Tree traversal

In computer science, **tree traversal** (also known as **tree search**) is a form of graph traversal and refers to the process of visiting (checking and/or updating) each node in a tree data structure, exactly once. 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

Unlike linked lists, one-dimensional arrays and other linear data structures, which are canonically traversed in linear order, trees may be traversed in multiple ways. They may be traversed in depth-first or breadth-first order. There are three common ways to traverse them in depth-first order: in-order, pre-order and post-order.[1] Beyond these basic traversals, various more complex or hybrid schemes are possible, such as depth-limited searches like iterative deepening depth-first search.

Data structures for tree traversal

Traversing a tree involves iterating 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 be defined by recursion or, more subtly, corecursion, in a very natural and clear fashion; in these cases the deferred nodes are stored implicitly in the call stack.

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

Depth-first search

These searches are referred to as *depth-first search* (DFS), as the search tree is deepened as much as possible on each child before going to the next sibling. For a binary tree, they are defined as display operations recursively at each node, starting with the root, whose algorithm is as follows:[2] [3]

The general recursive pattern for traversing a (non-empty) binary tree is this: At node N do the following:

(L) Recursively traverse its left subtree. This step is finished at the node N again.

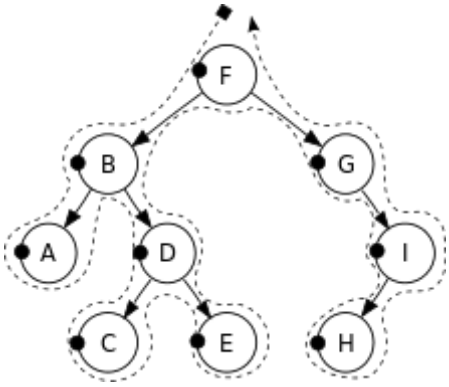
(R) Recursively traverse its right subtree. This step is finished at the node N again.

(N) Process N itself.

These steps can be done *in any order*. If (L) is done before (R), the process is called left-to-right traversal, otherwise it is called right-to-left traversal. The following methods show left-to-right traversal:

Pre-order (NLR)

1. Check if the current node is empty or null.
2. Display the data part of the root (or current node).
3. Traverse the left subtree by recursively calling the pre-order function.
4. Traverse the right subtree by recursively calling the pre-order function.



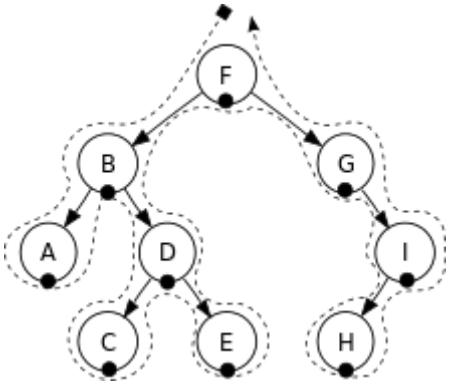
Pre-order: F, B, A, D, C, E, G, I, H.

The pre-order traversal is a topologically sorted one, because a parent node is processed before any of its child nodes is done.

In-order (LNR)

1. Check if the current node is empty or null.
2. Traverse the left subtree by recursively calling the in-order function.
3. Display the data part of the root (or current node).
4. Traverse the right subtree by recursively calling the in-order function.

In a binary search tree, in-order traversal retrieves data in sorted order.[4]



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

Out-order (RNL)

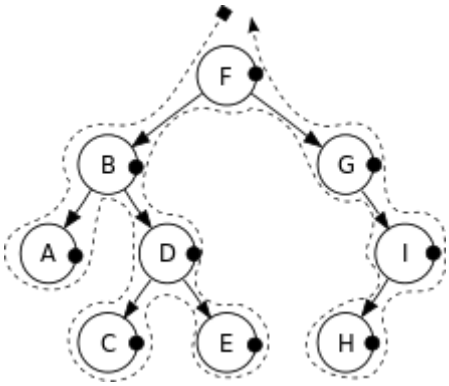
1. Check if the current node is empty or null.
2. Traverse the right subtree by recursively calling the out-order function.
3. Display the data part of the root (or current node).
4. Traverse the left subtree by recursively calling the out-order function.

In a binary search tree, out-order traversal retrieves data in reverse sorted order.

Post-order (LRN)

1. Check if the current node is empty or null.
2. Traverse the left subtree by recursively calling the post-order function.
3. Traverse the right subtree by recursively calling the post-order function.
4. Display the data part of the root (or current node).

The trace of a traversal is called a sequentialisation of the tree. The traversal trace is a list of each visited root. 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.[5]



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

Generic tree

To traverse any tree with depth-first search, perform the following operations recursively at each node:

1. Perform pre-order operation.
2. For each *i* from 1 to the number of children do:
    1. Visit *i*-th, if present.
    2. Perform in-order operation.
3. Perform post-order operation.

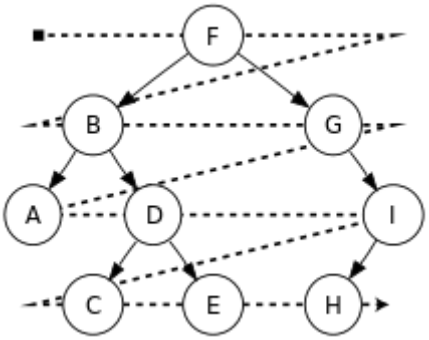
Depending on the problem at hand, the pre-order, in-order or post-order operations may be void, or you may only want to visit a specific child, so these operations are optional. Also, in practice more than one of pre-order, in-order and post-order operations may be required. For example, when inserting 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 search

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* (BFS), as the search tree is broadened as much as possible on each depth before 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.



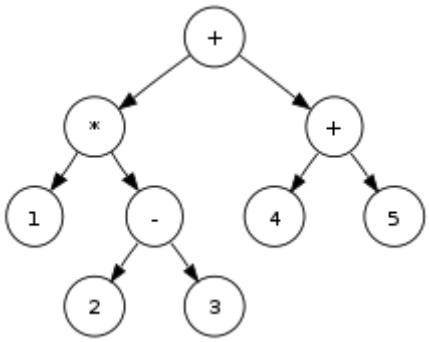
Level-order: F, B, G, A, D, I, C, E, H.

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. For example, traversing the depicted arithmetic expression in pre-order yields "+ * 1 - 2 3 + 4 5".

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 (Reverse Polish notation) of a binary tree. Traversing the depicted arithmetic expression in post-order yields "1 2 3 - * 4 5 + +".



Tree representing the arithmetic expression (1*(2-3))+(4+5)

Implementations

Depth-first search

Pre-order

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

```
iterativePreorder(node)
  if (node = null)
    return
  s ← empty stack
  s.push(node)
  while (not s.isEmpty())
    node ← s.pop()
    visit(node)
    //right child is pushed first so that left is processed first
    if (node.right ≠ null)
      s.push(node.right)
    if (node.left ≠ null)
      s.push(node.left)
```

In-order

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

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

Post-order

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

```
iterativePostorder(node)
  s ← empty stack
  lastNodeVisited ← null
  while (not s.isEmpty() or node ≠ null)
    if (node ≠ null)
      s.push(node)
      node ← node.left
    else
      peekNode ← s.peek()
      // if right child exists and traversing node
      // from left child, then move right
      if (peekNode.right ≠ null and lastNodeVisited ≠ peekNode.right)
        node ← peekNode.right
      else
        visit(peekNode)
        lastNodeVisited ← s.pop()
```

All the above implementations require stack space proportional to the height of the tree which is a call stack for the recursive and a parent stack for the iterative ones. In a poorly balanced tree, this can be considerable. With the iterative implementations 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:[6]

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 search

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.isEmpty())
    node ← q.dequeue()
```

```
    visit(node)
    if (node.left ≠ null)
      q.enqueue(node.left)
    if (node.right ≠ null)
      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 eventually. For infinite trees, simple algorithms often fail this. For example, given a binary tree of infinite depth, a depth-first search will go down one side (by convention the left side) of the tree, never visiting the rest, and indeed an in-order or post-order traversal 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 has infinitely many children, and each of these children has two children, a depth-first search 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 search 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 search of the depth 2 tree above will take $\omega \cdot 2$ steps: ω for the first level, and then another ω 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 (), the children of the root (1), (2), …, the grandchildren (1, 1), (1, 2), …, (2, 1), (2, 2), …, 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 $n \geq 1$), 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 search: replace the "down" edges connecting a parent node to its second and later children with "right" edges from the first child to the second child, from the second child to the third child, etc. Thus at each step one can either go down (append a (, 1) to the end) or go right (add one 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 one) 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. "Lecture 8, Tree Traversal" (http://webdocs.cs.ualberta.ca/~holte/T26/tree-traversal.html). Retrieved 2 May 2015.
2. http://www.cise.ufl.edu/~sahni/cop3530/slides/lec216.pdf
3. "Preorder Traversal Algorithm" (http://www.programmerinterview.com/index.php/data-structures/preorder-traversal-algorithm/). Retrieved 2 May 2015.
4. Wittman, Todd. "Tree Traversal" (https://web.archive.org/web/20150213195803/http://www.math.ucla.edu/~wittman/10b.1.10w/Lectures/Lec18.pdf) (PDF). *UCLA Math*. Archived from the original (http://www.math.ucla.edu/~wittman/10b.1.10w/Lectures/Lec18.pdf) (PDF) on February 13, 2015. Retrieved January 2, 2016.
5. "Algorithms, Which combinations of pre-, post- and in-order sequentialisation are unique?, Computer Science Stack Exchange" (http://cs.stackexchange.com/questions/439/which-combinations-of-pre-post-and-in-order-sequentialisation-are-unique). Retrieved 2 May 2015.
6. Morris, Joseph M. (1979). "Traversing binary trees simply and cheaply". *Information Processing Letters*. **9** (5). doi:10.1016/0020-0190(79)90068-1 (https://doi.org/10.1016%2F0020-0190%2879%2990068-1).

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

- Storing Hierarchical Data in a Database (http://www.sitepoint.com/hierarchical-data-database/) with traversal examples in PHP
- Managing Hierarchical Data in MySQL (https://web.archive.org/web/20110606032941/http://dev.mysql.com/tech-resources/articles/hierarchical-data.html)
- Working with Graphs in MySQL (http://www.artfulsoftware.com/mysqlbook/sampler/mysqled1ch20.html)
- 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)
