A Single Traversal Algorithm for Serializing any Binary Tree

Austin Owino Wetoyi
Vrije Universiteit
Faculty of Science and Bio-Engineering Sciences
Department of Computer Science
(Brussels, Belgium)
Email: *research [AT] austinowino.com*

ABSTRACT— ***It is possible to save a binary tree to a file and restore it at a later time. Because these two operations are frequently necessary and therefore important, an efficient algorithm to represent a binary tree in a compact way is very desirable. For special types of binary trees, there are efficient algorithms, otherwise, there is a method that stores both the inorder and either preorder traversal or postorder traversal of the binary tree which requires two traversals of the binary tree during construction of the storage array. In this paper, I present an approach for which only one traversal is sufficient for construction of the storage array. I also present a more efficient complement algorithm for restoration of the binary tree.***

Keywords— Binary Tree, Tree Traversal, Data Compression, Binary Tree Linearization

1. INTRODUCTION

Writing a tree to a file is called *serialization* of the tree and reading back from the file to reconstruct the exact same tree is known as *deserialization* of the tree. These two operation are necessary in any of the following situations:

- There is need to transmit a tree via network and load it into another host
- The tree must be usable after a power outage

The process of storing a tree degrades to representing the binary tree as a linear structure like an array from which the nodes can then each be written in turn to a file on a file system.

2. BINARY TREE REPRESENTATIONS

This section is a look at two ways of implementation of a binary tree.

4.1 Array Implementation

One way of storing an a binary tree in an array and still preserve its functionalityas a binary tree is to store the root of the tree in the first element of the array. Then, for each node in the tree that is stored at subscript k, the node's left child can be stored at subscript $2k+1$ and the right child can be stored at subscript $2k+2$, (k = 0, 1, 2, ...).

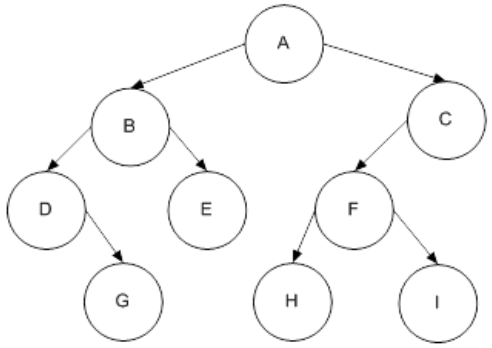

Figure 1: A non-complete binary tree

This way, the tree in ***Figure 1*** will be stored in an array as shown in ***Table 1***.

Table 1: Array implementation of the binary tree in Figure 1

A	B	C	D	E	F			G			H	I

We end up with 4 elements wasted in the array. The amount of space wasted depends on the shape of the tree. This implementation works very well for a complete binary tree, a binary tree in which each level of the tree is completely filled except possibly the bottom level, but there, the nodes are in the leftmost positions i.e. if the bottom level is not filled out, it has all of its nodes filled in from left to right. The completeness property guarantees that the data items will be stored in consecutive locations at the beginning of the array; if there are N nodes in a complete tree, the nodes will correspond to the first N positions of the array.

4.2 Linked Representation

It is clear that if you store a binary tree that is not complete using the scheme above, you can end up with a great deal of wasted space in the array. Because of this, a binary tree that is not guaranteed to be complete is implemented as a linked structure in which each node of the structure has two links, one to its left child if any and another to its right child if any. See Figure 2. This way, a node can be anywhere in memory.

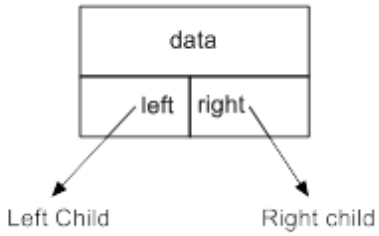

Figure 2: Visual representation of a node of a binary tree

The two link fields ***left*** and ***right*** in a node are pointers to nodes representing its left and right children, respectively, or are null pointers if the node does not have a left or a right child. A leaf node is thus characterized by having null values for both ***left*** and ***right***. See ***Figure 3***.

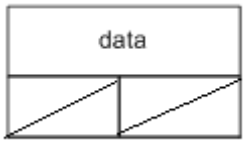

Figure 3: Visual representation of a leaf node of a binary tree

A linked structure also provides additional flexibility when the tree grows or shrinks. The size of an array cannot be changed once created; the changing of array size whenever necessary can only be simulated by creating a new array and then copying all data from the array with the old size to the new array which is clearly an expensive operation in terms of time and space usage. Also, insertion or deletion of a node in an array implementation would require the expensive shifting of other nodes, because of the requirement that elements of an array are adjacent (separated by the same distance) in computer memory. In a linked structure implementation, insertion and/or deletion of a node involves only the relatively cheap operation of manipulation of links.

3. THE TWO-TRAVERSAL METHOD

A binary tree can be stored on file (or serialized) as two arrays. One of the arrays stores the preorder traversal of the binary tree and the other stores either postorder traversal of the binary tree or preorder traversal of the binary tree. A unique binary tree cannot be built or deserialised from a combination of preorder and postorder traversals [1].

The following is the procedure for construction of a binary tree from two traversals.

1. *If preorder traversal is given, then the first node in the linear order is the root node. If the postorder traversal is given, then the last node is the root node.*
2. *Once the root node is known, the nodes in the left subtree and the right subtree are established by using inorder sequence.*
3. *The above two steps are repeated to form subtree and finally the required binary tree.*

Table 2 is a description of the binary tree of Figure 1 using its inorder traversal and its preorder traversal. This example is from [1].

Table 2: A representation of a binary tree using inorder and preorder traversals

inorder	D	G	B	E	A	H	F	I	C
preorder	A	B	D	G	E	C	F	H	I

From the preorder traversal, clearly A is the root (first node in the preorder is the root). In the inorder traversal, D, G, B, and E are the nodes in the left subtree (all nodes left of root A) and H, F, we, and C in the right subtree (all nodes right of A) as shown in ***Figure 4***.

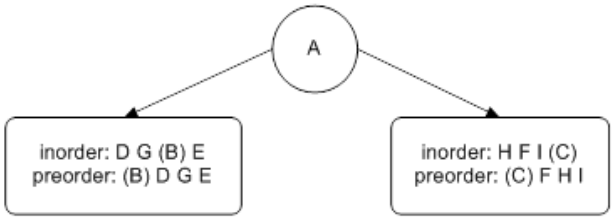

Figure 4

By repeating the same procedure, B is the root node of the left subtree and C is the root of the right subtree. These root nodes are bracketed. See ***Figure 5***. E is the right node of B. D is the root node of the left subtree and F is the root of the right subtree.

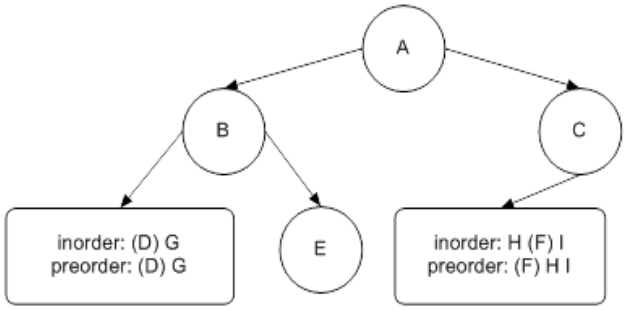

Figure 5

H and I are the left and right child nodes of F. G is the right child node of D. This way, you end up with the binary tree in Figure 1. Because, it is possible to reconstruct a binary tree from its two traversals, we can just store these two traversals which are just arrays. Then when we want to use the tree, we use the above procedure to reconstruct it first. In this paper, I will call this method the ***two-traversal method***.

Note: The inorder traversal visits nodes of a binary search tree (BST) in non-decreasing (ascending) order of the values of the keys of the nodes. So using this method, to store a BST, you only need to store either the postorder traversal or the preorder traversal.

4. THE ONE TRAVERSAL METHOD

The implementation of the binary tree in Figure 1 as an array is shown in ***Table 1***. The problem with the scheme is that space is wasted in the array implementing the tree if the array is to support the functionality of the binary tree.

If we need the array only for storage and not at the same time implementing the functionality of the binary tree, then we could remove the wasted spaces in the array and remain with the array as shown in ***Table 3***.

Table 3

A	B	C	D	E	F	G	H	I

When the empty elements are not there, the array is no longer a representation (implicit) of the binary tree. To restore the tree, you must also know the index of the element each key would occupy in the array that implements the binary tree. In other words we need the information in ***Table 4***.

Table 4

index	0	1	2	3	4	5	8	11	12
key	A	B	C	D	E	F	G	H	I

Given only the information ***Table 4***, can we efficiently reproduce the binary tree in ***Figure 1*** that it represents? And given the binary tree in ***Figure 1***, can we efficiently produce the information in ***Table 4*** that stores it? The answer to each of these is yes and I now present the algorithms for these two operations.

But before we look at the two algorithms, there is another condition. The algorithm that reconstructs the binary tree works only if the indices are in the sequence the nodes of the binary tree would be visited in preorder traversal. The content of ***Table 4*** must therefore be as shown in ***Table 5***.

Table 5: Preorder sequence of the nodes of binary tree of figure 1.

index	0	1	3	8	4	2	5	11	12
key	A	B	D	G	E	C	F	H	I

Therefore, the serializing algorithm should produce the preorder sequence of the keys and the indices of the elements the keys would occupy in an array implementing the binary tree.

4.3 The Serialization Algorithm

The serialization algorithm is simply the inorder traversal of the binary tree with the following actions performed during a visit;

1. store the index of the node,
2. compute the index of it left child, then recursively traverse the tree whose root is this child node.
3. compute the index of it right child, then recursively traverse the tree whose root is this child node.

The algorithm is shown in ***Algorithm 1***

Algorithm 1: The serialization algorithm

```
ALGORITHM serialize(root, preIndex[N], preData[N], i=0, index=0)
/*
 * encodes the structure of a binary tree
*/
//INPUT: root of a binary tree
//OUTPUT: array containing storing the binary tree
if (root != null AND i < N)
   preIndex[i] = index
   preData[i] = root.data
   serialize(root.left, preIndex[N], preData[N], i+1, 2*index+1)
   serialize(root.right, preIndex[N], preData[N], i+1, 2*index+2)
```

4.4 The deserialization algorithm

The algorithm reconstructs a binary tree from the preorder sequence of the indices of the array elements the keys would be stored in if the binary tree were to be implemented as an array. Since the mapping of indices to keys is one-to-one, if a language supports records or structures, you don't need to use two arrays as ***Table 5*** illustrates. Instead, you use only one array whose elements are each a composition of a key node and its array index as its fields. The form in which the algorithm is presented is simplified to work only with a sequence of the indices. Modifying the algorithm to work with both indices and key is trivial.

Algorithm 2: The iterative deserialization algorithm

```
ALGORITHM iterativeTreeReconstructor(preArray[N])
/*
 * Reconstructs binary tree from preorder sequence of the indices
 * the keys in its nodes would occupy if the binary tree were to
 * be stored in an array
*/
//Input: array of indices of the elements that would store the nodes
//Output: the reconstructed binary tree
parentStack = empty stack
newNode = a new node whose key correspond to preArray[0]
parentStack.push(node)
for index = 1 to N - 1
if (preArray[index] == 2 * parentStack.peek() + 1)
```

```
newNode = new node corresponding to preArray[index]
parentNode = parentStack.peek()
parentNode.left = newNode
parentStack.push(newNode)
else
while (parentStack.peek() != preArray[index] / 2 - 1)
parentStack.pop()
newNode = new node corresponding to preArray[index]
parentNode = parentStack.peek()
parentNode.right = newNode
parentStack.push(newNode)
```

//================ end of iterative version ==================

Algorithm 3: The recursive deserialization algorithm

The following is the recursive version of the above algorithm.

```
ALGORITHM recursiveTreeReconstructor(preArray[N], parentStack, index = 0)
/*
 * Reconstructs a binary tree from the preorder sequence of
   the indices its nodes would occupy if the binary tree were
   to be stored in an array
*/
//Input: array of indices of elements that would store the keys
//Output: the reconstructed binary tree

if (index == 0 && N > 0)
newNode = a new node whose key correspond to preArray[0]
parentStack.push(newNode)
else
if (index < N)
//left subtree
if(preArray[index] == 2 * parentStack.peek() + 1)
newNode = new node corresponding to preArray[index]
parentNode = parentStack.peek()
parentNode.left = newNode
parentStack.push(newNode)
recursiveTreeReconstructor(preArray[N], index + 1)
//right subtree
else
while(parentStack.peek() != preArray[index] / 2 - 1)
parentStack.pop()
newNode = new node corresponding to preArray[index]
parentNode = parentStack.peek()
parentNode.right = newNode
parentStack.push(newNode)
recursiveTreeReconstructor(preArray[N], index + 1)
//================ end of recursive version ====================
```

5. CONCLUSION

This paper has demonstrated that the structure (the shape) of any binary tree, complete or not, can be stored in an array of integers of the same number of elements as the nodes of the binary tree. The paper has also shown how the binary tree can be reconstructed from the array of integers. The following are advantages this method, the ***one-traversal method***, has over the method of reconstruction from two traversals, the ***two-traversal method***.

4.5 Shorter process of storage.

The ***one-traversal method*** requires only one traversal to store the tree while the ***two-traversal method*** will require

two traversals.

4.6 Efficiency

The ***one-traversal method***during reconstruction, does searching only done when creating the right subtree. In the ***two-traversal method***, searching is done each time the root node has been identified i.e. at each iteration.

4.7 Storage space

If we take it that the preorder traversal sequence defines the binary tree, then any other information is extraneous. This information for the ***two-traversal method*** is the inorder traversal sequence of the keys in the nodes while for ***one-traversal method***, it is a sequence of integers. For any non-trivial application, the data type for key is always likely to be of a larger size than integer data types. This means that the ***two-traversal method*** will store a binary tree in a smaller space than the ***two-traversal method***.

6. FUTURE WORK

A large fraction of the space in a typical binary tree node implementation is devoted to structural overhead, not storing data; each node contains two pointers in addition to carrying the data. The serialization that is achieved by the algorithm illustrated in this paper results in compressed form of the tree if the tree being serialized is implemented as a linked structure. For every node, the algorithm needs just one more piece of information while the overhead from a node implementation is 2. But the goal in data structure compression is to represent the structure in small space, but at the same time preserve its functionality. The serialization algorithm specifies the destination address (index) of a key in the array that would implement the tree. Because each key has its address in the array that would implement the tree known, it is possible that the storage array can permit queries using the rank and select (or similar) algorithms as explained in [2].

7. REFERENCES

[1] V. V. Muniswamy, Advanced Data Structures and Algorithms in C++, Mumbai: Jaico Publishing House, 2009.

[2] G. Jacobson, "Space-efficient static trees and graphs.," in *30th IEEE Symposium*, 1989.

8. BIBLIOGRAPHY

[3] S. L. O. P. Á. S. P. Antonio Fariña, "Rank and Select for Succinct Data Structures. Electronic Notes in Theoretical Computer Science. Volume 236, Pages 131–145," in *3rd International Workshop on View*, 2 April 2009.

9. APPENDIX

The following is a implementation in C++ of the reconstruction algorithms bundled together with the main function as a driver. The input array is a preorder sequence of the indices of the elements of the array that would store the nodes of the binary tree.

```
#include <iostream>
#include <stack>
void iterativeTreeReconstructor(int *preArray, int arraySize)
{
std::cout << "new node " << preArray[0]
<< " the ROOT node" << std::endl;
std::stack<int> node;
node.push(preArray[0]);
for (int nextNodeIndex = 1; nextNodeIndex < arraySize; nextNodeIndex++)
{
//left subtree
if (preArray[nextNodeIndex] == 2 * node.top() + 1)
{
std::cout << "new node " << preArray[nextNodeIndex]
<< " LEFT child of " << node.top() << std::endl;
node.push(preArray[nextNodeIndex]);
}
else
//right subtree
{
while(node.top() != preArray[nextNodeIndex] / 2 - 1) node.pop();
std::cout << "new node " << preArray[nextNodeIndex]
<< " RIGHT child of " << node.top() << std::endl;
node.push(preArray[nextNodeIndex]);
}
}
}
//==================================
void recursiveTreeReconstructor(int *preArray, int arraySize, std::stack<int>*
node, int nextNodeIndex = 0)
{
if (nextNodeIndex == 0 && arraySize > 0)
{
std::cout << "new node " << preArray[0] << " the ROOT node" << std::endl;
node->push(preArray[0]);
recursiveTreeReconstructor(preArray, arraySize, node, 1);
}
else
if (nextNodeIndex < arraySize)
{
//left subtree
if(preArray[nextNodeIndex] == 2 * node->top() + 1)
{
std::cout << "new node " << preArray[nextNodeIndex]
<< " LEFT child of " << node->top() << std::endl;
node->push(preArray[nextNodeIndex]);
recursiveTreeReconstructor(preArray, arraySize, node, ++nextNodeIndex);
}
//right subtree
else
{
while(node->top() != preArray[nextNodeIndex] / 2 - 1)
node->pop();
std::cout << "new node " << preArray[nextNodeIndex]
```

```
<< " RIGHT child of " << node->top() << std::endl;
node->push(preArray[nextNodeIndex]);
recursiveTreeReconstructor(preArray, arraySize, node, ++nextNodeIndex);
}
}
}
//=====================================

int main()
{
int pre[] = {0,1,3,8,4,2,5,11,12};
std::cout << "\n ==== iterative version ==== \n";
iterativeTreeReconstructor(pre,sizeof(pre)/sizeof(int));
std::cout << "\n ==== recursive version ==== \n";
std::stack<int> nodeStack;
recursiveTreeReconstructor(pre,sizeof(pre)/sizeof(int),&nodeStack);
//===================
std::cout << std::endl << "Strike ENTER/RETURN key to quit...";
std::cin.clear();
std::cin.get();
//==============
return 0;
}
```

The output of the above program is

```
==== iterative version ====
new node 0 the ROOT node
new node 1 LEFT child of 0
new node 3 LEFT child of 1
new node 8 RIGHT child of 3
new node 4 RIGHT child of 1
new node 2 RIGHT child of 0
new node 5 LEFT child of 2
new node 11 LEFT child of 5
new node 12 RIGHT child of 5
new node 6 RIGHT child of 2
new node 13 LEFT child of 6

 ==== recursive version ====
new node 0 the ROOT node
new node 1 LEFT child of 0
new node 3 LEFT child of 1
new node 8 RIGHT child of 3
new node 4 RIGHT child of 1
new node 2 RIGHT child of 0
new node 5 LEFT child of 2
new node 11 LEFT child of 5
new node 12 RIGHT child of 5
new node 6 RIGHT child of 2
new node 13 LEFT child of 6

Strike ENTER/RETURN key to quit...
```