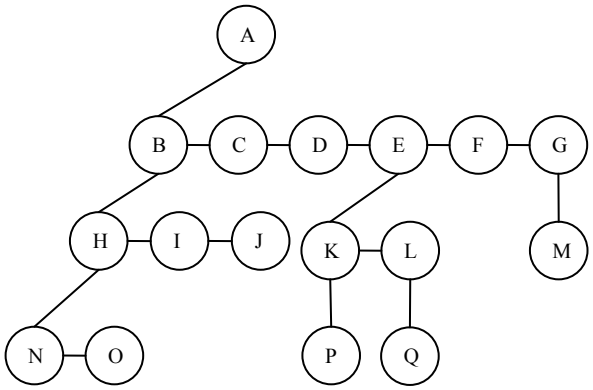
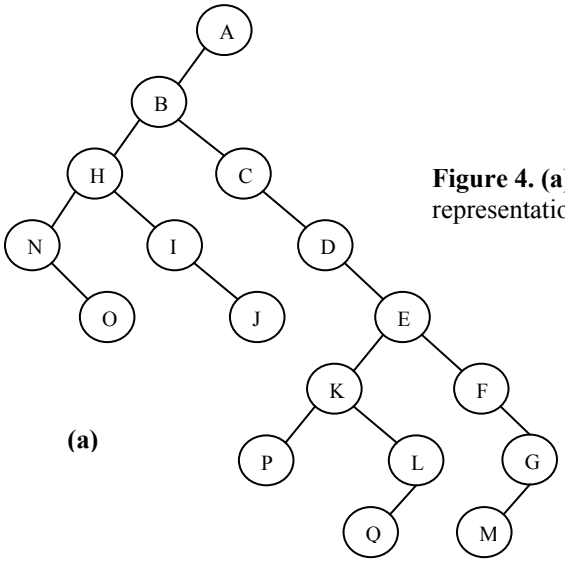


Figure 3. Preserve only the parent to the first child and the first child to its sibling relationships.

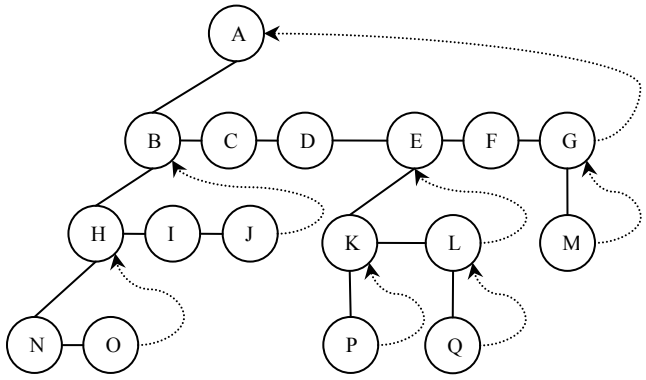
This tree is rotated keeping the root as the center clockwise, giving the desired tree as shown in Figure 4(a). In general, if two or more *k*-ary trees are considered in a forest for following the same algorithm, then the right child of the root of the final tree would contain the root of the second tree in the given forest.

Threaded binary trees: As defined in [3, 4] this is an extrapolation of the existing method using the empty links of the computed binary tree. The threading concept is similar to a threaded binary tree [1, 2, 3, 4]. Here the right thread of the rightmost child, of a family, goes to the parent; this has been shown in Figure 4(b). This may help in obtaining some desired sequence while traversing the given tree, at a lesser cost.



(a)

Figure 4. (a) The equivalent binary tree representation of the given *k*-ary tree in Figure 1.



(b)

Figure 4. (b) Threaded binary tree, as an extrapolation.

3. The Proposed Algorithm

Our aim is to convert a *k*-ary tree or a forest to a binary tree such that any information stored in the tree, is not lost. Alternatively, we can say that property preservation of the existing tree is mandatory when it is converted to a binary tree. It is generated in such a way that neither any new restrictions are imposed on the tree nor any new undesirable property is added to it. Now let us take a tree, other than

binary tree, for which we generate an equivalent binary tree structure. Let us consider the same *k*-ary tree as shown in Figure 1.

If we perform a *level order traversal* or *breadth first search (BFS)* procedure on the given tree, then we get the following sequence: A B C D E F G H I J K L M N O P Q. In this order we generate a tree in which all the nodes are stored serially, however, as soon as the parent (of a node in the given tree) changes the next available link of the node in succession is used as a thread to the new parent node.

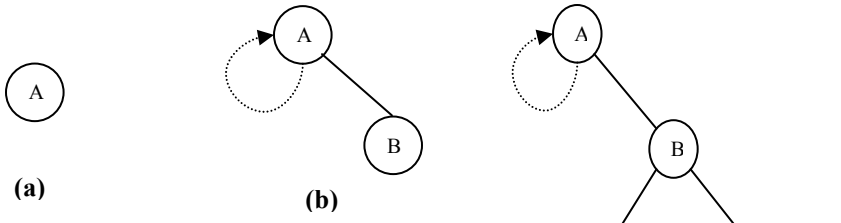


Figure 5. (a) The tree obtained after processing node A. **(b)** At the time of processing node B a thread is generated as A is the parent of B in the given tree. **(c)** Nodes C through G are inserted into the tree in the form of a (complete) binary tree structure without inserting any thread as parent does not change, for each of these nodes

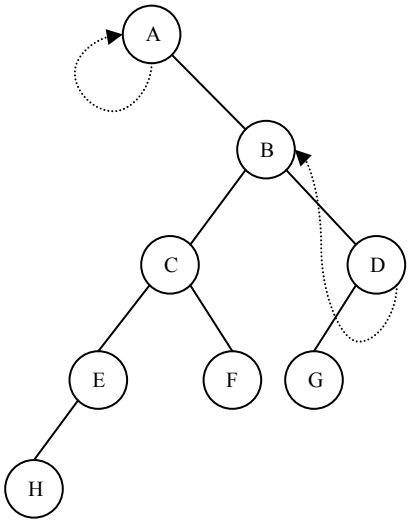


Figure 5. (d) Parent changes when node H is under consideration to insert into the new tree; so, a thread is introduced from the immediate nil link of node D to B as H is the leftmost child of B.

Before glancing through the algorithm, let us illustrate this procedure on a tree shown in Figure 1.

- On processing A, the root node of the given tree, we have only node A; see Figure 5(a).

- When we move to B, which is the first child of A, the parent changes; so there is a thread generated from the left link position of A to point to A. At the same time B is linked as the right child of A; see Figure 5(b).
- Now for the nodes C through G the parent remains the same (as it is for node B). So, no thread is inserted and the tree under construction grows naturally in the form of a (complete) binary tree structure, as shown in Figure 5(c).
- When H is considered for its insertion into the constructed tree, the parent changes from A to B; see Figure 1. Before inserting H into the tree, a thread is introduced from the immediate next available free link (i.e., the right link of D) to B (see Figure 5(d)), as H is the leftmost child of B in the given *k*-ary tree. This is how we preserve the parent-child relationship among the nodes in the given *k*-ary tree.
- The same procedure is followed for inserting nodes up to J into the new tree. Here it is worthwhile to note that C and D have no child, so there are no threads from any node of the constructed tree to any of them. While processing nodes K and L, again a new parent is found, which is E. So, before inserting K and L into the new tree, a thread is introduced, from the immediate next free link, which is available as the right child of F, to point to E (i.e., the parent of K and L). Figure 5(e) shows it as the constructed tree up to the insertion of node L.

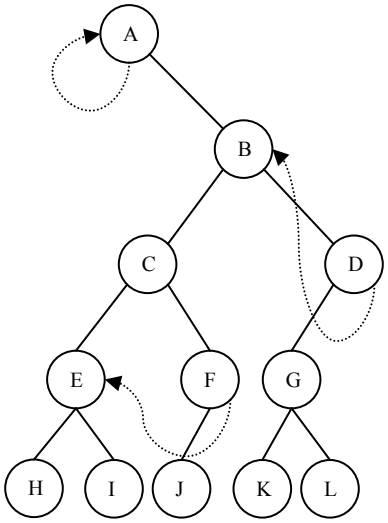


Figure 5. (e) Nodes considered for their insertion into the new tree up to node L. C and D have no child; hence no thread is pointing to any of them.

- The finally constructed tree for the given *k*-ary tree (in Figure 1) using the proposed algorithm in this paper, is shown in Figure 5(f).

Now, if we have a forest (see Figure 6) then the corresponding binary tree is what we see in Figure 7(a) obtained by using the proposed algorithm, whereas Figure 7(b) shows the binary tree obtained using the existing algorithm [4].

Assumption:

- All the children of a node in the given *k*-ary tree are assumed to be ordered from left to right as siblings, and no nil links (or null branches) are there in between any pair of siblings. This helps us in uniquely regenerating the *k*-ary tree as and when required.

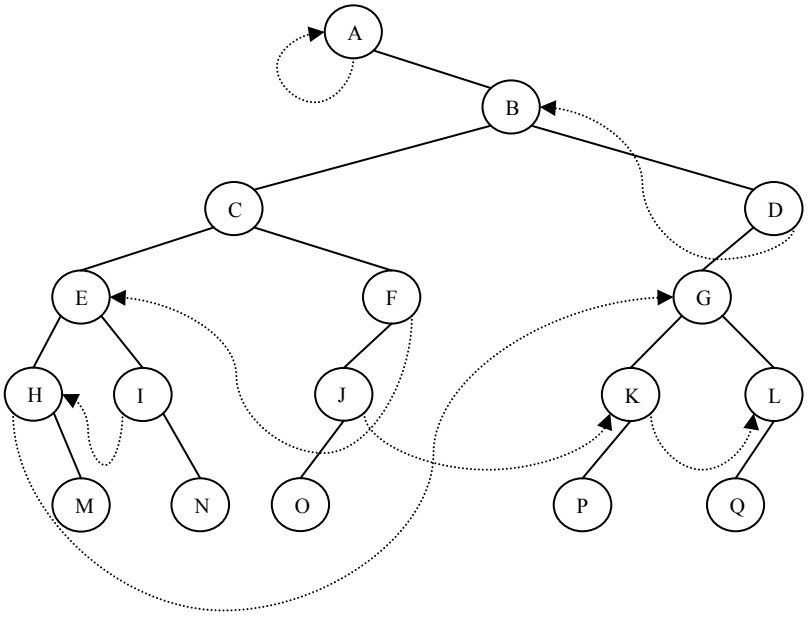


Figure 5. (f) The final binary tree computed using the algorithm proposed in this paper.

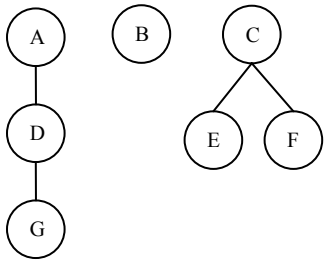


Figure 6. A sample forest.

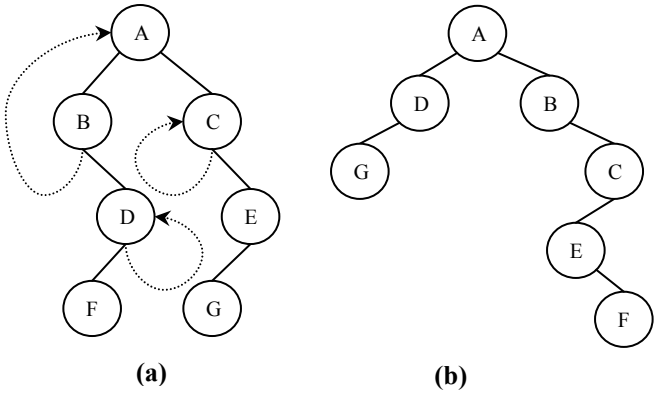


Figure 7. (a) Binary tree generated by the new algorithm. **(b)** Binary tree generated by natural correspondence (i.e., the existing algorithm).

Algorithm at a glance:
Input: A *k*-ary tree.
Output: An equivalent binary tree structure.

1. [Initialization] Insert the root node(s) in a queue and in the binary tree.
2. [Form the structure] Pick a node from the queue.
   If it has no child then
   a. [Discard it] Jump to Step 3 {If a node has no child nodes then we do not need any thread for that node. When we keep threads then a sequence of threads one after the other is obtained. This is of no use for regeneration of the tree. Furthermore, this is a redundant information to process.}
   Else {It has a child}
   a. [Initiate the family] Assign the "next in sequence" pointer to the parent node {picked from the queue} and make it a thread. {Set the thread to identify the parent.}
   b. [Process the family] Assign all the children to the binary tree in sequence using the "next in sequence" pointer; also add them to the queue in sequence.
3. [Loop/Exit] Is the queue empty? If yes then stop processing, else perform Step 2 onwards.

The algorithm that is stated above at a glance considers a data structure queue and with the help of this data structure it computes the desired binary tree structure of a given *k*-ary tree. A thread is introduced in the next available link of the binary tree structure under construction when the parent changes and the children of the new parent are to be introduced in the binary tree. When the queue is updated by inserting a new node of the given *k*-ary tree, the "next in sequence" pointer information is also updated by inserting two link fields (left link and right link) of the node; as soon as a new parent of a child in the given *k*-ary tree is under consideration, the "next in sequence" pointer is linked to that parent node in the constructed binary tree structure with the help of a thread. Subsequently, the family of that parent node is also updated using the available "next in sequence" pointer in

the form of the desired binary tree structure. The necessary data structures used are explained as follows.

Data structures:

- ***k*-ary tree**
  1. It is created using a structure in which we have a node and *k* links; see Figure 8(a).
  2. In case of a forest the root nodes are set up in an initial array.

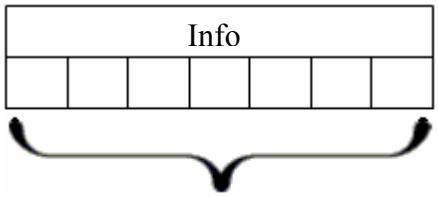


Figure 8. (a) *K*-ary tree structure.

- **Binary tree structure**

  It is a threaded binary tree, in which we have an info part and two links as left child and right child. Along with this we have two 1 bit variables left thread and right thread, which are used as indicators. A left thread = 1 indicates that the left child is a thread and it points to a parent node, otherwise, if it is 0 then the left child is a pointer to a child node; see Figure 8(b).

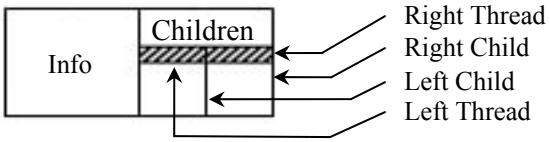


Figure 8. (b) Binary tree structure.

- **The queue**

  It is a FIFO list in which each entry comprises of two pointers (see Figure 8(c)).
  - A *k*-ary tree pointer: It is used to obtain the subsequent *k*-ary tree node to be processed from the queue.
  - A binary tree pointer: It is used to obtain the binary tree node corresponding to the *k*-ary tree node.

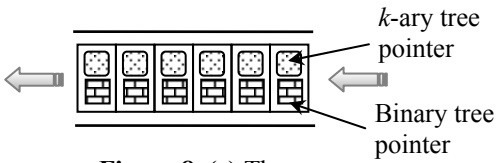


Figure 8. (c) The queue.

- **The "next in sequence" pointer {NIS( )}**

    - It is a FIFO list that is a queue of binary tree pointers; see Figure 8(d).

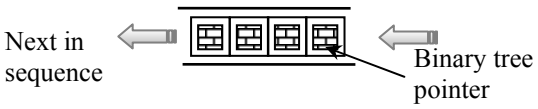


Figure 8. (d) "Next in sequence" pointer.

    - Related functions for the "next in sequence" pointer:
        - NIS(ptr): When this function is called it adds a binary tree pointer to the queue rear. It is called twice, every time a node is introduced into the binary tree under construction for inserting the left and the right links of the node in succession that may be used as a thread or a link.
        - NIS( ): It returns a binary tree pointer from the front of queue. It is called before a node is added to the binary tree to get the position of insertion.

Improvement on the data structures:

The queue and the "next in sequence" pointer store the same data at some point of time. If we can merge it we can save some space on redundant information being stored at the cost of a few flag variables. Here only the binary tree pointer is used. The "next in sequence" pointer is to be the left link of the node pointed by the binary tree pointer and subsequently the right link of that node. The pointers, the binary tree pointer and the *k*-ary tree pointer, remain same as described above.

The queue is modified a bit to handle the improvement. It has two indicators:

- One for node being processed in the *k*-ary tree. The *k*-ary tree flag.
- Second the node being used in the binary tree. The binary tree flag.

The *k*-ary tree flag:

It is a 1-bit variable which shows that the node is visited or not. A 1 indicates that the node is processed while a 0 shows it is not. When a node is first inserted in this queue (as a child or as the root) this flag is set to 0. Once we visit its children we set the flag to 1.

The binary tree flag:

It is a 2-bit variable that shows the binary tree, – whether the node's right and left child being empty (or not).

- A value 0 shows both child being unused.
- A value 1 shows left child is used and right child being unused.
- A value 2 shows both children being used.

A node can be deleted from the front of the queue when the *k*-ary tree flag = 1 and the binary tree flag = 2.

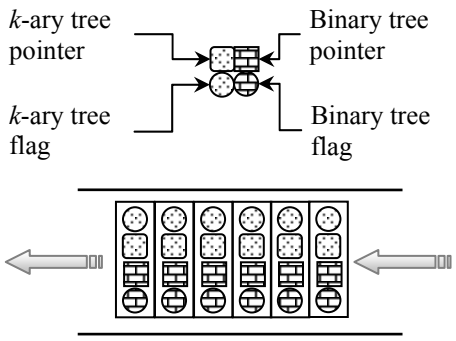


Figure 9. The improved queue.

4. Performance of the Algorithm

Advantages over the existing method:

- Simpler to implement.
- **No wastage:** Produces a binary tree wasting no links. Issue of a tree being skewed is resolved. However, when the conversion is done for a single tree the tree is right heavy. When we handle only one tree then it is evident that the left child of the root is a thread. Hence it can be discarded, and the child node for the root starts without any thread. However, if the scenario is a mixed one (when either a tree or a forest is given), then to avoid this skewed nature we can use an extra global flag that differentiates whether the given structure is a tree or a forest (only for the root node of the constructed binary tree structure). In case of a tree there are no threads to the root, however, in case of a forest the structure remains uniform throughout, and implicitly the skewed nature is avoided.
- **Tree Levels:** We assume that the number of levels of the existing *k*-ary tree is *L*. The depth of the binary tree generated by the existing method is described below.
  a. Best case: $L + k - 1$
  b. Worst case: $(L - 1) * k + 1$

  The depth of the binary tree generated by the proposed algorithm in this paper is $\log_2$(number of nodes in the given *k*-ary tree + number of non-leaf nodes in the given *k*-ary tree).

Comparison of the existing method to our algorithm:

- **Running time:** Both the methods need to do a level order traversal or a BFS to generate the computed binary tree. Hence both run on O(*n*) time, where *n* is the number of nodes of the given *k*-ary tree.
- **Structural information:** Preserves structural information of the original tree. Hence the original tree can be obtained from the derived tree. So, we can say that there is a one-to-one mapping of the binary tree obtained with the original tree.

- **Space required:** In both the cases the trees have the same number of nodes as in the original tree, hence the space required for the nodes and links in both the cases remain same. However, the threaded binary tree requires two bytes extra in both the representations. Hence the overall space requirement increases linearly but this is true even for the existing method's threaded approach; see Figure 4(b).
- **Traversals:** If the existing tree traversal algorithms like Inorder, Preorder, Postorder, DFS, and BFS algorithm [1, 2, 3, 4] cannot be applied to the tree formed using the new process; however, these algorithms can be directly applied to the binary tree using the existing process. The existing tree traversal algorithms, if they need to understand the parent child relationship of the original tree, then they also require some modification. However, this is equally true for the existing approach of converting a given *k*-ary tree to its equivalent binary tree structure.
- **Accessing the child nodes:** The existing approach can locate a child of a parent node in O(1) time. The tree generated using the proposed algorithm can also do it in constant time. To reach a child node from a parent node we need to sum up the number of child nodes from the root node as we come down to the parent node. Nodes those are visited can be subtracted from this number to get the exact position of the child node.

5. Conclusion

Binary tree is the simplest non-linear data structure. Other than binary tree data structures *k*-ary tree structures are as good as binary tree structures in terms of any sort of computation involving this kind of trees, but they use more memory space. In this paper we have proposed a new algorithm for converting a given *k*-ary tree into its equivalent binary tree that takes time linear to the number of nodes belonging to the given tree. This newly constructed tree is more balanced like a complete binary tree by the way how it is being constructed. This tree is also having the height less than the height of the equivalent binary tree computed using the existing method. This is obtained by using all links that are available in all non-leaf nodes.

REFERENCES

1. Cormen T. H., C. E. Leiserson, R. L. Rivest and C. Stein, *Introduction to Algorithms*, Second Edition, Prentice Hall of India Pvt. Ltd., New Delhi, 2003.
2. Deo N., *Graph Theory with Application to Engineering and Computer Science*, Prentice Hall of India Pvt. Ltd., New Delhi, 2000.
3. Horowitz E., S. Sahni and S. Rajasekaran, *Computer Algorithms*, Galgotia Publications Pvt. Ltd., New Delhi, 2004.
4. Knuth D. E., *The Art of Computer Programming (Volume 1): Fundamental Algorithms*, Third Edition, Addition Wesley: An Imprint of Pearson Education, Delhi, 2002.