# **Tree Traversal**

* Unlike linked lists, one-dimensional arrays, and other linear data structures, which are traversed in linear order, trees can be traversed in multiple ways.

## **Depth–First Search (DFS)**

**Depth-First Search**

* Depth–first search (DFS) is an algorithm for traversing or searching tree or graph data structures.
* One starts at the root (selecting some arbitrary node as the root for a graph) and explore as far as possible along each branch before backtracking.
* A tree is an undirected graph in which any two vertices are connected by exactly one path.
* In other words, any acyclic connected graph is a tree.

**Traversal Methods**

* For a tree, we have the following traversal methods:

1. **Preorder**
   1. The current node is visited before either of the children
   2. Then the left subtree is traversed
   3. Then the right subtree is traversed
2. **Inorder** 
   1. The left subtree is traversed
   2. Then the current node is visited
   3. Then the right subtree is traversed
3. **Postorder**
   1. The left subtree is traversed
   2. The right subtree is traversed
   3. The current node is visited after both the left subtree and the right subtrees

**Runtime**

* Each of these traversals visits every node in a binary tree exactly once.
* Thus, *n* visits occur for a tree of *n* nodes.
* Each visit performs the same operations on each node, independently of *n*, so it must be O(1).
* **Thus, each traversal is O(*n*).**



## **Recursive Algorithms**

* The general form for the three types of binary tree traversal have recursive algorithms that look like this:

// Use this for preorder traversal.

void **preorder**(Node n)

{

if (n == null) return;

**preorder**(n->item);

traverse(n->leftChild);

traverse(n->rightChild);

}

// Use this for inorder traversal.

void **inorder**(Node n)

{

if (n == null) return;

traverse(n->leftChild);

**inorder**(n->item);

traverse(n->rightChild);

}

// Use this for postorder traversal.

void **postorder**(Node n)

{

if (n == null) return;

traverse(n->leftChild);

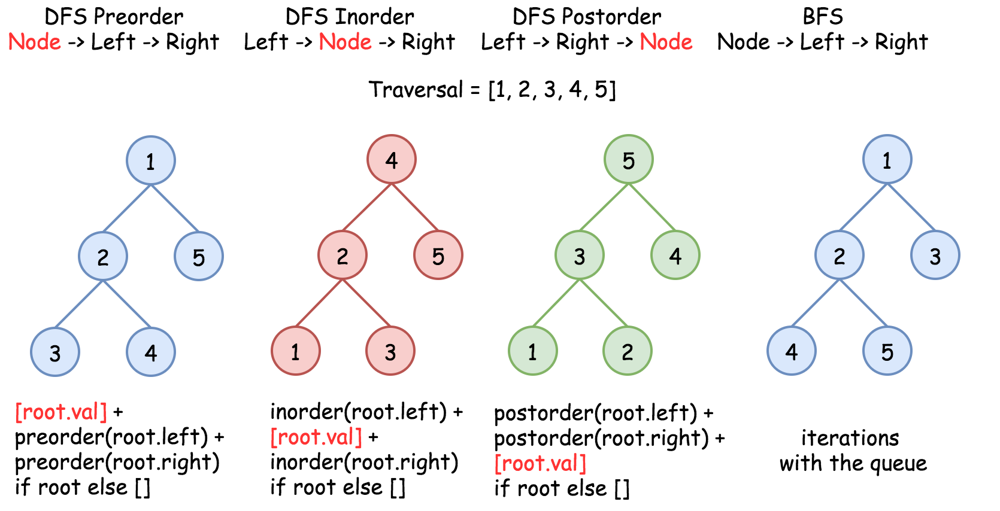
traverse(n->rightChild);

**postorder**(n->item);

}

Diagram

Description automatically generated with low confidence



## **Iterative Algorithms**

* Any recursive algorithm can be made into an iterative one.
* In recursive DFS traversals of a binary tree, we have three basic elements to traverse:
  + root
  + left subtree
  + right subtree
* Each traversal process nodes in a different order using recursion.
  + Preorder: root, left, right
  + Inorder: left, root, right
  + Postorder: left, right, root
* We can mimic the stack trace of a recursive algorithm by using an actual Stack data structure in an iterative one.
* Both preorder and inorder traversals are tail-recursive i.e. there is no extra code operation after the final recursive call. So implementation using stack is simple and easy to understand.
* But postorder traversal is non-tail recursive because there is an extra operation after the last recursive calls i.e. we process the root node.
* So an implementation of postorder using a stack would be a little tricky.
* But if we follow the basic order of processing nodes, it can be easy to visualize.
* To simulate the recursive traversal into an iterative traversal, we need to understand the flow of recursive calls in DFS traversal. We visit each node three times:

1. **First time:** when recursion visits the node coming from the top.

In preorder traversal, we process the node at this stage.

1. **Second time:** when recursion backtracks from the left child after visiting all the nodes in the left subtree.

In inorder traversal, we process the node after all the nodes in the left subtree are visited.

1. **Third time:** when recursion backtracks from the right child after visiting all the nodes in the right subtree.

In postorder traversal, we process the node after all the nodes in the left and right subtrees are visited.

* This matches the recursive traversal algorithm:

// Use this for postorder traversal.

void **traverse**(Node n)

{

if (n == null) return;

**preorder**(n->item);

traverse(n->leftChild);

**inorder**(n->item);

traverse(n->rightChild);

**postorder**(n->item);

}

* Here is the pseudocode for the algorithm:

1. Create an empty stack S.
2. Initialize current node as root
3. Push the current node to S and set current = current->left until current is NULL
4. If current is NULL and stack is not empty then
   1. Pop the top item from stack.
   2. Print the popped item, set current = popped\_item->right
   3. Go to step 3.
5. If current is NULL and stack is empty then we are done.\

1

/ \

2 3

/ \

4 5

Step 1 Creates an empty stack: S = NULL

Step 2 sets current as address of root: current -> 1

Step 3 Pushes the current node and set current = current->left

until current is NULL

current -> 1

push 1: Stack S -> 1

current -> 2

push 2: Stack S -> 2, 1

current -> 4

push 4: Stack S -> 4, 2, 1

current = NULL

Step 4 pops from S

a) Pop 4: Stack S -> 2, 1

b) print "4"

c) current = NULL /\*right of 4 \*/ and go to step 3

Since current is NULL step 3 doesn't do anything.

Step 4 pops again.

a) Pop 2: Stack S -> 1

b) print "2"

c) current -> 5/\*right of 2 \*/ and go to step 3

Step 3 pushes 5 to stack and makes current NULL

Stack S -> 5, 1

current = NULL

Step 4 pops from S

a) Pop 5: Stack S -> 1

b) print "5"

c) current = NULL /\*right of 5 \*/ and go to step 3

Since current is NULL step 3 doesn't do anything

Step 4 pops again.

a) Pop 1: Stack S -> NULL

b) print "1"

c) current -> 3 /\*right of 1 \*/

Step 3 pushes 3 to stack and makes current NULL

Stack S -> 3

current = NULL

Step 4 pops from S

a) Pop 3: Stack S -> NULL

b) print "3"

c) current = NULL /\*right of 3 \*/

Traversal is done now as stack S is empty and current is NULL.

**Efficiency**

* What is the efficiency of traversing a tree with n nodes?
* Each node is visited exactly once and the amount of work done per visit is generally O(1), so the total traversal is time O(n).
* Therefore, preorder, inorder, and postorder all take O(n) time to complete become each traversal algorithm touches each node exactly one time.

**Expression Trees**

* For an expression tree, the three types of traversal correspond to outputting the expression in prefix, infix, and postfix notation.
* Prefix and postfix notation (also known as Polish and reverse Polish notation) are useful, since they never require parentheses to denote the order of the operations.
* Traversing our tree for the expression above yields:

**Prefix**: \* 5 - + \* 3 4 / 6 2 + 8 2  
**Postfix**: 5 3 4 \* 6 2 / + 8 2 + - \*

**Infix**: (5 \* (((3 \* 4) + (6 / 2)) – (8 + 2)))

* Infix notation is ugly, since it needs all of the parentheses to denote the order of operations!
* Outputting the infix notation actually requires us to visit the operator nodes in prefix, postfix, and infix order. (The prefix and postfix visits output the parentheses.)