Journal of Computer Technology and Application 3 (2012) 374-382

Recursive and Nonrecursive Traversal Algorithms for Dynamically Created Binary Trees

Robert Logozar

Polytechnic of Varazdin, HR-42000 Varazdin, Croatia

Received: April 17, 2012 / Accepted: April 27, 2011 / Published: May 25, 2012.

Abstract: The modeling of dynamical systems from a time series implemented by our DSA program introduces binary trees of height D with all leaves on the same level, and the related subtrees of height $L \leq D$. These are called ϵ-trees and ϵ-subtrees. The recursive and nonrecursive versions of the traversal algorithms for the trees with dynamically created nodes are discussed. The original nonrecursive algorithms that return the pointer to the next node in preorder, inorder and postorder traversals are presented. The space-time complexity analysis shows, and the execution time measurements confirm, that for these $O(2^D)$ algorithms the recursive versions have approximately 10-25% better time constants. Still, the use of nonrecursive algorithms may be more appropriate in several occasions.

Keywords: binary ϵ-trees, algorithms, tree traversal, preorder, inorder, postorder, recursive, nonrecursive, space-time complexity

1. Introduction

The choice and comparison of recursive versus nonrecursive algorithms is a known subject from the algorithm-study in computer science. It is found in the major textbooks, it is investigated by scientists and discussed by professionals. In this paper, we present the design and complexity analysis of a few testing and practical functions that do their job on dynamically created tree nodes by using both recursive and nonrecursive tree traversal algorithms.

The implementation of the algorithms and their testing is done within our DSA (Dynamical Systems Automata) program for the modeling of dynamical systems from a time series, based on J. P. Crutchfield's theory of *ϵ-machines* [1]. In [2] we have presented the outline and some interesting aspects of the theory for the readers from the computing area. The resulting dynamical system models are based on the stochastic finite automata, which are analogous to the finite automata from the theory of computation. As such, the modeling scheme turns out to be an intriguing programming challenge from the perspective of computer science. Quite general structural and algorithmic problems were addressed during the development of the DSA modeling tool, which are interesting for their design analysis and efficiency testing. From quite a few of them, here we concentrate on the tree structures and the algorithms that traverse through and operate on their nodes.

The short outline of the paper is as follows: in section 2 we start with the definition of the binary ϵ-trees and subtrees that are used in our program. In a full formal approach we contrast their properties to the general binary trees. Section 3 is devoted to the comparison of recursive and nonrecursive traversal algorithms and the requests that their different possible realizations impose on the necessary node organizational data. Section 4 deals with the implementation of the algorithms, giving examples of their concrete C++ code listings realized through the object-oriented paradigm. The original preorder traversal algorithm is explained in details and proved for correctness. In section 5 the space-time complexity analysis is given for both sorts of algorithms, which is then confirmed by the execution time measurements in section 6. After the concluding words of section 7, appendices provide the solutions for the remaining two original nonrecursive traversal algorithms, and outline two other interesting pieces of the program code.

Corresponding author: Robert Logozar, M.Sc., research fields: modeling of dynamical systems, algorithms and data structures. E-mail: robert.logozar@velv.hr.

2. ϵ-Trees and ϵ-Subtrees

Tree is a well-known data structure that can be defined in both, recursive [3, 4, 5] and nonrecursive [6] way. We start with the ***general binary tree***, whose nonrecursive definition can be summarized as follows:

Def. 1. A *binary tree* T is a finite set of nodes that is either empty or consists of a root node and two disjoint *binary trees* called the *left* and *right subtrees*.

It is assumed that all the terms that accompany the tree data structures are defined in the usual and natural way. We shall emphasize that *root* is the node with no incoming edges (no parent node), on the level $l = 0$. Also, a *leaf* of a general binary tree is a node with no descendants (children), and the maximal leaf level is the *tree height* or *tree depth*.

Now we state the following specialization for the (sub)trees that are used in the ϵ*-machines* modeling:

Def. 1E. A (*binary*) ϵ*-tree* of height (depth) D, $D = 0, 1, 2, ...$ is a (binary) tree with all its leaves on the same level $l = D$. The obvious consequence is:

Corollary 1E.1. In a (binary) ϵ- tree of height (depth) D every node on level $l = D$ is a leaf.

Corollary 1E.2. A binary ϵ-tree of height D has minimally 1 and maximally 2^D leaves. Proof follows trivially from the def. of binary (ϵ-)trees, with maximally 2^l nodes on the level l, $l = 0, 1, 2, ... D$.

Corollary 1E.3. In a binary ϵ-tree of height D, there are minimally $D + 1$ and maximally $2^{D+1} - 1$ nodes. *Proof.* We use Cor. 1E2. The minimal, 1-leaf binary ϵ-tree has $D + 1$ nodes by definition. The *full* (*complete*) ϵ -tree, with 2^D leaves, has $N_{Nds,max} = \sum_{l=0}^{D} 2^l = 2^{D+1} - 1$ nodes, as can be easily proven by induction.

Theorem 1E.1. There are $2^{2^D} - 1$ different binary ϵ-trees of height D. *Proof.* From cor. 1E2 there are maximally 2^D leaves. The nonexistence (0) or existence (1) of each of the leaves define an ϵ- tree. So we need the number of the ordered 2^D-tuples of bits, which is 2^{2^D}, less one for the 2^D-tuple with no leaves at all. QED.

Def. 1ES. Let T_ϵ be an ϵ-tree of height (depth) D, $D = 0, 1, 2, ...,$ that we shall call *supertree*, or more operationally *main tree*. Then some S_ϵ binary ϵ-tree is a binary ϵ*-subtree* of T_ϵ if all its nodes are within the main ϵ-tree. S_ϵ is fully defined by its root, which is some node from the main ϵ-tree on level l_{StR}, and its height (depth) $L \geq 0$.

Note. ϵ-subtree is conceptually different from the subtree in Def. 1, whose leaves always coincide with the leaves of its binary supertree. In that sense the ϵ-subtrees are more general. In our modeling they present the (sub)structure within the structure of ϵ-super or main tree, as was just defined in Def. 1ES.

Corollary 1ES.1. From Def. 1ES we immediately conclude: $0 \leq L \leq D$, $l_{StR} \leq D - L$.

Corollary 1ES.2. Let S_ϵ be an ϵ-subtree of height L with the root that is a node on a level l_{StR} of some main (super) ϵ-tree T_ϵ of height D. Let l_{Nd} be the level of an ϵ-subtree node relative to the main tree root, $0 \leq l_{Nd} \leq D$, which will be called the *absolute lev*el. Then the node relative l'_{Nd} level within the ϵ-subtree is $l'_{Nd} = l_{Nd} - l_{StR}$, $0 \leq l'_{Nd} \leq L$.

Corollary 1ES.3. From 1ES.2 it immediately follows that the ϵ-subtree root is at $l'_{StR} = 0$ level because for this node the absolute level is $l_{Nd} = l_{StR}$. Also the ϵ-subtree leaves are at the relative level $l'_{Lf} = L$, because $l_{Lf} = L + l_{StR}$. Following the Def. 1E, this makes the ϵ-subtree a regular ϵ-tree of height L.

Corollary 1ES.4. The smallest ϵ-subtree is that of the height $= 0$, with its single, root node placed in any of the nodes of its main tree, and the largest ϵ-subtree is equal to the main tree itself, $l_{StR} = 0$, $L = D$.

Corollary 1ES.5. The concept of ϵ-subtrees from Def. 1ES cannot be applied to the general binary trees from Def. 1, because only the ϵ-trees, from the Def. 1E, have the additional criterion for their leaves.

The reason for organizing ϵ-subtrees within some main ϵ-tree can be to investigate the inner structure of the latter. For that, only the nodes of the main ϵ-tree must be actually stored, because the main ϵ-tree encompasses all the nodes from its ϵ-subtrees. The ϵ-(sub)trees can have further ϵ-subtrees of their own.

In this way one can investigate the nature of a (binary) time series from which the (binary) main ϵ-tree

is formed. Only unique ϵ-subtrees are of interest—those that are different from each other. Such ϵ-subtrees are shown to be the causal states of the dynamical system that emits such a time series [2].

3. Binary Tree Traversals

The tree traversal is a process in which every tree node is visited once and only once. The aim of traversal is to perform some action on each node, or with respect to each node that is visited, as when counting, or selecting them.

Def 2A. For the binary trees the following three types of traversals are usually defined:

- *preorder* – starting from the root the parent node is visited first, then the left child if it exists, and after that the right child (the attributes left and right can be interchanged here and in other two traversals);
- *inorder* – the farthest left child is visited if it exists, then the parent node, and after that the right child;
- *postorder* – the farthest left child is visited if it exists, then its right sibling if it exists, and after that their common parent node.

In each of the traversals, the "boundary nodes" that are always checked in some way are afore defined: (a) leaves from which there is no way down, and (b) the root from which there is no way up the tree.

Each of the traversal types has its specifics and (dis)advantages that depend on the application or the form of the tree. Sometimes all three traversal orders can be used, and sometimes one is more efficient than the others. A good example for that is the deletion of the dynamically created nodes when the postorder traversal is the most logical and the simplest choice.

The minimal information that must be stored in a binary tree node is two *indicators* (indices, pointers) to its left and right children. If both of these are NULL, the node in a general binary tree is a leaf. Recursive algorithms will implicitly recognize the root level as the end of the call-stack unwinding process. For nonrecursive algorithms, an extra stack must be organized to enable tracking of the (un)visited nodes.

Alternatively, the additional indicator about the parent can be stored in every node. For general trees the root is the only node with null indicator to parent. This additional datum also enables the construction of nonrecursive algorithms without the supplementary stack.

Furthermore, since our intention is to apply the traversal algorithms on the ϵ-(sub)trees, too, the Corollary 1ES.3 gives us the criterion for checking if the ϵ-tree nodes are leaves. To apply it effectively, we would want to have the node absolute level stored in each node, and thus enable straightforward calculation of the node relative level.

The above deliberation is summarized as follows:

i. For the binary trees with nodes that hold the minimal structural information, that is, only the indicators to their left and right children, the recursive algorithms can be used straightforwardly. The nonrecursive algorithms would require a supplementary stack for recording the traversal return path (confer [6] ch. 1, [3] ch.2). Also, the implementation of ϵ-subtrees according to Def. 1ES would in this case demand additional efforts to find out the relative node levels in order to check the leaf criterion.

ii. If an indicator to parent is added to every node, we can design the nonrecursive algorithms without having to organize the extra stack. The cost of one more indicator per node (the rise of 50% comparing to the case (i)), can be justified with the benefits of easier navigation up the tree. Again, the implementation of the traversals for ϵ-subtrees would require separate calculation of the node levels, resulting in an additional time cost.

iii. Furthermore, by adding the node absolute level (with respect to the main tree that encompasses all other subtrees) to each node, the recursive and nonrecursive traversal algorithms can be efficiently implemented also to the binary ϵ-subtrees (Def. 1ES), without the need for additional data structures and much overhead computation. The cost is now $\approx$100% rise of the structural node data comparing to the minimal case of point (i), but the redundant in-

dicators enable simpler implementation of the nonrecursive and ϵ-subtree-related algorithms.

By adding further indicators for more than two descendants, multiway trees of order greater than *two* can be organized. However, such indicators differ in nature from the ones introduced in points (ii) and (iii) above since they cannot be proclaimed redundant.

Listing 1 shows an implementation of the binary tree node–-the basic block of the tree structure–-according to the point 3 scheme. The class `CBinTreeNode` is declared in the C++ code. The node-specific, useful, information is stored in the variable `m_uCount`.[1] The indicators to parent, to left, and to right child nodes are the member variables of the pointer type that points to the class itself, making their declaration recursive.

The tree nodes are created dynamically in the *tree feed* process, in which the words of length D are extracted from a time series to define the tree nodes [2]. The created main ϵ-tree and its ϵ-subtrees can now be analyzed by different tree traversals.

4. Implementation of the Algorithms

The recursive versions of algorithms in which a certain ***Function*** is performed on each tree node through the standard traversals are shown in Fig.1. In the pseudo-code, we have denoted that some kind of reference (`BTrNd-ref`) to the binary tree node must be passed to the recursive functions. As is well known, the *pre*, *in* or *post* traversal order is achieved by simply placing the *Function* prior, in between, or posterior to the recursive calls. The additional check that the node `Nd` is "not a leaf" makes the functions applicable also to our ϵ-subtrees (see Cor. 1ES.3). It can be omitted for the standard binary trees from Def. 1. The conditions for the recursive calls are written here in the explicit way to make the recursive structure clearer.

A more concise and elegant solution is shown in the C++ implementation of the recursive preorder function `CalcTreeStatPreOrdRR` (Listing 2). It is encapsulated within the DSA program's base tree class called `CABinTree`[2]. The function that operates on the tree nodes is *UpdateBinTreeStatistics* (Listing B1 in Appendix B). It checks if the node is a left or right child and a left or right leaf, and updates the ϵ-tree statistics accordingly. To establish if the node is a left or right descendant the parent indicator is needed (confer Listing 1). The number of `if` statements is reduced comparing to Fig. 1, and the function can be also applied to the nonexistent `(NULL)` node (the solution after [4], ch. 4).

Listing 1. Member variables of the `CBinTreeNode` class.

```
class CBinTreeNode
{
 private:
  UINT m_uCount;   // Node count.
  UINT m_uAbsLev; // Absolute node level.
        // Root nodes have m_uAbsLev:
       //   = 0 for the trees of mainTree type;
      //   >= 0 for the trees of subTree type.
  CBinTreeNode* m_pParent;  // Ptr. to parent.
  CBinTreeNode* m_pLChild;  // Ptr. to left ch.
  CBinTreeNode* m_pRChild; // Ptr. to right ch.
   // ...    ...
  // ...     ...
}
```

```
FunctionThruTrvPreOrdRR(BTrNd-ref Nd)
   Function(Nd);
   if Nd.LChild ≠ NULL and Nd not a leaf then
      FunctionThruTrvPreOrdRR(BTrNd.LChild);
   if Nd.RChild ≠ NULL and Nd not a leaf then
      FunctionThruTrvPreOrdRR(BTrNd.RChild);

FunctionThruTrvInOrdRR(BTrNd-ref Nd)
   if Nd.LChild ≠ NULL and Nd not a leaf then
      FunctionThruTrvInOrdRR(BTrNd.LChild);
   Function(Nd);
   if Nd.RChild ≠ NULL and Nd not a leaf then
      FunctionThruTrvInOrdRR(BTrNd.RChild);

FunctionThruTrvPostOrdRR(BTrNd-ref Nd)
   if Nd.LChild ≠ NULL and Nd not a leaf then
      FunctionThruTrvPostOrdRR(BTrNd.LChild);
   if Nd.RChild ≠ NULL and Nd not a leaf then
      FunctionThruTrvPostOrdRR(BTrNd.RChild);
   Function(Nd);
```

Fig. 1. Performing some ***Function*** on tree nodes in preorder, inorder and postorder traversals of a binary ϵ- tree by appropriately named recursive (RR) functions.

[1] In the DSA program the variable counts the number of occurrences of the substrings correspondent to the node, within the time series emitted from a dynamical system.

[2] Here A, for *Automata*, stands instead of ϵ in the name of the ϵ-binary trees. The function name is adapted to accommodate the needs of this paper.

Listing 2. Recursive implementation of the tree statistics calculation, done on every node in the preorder traversal.

```
CBinTreeNode* CABinTree::
   CalcTreeStatPreOrdRR(CBinTreeNode* pNd)
{
  if(pNd != NULL)
  {
   UpdateBinTreeStatistics(pNd);
   if(!IsLeaf(pNd))
   {
    CalcTreeStatPreOrdRR(pNd->GetpLChild());
    CalcTreeStatPreOrdRR(pNd->GetpRChild());
   }
  }
  return pNd;
}
```

Again, the additional `if` statement for the leaf status is not needed for the standard binary subtrees.

To do the same job the nonrecursive way, we follow the idea exposed in Fig. 2. The recursion from the previous solution is replaced by iteration in which some ***Function*** is performed on the nodes visited in a traversal of *SomeOrd* type, where *SomeOrd* = *PreOrd*, *InOrd*, or *PostOrd*. For each type of traversal the initialization and traversal functions must be provided. Given the reference to an ϵ-(sub)tree the initialization function returns the reference to the first node in the traversal. The traversal function takes a reference to the current node `NdC` and returns the reference to the next node, storing the traversal completion status in the `bDone` variable passed by reference.

An idea for the nonrecursive traversals through the tree nodes with redundant organizational data applied to specific multiway trees can be found in [5]. Our algorithms were tailored for the **binary** trees, i.e. specifically for the binary ϵ-trees. But, as is already shown above, they are readily adapted for the general binary trees by simplifying the leaf criteria.

```
// Auxiliary functions:
BTrNd-ref InitTrvSomeOrdNR(εBTr-ref Tr);
BTrNd-ref TrvSomeOrdNR(εBTr, bool-ref bDone);
 //TrvSomeOrd = TrvPreOrd, TrvInOrd, TrvPostOrd.

FunctionThruSomeOrdTrvNR(εBTr-ref Tr)
  bDone is false;
  NdC = InitSomeOrdTrvNR(Tr);

  while bDone is false
     Function(NdC);
     NdC = SomeOrdTrvNR(NdC, bDone);
```

Fig. 2. Performing some *Function* on the tree nodes in "some" order nonrecursive traversal of a binary ϵ-tree. The solution requires a pair of auxiliary functions for each traversal type: one to initialize it, and another to find the next node in the tree traversal.

Listing 3 gives the initialization and traversal functions for the case *SomeOrd* = *PreOrd* from our DSA program. The initialization is trivial because the preorder traversal starts from the root node. If the node is not a leaf, we **visit** the left child if it exists. Only if the left child does not exist we **visit** the right child. If the node is a leaf (at the leaf level for the ϵ-trees), we start the pullout to the root. The crucial function here is `GetpRSibling` (Listing 4), placed in the condition of the `while` loop. It inspects if the current node pointed by `pNdC` has its right sibling by checking that the current node is not the root, and that it is not a right child. If so, the pointer to the parent's right child is returned, either regular or `NULL`. Otherwise, it always returns `NULL`, since the right child cannot have a right sibling. Thus, when `GetpRSibling` returns `NULL`, the traversal is at the left node with no right sibling, or

Listing 3. Nonrecursive preorder initialization and traversal functions which return pointers to the next node.

```
CBinTreeNode* CABinTree::InitPreOrd
                (CABinTree* pTr) const
{
   return pTr->m_pRoot; // The class member var.
}

CBinTreeNode* CABinTree::TravrsPreOrd
    (CBinTreeNode* pNdC, bool& bDone) const
{
  CBinTreeNode* pNd;
  if(Level(pNdC) < m_uHeight) // Not a leaf.
    if((pNd = pNdC->GetpLChild()) != NULL)
       pNdC = pNd;           // To LChild first.
    else
       pNdC = pNdC->GetpRChild(); // To RChild.
  else // The current node is a leaf.
  {
    while((pNd = GetpRSibling(pNdC)) == NULL) // No
    {                                  // right sibling.
      if(Level(pNdC) > 0)
         pNdC = pNdC->GetpParent(); //To parent.
      else                           // At the root!
      {
        bDone = true;  // Traversal done, no more
        return pNdC;  // right sibligns.
      }
    }
    pNdC = pNd; // To the unvisited right sibling.
  }
  return pNdC;
}
```

Listing 4. Auxiliary function `GetpRSibling`.

```
CBinTreeNode* CABinTree::
   GetpRSibling(const CBinTreeNode* pNd) const
{
  CBinTreeNode* pNdR;
  if(Level(pNd) > 0 && (pNdR = pNd->
            GetpParent()->GetpRChild()) != pNd)
     return pNdR; // *pND is a LChild, return ptr.
         // to its right sibling (could be NULL).
  else  // Current is the (sub)tree root, or the
       // RChild, so there is no right sibling.
     return NULL;
}
```

at the right node that was already visited, or at the root level. If not at the root level, we pull up to the parent node. Else, the node is root, and the traversal is done. If `GetpRSibling` returns a valid pointer, we **visit** the node. It could not have been visited before because on the way down a right child was not visited if its left sibling existed. And if the left sibling did not exist, this was a solo right node that could not have a right sibling. Thus, when the function `GetpRSibling` function returns `NULL`, even if the right node exists we ought to skip it because it was visited on the way down, and the pullout to the parent continues. Summarily: on the way down traversal visited every left node and all the single right nodes without left siblings. On the way up only the right nodes with left siblings were visited. Thus, all the tree nodes were visited once and only once, which proves the correctness of the traversal.

In a similar way, the other two nonrecursive traversals were implemented by functions `InitInOrd`, `TravrsInOrd`, and `InitPostOrd`, `TravrsPostOrd` (after [7]), which are left for Appendix A.

5. Space-Time Analysis

The organizational data requirements of the tree node data structures and the corresponding variations of the algorithms' implementations and capabilities, were already discussed in Sec. 3. With dynamically created tree nodes, the space complexity S of the tree structure and the algorithms operating on it will be proportional to the number N_{Nds} of nodes. Let s_{Nd} represent the space for storing one tree node and let free terms be neglected. Then the space complexity is: $S(N_{Nds}) = s_{Nd}N_{Nds} = O(N_{Nds})$. From Cor. 1E.3 it can be expressed via the tree height D as:

$$s_{Nd}(D+1) \le S(D) \le s_{Nd}(2^{D+1}-1)\ , \qquad (1a)$$

$$\langle S(D)\rangle_{ave} \approx s_{Nd}\left(2^D + \tfrac{D}{2}\right)\ . \qquad (1b)$$

In the recursive solutions, we shall additionally use the stack space for the function call stack frames of size s_{StcFr}. For ϵ-trees the recursion and the corresponding stack depth will be breathing between 1 and D call stack frames. For the general trees the range will be the same with the upper value averaging to $D/2$ frames. By taking the maximum number that will be required by the algorithm, the space complexity S_{RR} for the recursive algorithms is:

$$\langle S_{RR}(D)\rangle_{ave} \approx s_{Nd}\left(2^D + \tfrac{D}{2}\right) + s_{StcFr}D\,. \qquad (1c)$$

The nonrecursive solution's single call per iteration will always spend the constant space of one stack frame. Summarily, the exponential term prevails, and the space complexity order of magnitude is:

$$S(D) = O(2^D)\ ,\ S(N_{Nds}) = O(N_{Nds})\,. \qquad (1d)$$

The time complexity T has an analogous behavior. With t_{Fn} denoting the time needed for the execution of the *`Function`* (see Fig. 1 and 2), and t_{Cll} being some average time needed for the function call and return, the time complexity T_{RR} for the recursive functions can be written as:

$$(t_{Fn}+t_{Cll})D \le T_{RR}(D) \le (t_{Fn}+t_{Cll})(2^{D+1})\,, \qquad (2a)$$

$$\langle T_{RR}(D)\rangle_{ave} \approx (t_{Fn}+t_{Cll})\left(2^D + \tfrac{D}{2}\right)\ , \qquad (2b)$$

$$T_{RR}(D) = O(2^D)\ ,\ T_{RR}(N_{Nds}) = O(N_{Nds})\,. \qquad (2c)$$

For the nonrecursive functions organized according to the algorithm from Fig. 2, the situation is similar. For the first node, the traversal initialization function is called, and for every next node the corresponding traversal function. Every time the *`Function`* is applied, the time t_{Fn} is spent. However, while the location of the next node to be visited is built in the very structure of the recursive algorithms, the nonrecursive traversal algorithms do spend additional time for the

task. This can be time consuming and less efficient especially for asymmetrical and less diversified trees where the function `GetpRSibling` can spend quite some time before the pullout results in the next right child to be visited. Let t_{Nxt} denotes the average time for the location of the next node in the traversal, and let $t_{NR,tot} = t_{Fn} + t_{Cll} + t_{Nxt}$ be the total time constant. Then the time complexity T_{NR} for the nonrecursive algorithm is:

$$t_{NR,tot}(D+1) \le T_{NR}(D) \le t_{NR,tot}(2^{D+1}-1) \quad (3a)$$

$$\langle T_{NR}(D)\rangle_{ave} \approx t_{NR,tot}\left(2^{D}+\frac{D}{2}\right), \quad (3b)$$

$$T_{NR}(D) = O(2^{D}), \; T_{NR}(N_{Nds}) = O(N_{Nds}). \quad (3c)$$

The time constants t_{Cll} are assumed to be the same for the recursive calls (and returns) in eq. (2), and for the call (and return) from the traversal functions in (3). We could hope to reduce the $t_{NR,tot}$ for t_{Cll} if avoiding the call to the `TrvSomeOrd` in the function scheme in Fig. 2. This requires that the nonrecursive move to the next node is implemented directly within the `while` loop. Such embedded nonrecursive algorithms are in fact most often found in literature [4, 6]. Alternatively, we could declare the traversal functions as inline and again hope that the compiler will accept it for the functions this big.

6. Execution Time Measurements

The above theoretical presumptions will now be contrasted to the measured values. Functions that serve specifically for the testing of the tree traversal algorithms were added to our DSA program. The execution times for the exemplary recursive and nonrecursive algorithms finding the tree node statistics are given in Table 1. The recursive algorithm is that from Listing 2, and the nonrecursive is constructed according to the idea of Fig. 2, with some ***`Function`*** being replaced by the `UpdateBinTreeStatistics` (Listing B1).

Two dynamical systems ("no consecutive zeros" and Bernoulli coin flip) fed their ϵ-trees of height 20, with 20×10^6 words each. The former formed a rarely filled and slightly asymmetrical tree, and the latter produced the full tree. The first row in the table shows the average time $\overline{\Delta t}$, followed by the absolute and relative values of its standard deviation ($\pm$). The fourth row gives the average time per tree node, and the last row is the measured time relative to the time $\overline{\Delta t}_{RR}$ of the recursive function. The time measurement was organized as shown in Listing B2. The time functions query the performance counter, so that the accuracy is close to the processor clock cycle period. As a safe estimate, let's say that the precision is better than 10 ns.

Prior starting the measurement, the usual precautions were done, like turning off all the unnecessary user and system processes. Furthermore, each function was prerun in order to make the initial conditions for all tested functions as similar as possible (confer Listing B2). Thus we can expect to avoid the influence of the operating system's processes that could have been running prior the function call. In this way we also assure that the processor's cache memories and the instruction pipelines will be prepared optimally and similarly for all the measured functions.

The execution time statistic was made on the basis of at least 10 stable and consistent measurements, without greater fluctuations. Having said all this, it is obvious that the standard deviation, which significantly exceeds the clock precision, must be caused solely by the remaining and unavoidable sway of the multitasking OS (Windows XP, running on Pentium 4 at 2.6 GHz).

Table 1. Execution time measurements of the recursive and nonrecursive algorithms. The algorithms call the exemplary function UpdateBinTree`Stat` (see Appendix B).

$D = 20$	ϵ-Tree 1 $N_{Nds} = 46366$ 2.21% full		ϵ-Tree 2 $N_{Nds} = 2097151$ 100% full	
Calc. tree statistics	*Recursive (RR)*	*Nonrecurs. (NR)*	*Recursive (RR)*	*Nonrecurs. (NR)*
$\overline{\Delta t}$ / ms	5.024	5.742	285.35	320.67
$\pm$/ ms.	0.027	0.071	0.45	0.63
$\pm$ rel.	0.53%	1.23%	0.16%	0.20%
$\frac{\overline{\Delta t}}{N_{Nds}}$/µs	0.108	0.124	0.136	0.153
$\overline{\Delta t}$ / $\overline{\Delta t}_{RR}$	100.0%	114.3%	100.0%	112.4%

The measurements show that recursive algorithms are consistently faster, which came as a surprise to us (e.g. compare [6] ch. 2). But on the other side, this confirmed the dominance of the time constants in (3b) over those in (2b). The lag of nonrecursive algorithms for 12-14% is significantly greater than the measurement standard deviation of the order of 1% and less.

Additional measurements showed that there is no significant improvement when the call to traversal functions is exchanged for the embedded nonrecursive solution, as suggested earlier. Furthermore, for some other (simpler) functions, the nonrecursive solutions lag up to 30%, or stated the other way around, the recursive ones are roughly 25% faster.

Obviously, the simplicity of recursive algorithms won over the cost for the additional accessing of the dynamically allocated tree nodes in the nonrecursive traversal functions. The only extra expense of recursive solutions is a more intensive use of the system stack memory (eq. (1c) vs. (1d)). But it is linear in the tree height and makes no burden for modern computers.

A thing that requires further elaboration is the slight increase of the average execution times per tree node with the (huge) increase of the dynamically allocated nodes, as can be seen in the last but one row of Table 1. After the number of nodes increased approx. 45 times, the execution time per node rose ≈ 1.25 times.

7. Conclusion

When programming solutions for the real-world problems, a computer scientist is confronted with the need to design and study innovating data structures and algorithms. Within our DSA program –- which models dynamical systems from a time series in a "computational way"–-a specific sort of binary tree is used. It has all its leaves at the same level, and we call it binary ϵ-tree. For it we have defined and formally described ϵ-subtrees, which, according to our modeling scheme, present (sub)structures within structure.

The implementation of the tree traversals appropriate for the above binary trees with dynamically allocated nodes resulted in original and generally applicable nonrecursive algorithms. These were then complemented with the corresponding recursive versions for the comparison and testing purposes. The algorithms are applicable to either the ϵ-(sub)trees or the general binary trees by a simple adaptation of the leaf criteria according to the tree definitions.

In a brief analysis we have derived space-time complexities for both versions of the algorithms, showing that the recursive ones can be expected to have better time constants. That was confirmed by precise execution time measurements.

Nevertheless, the nonrecursive algorithms have their advantages, like when the system stack memory is limited, or when a step-by-step traversal through the tree nodes is desired. That is due to the ability of the nonrecursive algorithms to return the pointer to the next node in the tree traversal.

In a versatile programming application like ours, the programmer will choose a more appropriate version of an algorithm to meet specific needs.

References

[1] J. P. Crutchfield, Computational Mechanics Publications: http://csc.ucdavis.edu/~chaos/chaos/pubs.htm (April 2012).

[2] R. Logozar, A. Lovrencic, The Modeling and Complexity of Dynamical Systems by Means of Computation and Information Theories, Journal of Information and Organizational Sciences (JIOS), vol. 35. No. 2, 2011.

[3] E. Horowitz, S. Sahni, Fundamentals of Computer Algorithms, Pitman Publish. Limited, London 1979.

[4] N. Wirth, Algorithms & Data Structures (New Edition), Prentice/Hall International, London, 1986.

[5] A. B. Tucker *at al*., Fundamentals of Computing II, C++ Edition, McGraw-Hill, New York 1995.

[6] A. Aho, J. E. Hopcroft, J. D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, Massachusetts, 1974.

[7] R. Logozar, Modeling of Dynamical Systems by Stochastic Finite Automata, Master Thesis (in Croatian), Faculty of Electrical Engineering and Computing, University of Zagreb, 1999.

Appendix A.

Listing A1. Inorder traversal initialization and traversal functions which return pointers to the next node.

```
CBinTreeNode* CABinTree::InitInOrd
                (CABinTree* pTr) const
{
  CBinTreeNode *pNd, *pNdC = pTr->m_pRoot;
  while( (pNd = pNdC->GetpLChild()) != NULL &&
         Level(pNdC) < m_uHeight ) // To the L-most
     pNdC = pNd;                  // consec. LChild.
  return pNdC;
}

CBinTreeNode* CABinTree::TravrsInOrd
  (CBinTreeNode* pNdC, bool& bTrvrsDone) const
{
  CBinTreeNode* pNd;
  if( Level(pNdC) < m_uHeight &&    // Unvisited
     (pNd = pNdC->GetpRChild()) != NULL ) // STr.
  {                                 // to the right.
    pNdC = pNd;        // To the right child.
    while((pNd = pNdC->GetpLChild()) != NULL &&
          Level(pNdC) < m_uHeight) )  // While
       pNdC = pNd;     // possible, to the LChild.
  }
  else if(IsLeftChild(pNdC)) // An LChild without
                             // its own RChild.
    pNdC = pNdC->GetpParent();  // To Parent.
  else        // An RChild without its own RChild.
  {
    do
       pNdC = pNdC->GetpParent(); // Pullout while
    while(IsRightChild(pNdC));  // RChild.
    if(Level(pNdC) > 0)
       pNdC = pNdC->GetpParent();
    else                       // At the root.
       bTrvrsDone = true;
  }
  return pNdC;
}
```

Listing A2. Postorder traversal initialization and traversal functions which return pointers to the next node.

```
CBinTreeNode* CABinTree::InitPostOrd
                (CABinTree* pTr) const
{
  CBinTreeNode *pNd, *pNdC = pTr->m_pRoot;
  while (Level(pNdC) < m_uHeight) // To L-most leaf.
  {
    if((pNd = pNdC->GetpLChild()) != NULL)
       pNdC = pNd;  // To LChild.
    else pNdC = pNdC->GetpRChild();  // To RChild.
  }        // pNdC is pointing to the L-most leaf.
  return pNdC;
}

CBinTreeNode* CABinTree::TravrsPostOrd
  (CBinTreeNode* pNdC, bool& bTrvrsDone) const
{
  CBinTreeNode* pNd;
  if(pNdC == m_pRoot)  // If postorder at the root,
    bTrvrsDone = true; // then it is done.
  else if((pNd = GetpRSibling(pNdC)) != NULL)
  {              // LChild with the RSibling.
    pNdC = pNd;      // To RSibling, and look for its
    while(Level(pNdC) < m_uHeight)  // L-most leaf.
      if((pNd = pNdC->GetpLChild()) != NULL)
         pNdC = pNd;                 // To LChild.
      else
         pNdC = pNdC->GetpRChild(); // To RChild.
  }
  else      // RChild, or there is no RSibling.
    if(Level(pNdC) > 0)
       pNdC = pNdC->GetpParent();
    else                       // Done. No more
       bTrvrsDone = true;   // RSiblings.
  return pNdC;
}
```

Appendix B.

Listing B1. Function `UpdateBinTreeStatistics` used for testing of the recursive and nonrecursive algorithms.

```
void CABinTree::UpdateBinTreeStatistics
                      (CBinTreeNode* pNd)
{
  if(IsLeftChild(pNd))
    if(IsLeaf(pNd))
    {
       m_TreeStat.IncrLLeaves(); // Incr. L.Nodes
       m_TreeStat.AddToSumLLvsCnts
                    (pNd->GetuNCount());
    }
    else  m_TreeStat.IncrLNodes();
  else if(IsRightChild(pNd))
    if(IsLeaf(pNd))
    {
       m_TreeStat.IncrRLeaves(); // Incr. L.Nodes
       m_TreeStat.AddToSumRLvsCnts
                    (pNd->GetuNCount());
    }
    else  m_TreeStat.IncrRNodes();
}
```

Listing B2. The code excerpt showing the organization of the time measurement functions.

```
switch(trvOR) // trvOr = traversal type.
{
case preOrdRR: // Call recursive preorder alg.
  if(iInclPRFncs == 2) // Include the func. prerun.
    pNd = pSubTr->NodeCountPreOrdRR(); // Prerun.
  hrTimer.StartTimer();
  pNd = pSubTr->NodeCountPreOrdRR();
  dtsDur = hrTimer.StopTimer();
  break;
case preOrdNR: // Call nonrecursive preorder alg.
  if(iInclPRFncs == 2) // Include the func. prerun.
    pNd = pSubTr->NodeCountPreOrdNR(); // Prerun.
  hrTimer.StartTimer();
  pNd = pSubTr->NodeCountPreOrdNR();
  dtsDur = hrTimer.StopTimer();
  break;
 // ...    ...    ...
}
```