Tree traversal

From Wikipedia, the free encyclopedia

In computer science, **tree traversal** (also known as **tree search**) is a form of graph traversal and refers to the process of visiting (examining and/or updating) each node in a tree data structure, exactly once, in a systematic way. Such traversals are classified by the order in which the nodes are visited. The following algorithms are described for a binary tree, but they may be generalized to other trees as well.

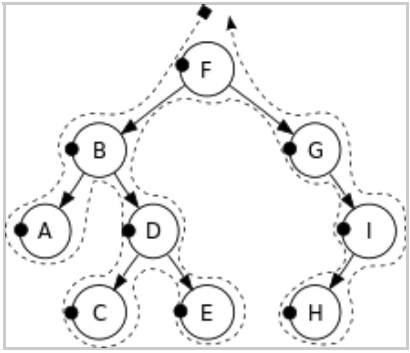
Contents

Types

Compared to linear data structures like linked lists and one-dimensional arrays, which have a canonical method of traversal (namely in linear order), tree structures can be traversed in many different ways. Starting at the root of a binary tree, there are three main steps that can be performed and the order in which they are performed defines the traversal type. These steps (in no particular order) are: performing an action on the current node (referred to as "visiting" the node), traversing to the left child node, and

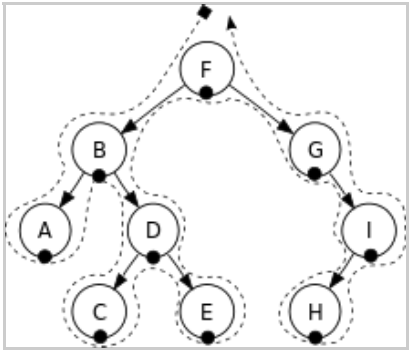
traversing to the right child node.

Traversing a tree involves iterating (looping) over all nodes in some manner. Because from a given node there is more than one possible next node (it is not a linear data structure), then, assuming sequential computation (not parallel), some nodes must be deferred – stored in some way for later visiting. This is often done via a stack (LIFO) or queue (FIFO). As a tree is a self-referential (recursively defined) data structure, traversal can naturally be described by recursion or, more subtly, corecursion, in which case the deferred nodes are stored implicitly – in the case of recursion, in the call stack.



Pre-order: F, B, A, D, C, E, G, I, H

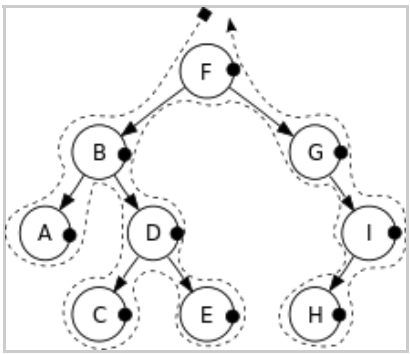
The name given to a particular style of traversal comes from the order in which nodes are visited. Most simply, does one go down first (depth-first: first child, then grandchild before second child) or across first (breadth-first: first child, then second child before grandchildren)? Depth-first traversal is further classified by position of the root element with regard to the left and right nodes. Imagine that the left and right nodes are constant in space, then the root node could be placed to the left of the left node (pre-order), between the left and right node (in-order), or to the right of the right node (post-order). There is no equivalent variation in breadth-first traversal – given an ordering of children, "breadth-first" is unambiguous.



In-order: A, B, C, D, E, F, G, H, I

For the purpose of illustration, it is assumed that left nodes always have priority over right nodes. This ordering can be reversed as long as the same ordering is assumed for all traversal methods.

Depth-first traversal is easily implemented via a stack, including recursively (via the call stack), while breadth-first traversal is easily implemented via a queue, including corecursively.



Post-order: A, C, E, D, B, H, I, G, F

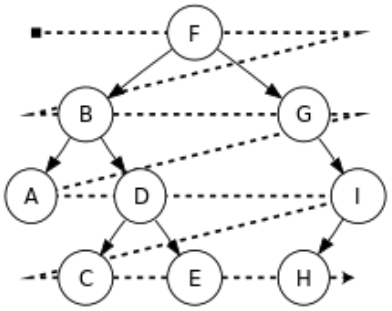
Beyond these basic traversals, various more complex or hybrid schemes are possible, such as depth-limited searches such as iterative deepening depth-first search.

Depth-first

There are three types of depth-first traversal: pre-order,[1] in-order,[1] and post-order.[1] For a binary tree, they are defined as display operations recursively at each node, starting with the root node, whose algorithm is as follows:

Pre-order

1. Display the data part of root element (or current element)
2. Traverse the left subtree by recursively calling the pre-order function.
3. Traverse the right subtree by recursively calling the pre-order function.



Level-order: F, B, G, A, D, I, C, E, H

In-order (symmetric)

1. Traverse the left subtree by recursively calling the in-order function.
2. Display the data part of root element (or current element).
3. Traverse the right subtree by recursively calling the in-order function.

Post-order

1. Traverse the left subtree by recursively calling the post-order function.
2. Traverse the right subtree by recursively calling the post-order function.
3. Display the data part of root element (or current element).

[2]

[3]

The trace of a traversal is called a sequentialisation of the tree. The traversal trace is a list of each visited root node. No one sequentialisation according to pre-, in- or post-order describes the underlying tree uniquely. Given a tree with distinct elements, either pre-order or post-order paired with in-order is sufficient to describe the tree uniquely. However, pre-order with post-order leaves some ambiguity in the tree structure.[4]

Generic tree

To traverse any tree in **depth-first order**, perform the following operations recursively at each node:

1. Perform pre-order operation
2. For each i (with $i = 1$ to n) do:
    1. Visit i-th, if present
    2. Perform in-order operation
3. Perform post-order operation

where n is the number of child nodes. Depending on the problem at hand, the pre-order, in-order or post-order operations may be void, or you may only want to visit a specific

child node, so these operations are optional. Also, in practice more than one of pre-order, in-order and post-order operations may be required. For example, when inserting into a ternary tree, a pre-order operation is performed by comparing items. A post-order operation may be needed afterwards to re-balance the tree.

Breadth-first

Trees can also be traversed in **level-order**, where we visit every node on a level before going to a lower level. This search is referred to as *breadth-first search*, as the search tree is broadened as much as possible on each depth before making going to the next depth.

Other types

There are also tree traversal algorithms that classify as neither depth-first search nor breadth-first search. One such algorithm is Monte Carlo tree search, which concentrates on analyzing the most promising moves, basing the expansion of the search tree on random sampling of the search space.

Applications

Pre-order traversal while duplicating nodes and edges can make a complete duplicate of a binary tree. It can also be used to make a prefix expression (Polish notation) from expression trees: traverse the expression tree pre-orderly.

In-order traversal is very commonly used on binary search trees because it returns values from the underlying set in order, according to the comparator that set up the binary search tree (hence the name).

Post-order traversal while deleting or freeing nodes and values can delete or free an entire binary tree. It can also generate a postfix representation of a binary tree.

Implementations

Depth-first

Pre-order

```
preorder(node)
  if node == null then return
  visit(node)
  preorder(node.left)
  preorder(node.right)
```

```
iterativePreorder(node)
  parentStack = empty stack
  while (not parentStack.isEmpty() or node ≠ null)
    if (node ≠ null)
      visit(node)
      if (node.right ≠ null) parentStack.push(node.right)
      node = node.left
    else
      node = parentStack.pop()
```

In-order

```
inorder(node)
  if node == null then return
  inorder(node.left)
  visit(node)
  inorder(node.right)
```

```
iterativeInorder(node)
  parentStack = empty stack
  while (not parentStack.isEmpty() or node ≠ null)
    if (node ≠ null)
      parentStack.push(node)
      node = node.left
    else
      node = parentStack.pop()
      visit(node)
      node = node.right
```

Post-order

```
postorder(node)
  if node == null then return
  postorder(node.left)
  postorder(node.right)
  visit(node)
```

```
iterativePostorder(node)
  parentStack = empty stack
  lastnodevisited = null
  while (not parentStack.isEmpty() or node ≠ null)
    if (node ≠ null)
      parentStack.push(node)
      node = node.left
    else
      peeknode = parentStack.peek()
      if (peeknode.right ≠ null and lastnodevisited ≠ peeknode.right)
        /* if right child exists AND traversing node from left child, move right */
        node = peeknode.right
      else
        visit(peeknode)
        lastnodevisited = parentStack.pop()
```

All the above implementations require call stack space proportional to the height of the tree. In a poorly balanced tree, this can be considerable. We can remove the stack requirement by maintaining parent pointers in each node, or by threading the tree (next section).

Morris in-order traversal using threading

A binary tree is threaded by making every left child pointer (that would otherwise be null) point to the in-order predecessor of the node (if it exists) and every right child pointer (that would otherwise be null) point to the in-order successor of the node (if it exists).

Advantages:

1. Avoids recursion, which uses a call stack and consumes memory and time.
2. The node keeps a record of its parent.

Disadvantages:

1. The tree is more complex.
2. We can make only one traversal at a time.
3. It is more prone to errors when both the children are not present and both values of nodes point to their ancestors.

Morris traversal is an implementation of in-order traversal that uses threading:

1. Create links to the in-order successor
2. Print the data using these links
3. Revert the changes to restore original tree.

Breadth-first

Also, listed below is pseudocode for a simple queue based level order traversal, and will require space proportional to the maximum number of nodes at a given depth. This can be as much as the total number of nodes / 2. A more space-efficient approach for this type of traversal can be implemented using an iterative deepening depth-first search.

```
levelorder(root)
  q = empty queue
  q.enqueue(root)
  while not q.empty do
    node := q.dequeue()
    visit(node)
    if node.left ≠ null then
      q.enqueue(node.left)
    if node.right ≠ null then
      q.enqueue(node.right)
```

Infinite trees

While traversal is usually done for trees with a finite number of nodes (and hence finite depth and finite branching factor) it can also be done for infinite trees. This is of particular interest in functional programming (particularly with lazy evaluation), as infinite data structures can often be easily defined and worked with, though they are not (strictly) evaluated, as this would take infinite time. Some finite trees are too large to represent explicitly, such as the game tree for chess or go, and so it is useful to analyze them as if they were infinite.

A basic requirement for traversal is to visit every node. For infinite trees, simple algorithms often fail this. For example, given a binary tree of infinite depth, a depth-first traversal will go down one side (by convention the left side) of the tree, never visiting the rest, and indeed if in-order or post-order will never visit *any* nodes, as it has not reached a leaf (and in fact never will). By contrast, a breadth-first (level-order) traversal will traverse a binary tree of infinite depth without problem, and indeed will traverse any tree with bounded branching factor.

On the other hand, given a tree of depth 2, where the root node has infinitely many children, and each of these children has two children, a depth-first traversal will visit all nodes, as once it exhausts the grandchildren (children of children of one node), it will move on to the next (assuming it is not post-order, in which case it never reaches the root). By contrast, a breadth-first traversal will never reach the grandchildren, as it seeks to exhaust the children first.

A more sophisticated analysis of running time can be given via infinite ordinal numbers; for example, the breadth-first traversal of the depth 2 tree above will take ?·2 steps: ? for

the first level, and then another ? for the second level.

Thus, simple depth-first or breadth-first searches do not traverse every infinite tree, and are not efficient on very large trees. However, hybrid methods can traverse any (countably) infinite tree, essentially via a diagonal argument ("diagonal" – a combination of vertical and horizontal – corresponds to a combination of depth and breadth).

Concretely, given the infinitely branching tree of infinite depth, label the root node $()$, the children of the root node $(1), (2), \ldots$, the grandchildren $(1,1), (1,2), \ldots, (2,1), (2,2), \ldots$, and so on. The nodes are thus in a one-to-one correspondence with finite (possibly empty) sequences of positive numbers, which are countable and can be placed in order first by sum of entries, and then by lexicographic order within a given sum (only finitely many sequences sum to a given value, so all entries are reached – formally there are a finite number of compositions of a given natural number, specifically 2^{n-1} compositions of $n = 1$;), which gives a traversal. Explicitly:

```
0: ()
1: (1)
2: (1,1) (2)
3: (1,1,1) (1,2) (2,1) (3)
4: (1,1,1,1) (1,1,2) (1,2,1) (1,3) (2,1,1) (2,2) (3,1) (4)
```

etc.

This can be interpreted as mapping the infinite depth binary tree onto this tree and then applying breadth-first traversal: replace the "down" edges connecting a parent node to its second and later children with "right" edges from the 1st child to the 2nd child, 2nd child to third child, etc. Thus at each step one can either go down (append a (,1) to the end) or go right (add 1 to the last number) (except the root, which is extra and can only go down), which shows the correspondence between the infinite binary tree and the above numbering; the sum of the entries (minus 1) corresponds to the distance from the root, which agrees with the 2^{n-1} nodes at depth n-1 in the infinite binary tree (2 corresponds to binary).

References

1. ^ **a b c** http://webdocs.cs.ualberta.ca/~holte/T26/tree-traversal.html
2. ^ http://www.cise.ufl.edu/~sahni/cop3530/slides/lec216.pdf
3. ^ http://www.programmerinterview.com/index.php/data-structures/preorder-traversal-algorithm/
4. ^ Which combinations of pre-, post- and in-order sequentialisation are unique? (http://cs.stackexchange.com/questions/439/which-combinations-of-pre-post-and-in-order-sequentialisation-are-unique)

General

- Dale, Nell. Lilly, Susan D. "Pascal Plus Data Structures". D. C. Heath and Company. Lexington, MA. 1995. Fourth Edition.

- Drozdek, Adam. "Data Structures and Algorithms in C++". Brook/Cole. Pacific Grove, CA. 2001. Second edition.
- http://www.math.northwestern.edu/~mlerma/courses/cs310-05s/notes/dm-treetran

External links

- Animation Applet of Binary Tree Traversal (http://www.cosc.canterbury.ac.nz/people/mukundan/dsal/BTree.html)
- The Adjacency List Model for Processing Trees with SQL (http://www.SQLSummit.com/AdjacencyList.htm)
- Storing Hierarchical Data in a Database (http://www.sitepoint.com/hierarchical-data-database/) with traversal examples in PHP
- Managing Hierarchical Data in MySQL (http://dev.mysql.com/tech-resources/articles/hierarchical-data.html)
- Working with Graphs in MySQL (http://www.artfulsoftware.com/mysqlbook/sampler/mysqled1ch20.html)
- Non-recursive traversal of DOM trees in JavaScript (http://www.jslab.dk/articles/non.recursive.preorder.traversal)
- Sample code for recursive and iterative tree traversal implemented in C. (http://code.google.com/p/treetraversal/)
- Sample code for recursive tree traversal in C#. (http://arachnode.net/blogs/programming_challenges/archive/2009/09/25/recursive-tree-traversal-orders.aspx)
- See tree traversal implemented in various programming language (http://rosettacode.org/wiki/Tree_traversal) on Rosetta Code
- Tree traversal without recursion (http://www.perlmonks.org/?node_id=600456)

Categories: Trees (data structures) | Graph algorithms | Recursion | Iteration in programming

---