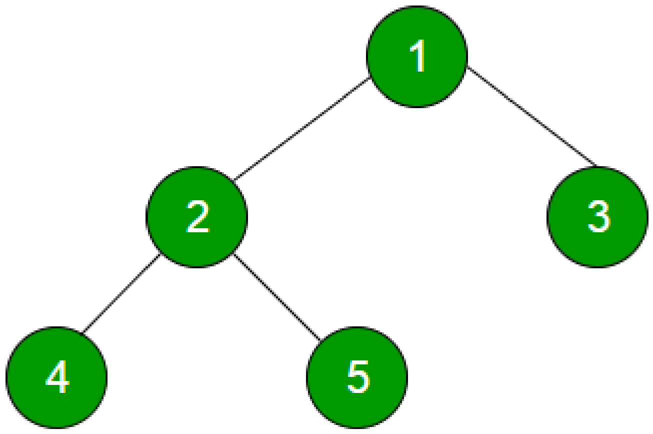
Tree Traversals

Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways. Following are the generally used ways for traversing trees.



Two ways of Traversal

1. **Breadth First or Level Order Traversal:** `1 2 3 4 5`

   (a) **Recursive approach**

   > *Time Complexity:* O(n^2) in worst case. For a skewed tree, printLevel() takes O(n) time where n is the number of nodes in the skewed tree. So time complexity of printLevelOrder() is O(n) + O(n-1) + O(n-2) + .. + O(1) which is O(n^2).

   (b) Iterative approach

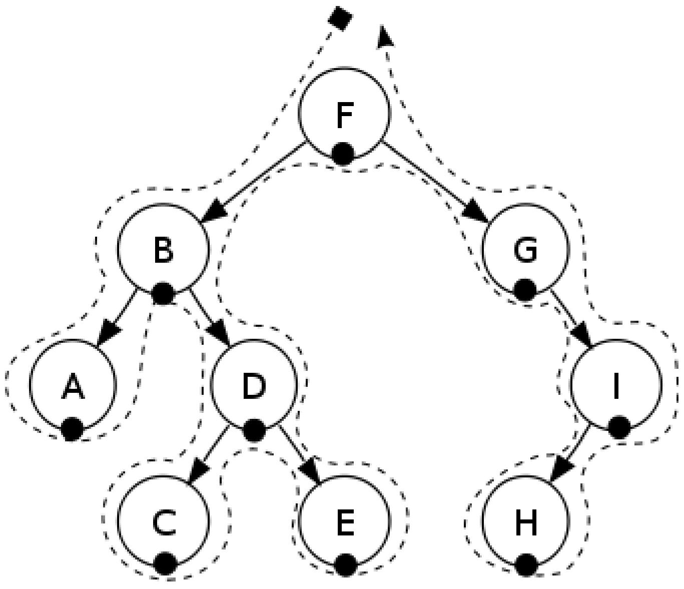
   > *Time Complexity:* O(n) where n is number of nodes in the binary tree

2. **Depth First Traversals:**

(a) In-order (Left, Root, Right) : `4 2 5 1 3`

```
INORDER-TREE-WALK(x)
1   if x != null
2       INORDER-TREE-WALK(x.left)
3       print x.key
4       INORDER-TREE-WALK(x.right)
```

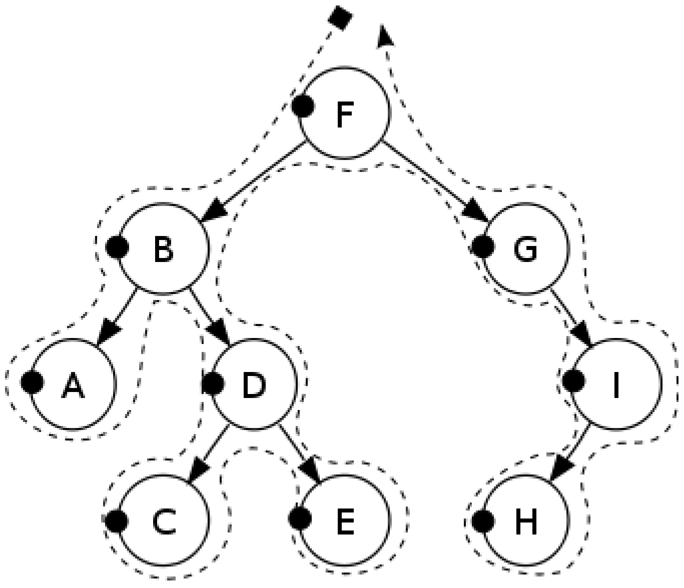
In case of binary search trees (BST), In-order traversal gives nodes in non-decreasing order.



(b) Pre-order (Root, Left, Right) : `1 2 4 5 3`

```
Algorithm Pre-order(root)
   1. print root.key
   2. Traverse the left subtree, i.e., call Pre-order(root.left)
   3. Traverse the right subtree, i.e., call Pre-order(root.right)
```

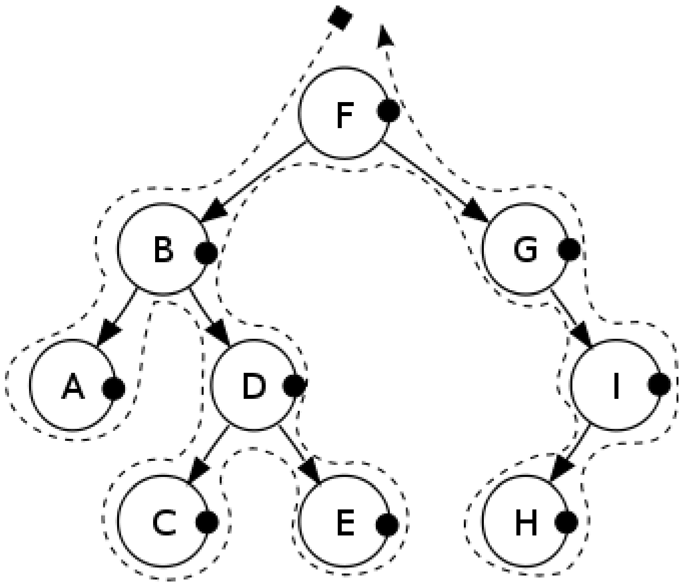
Pre-order traversal is used to create a copy of the tree. Pre-order traversal is also used to get prefix expression on of an expression tree.



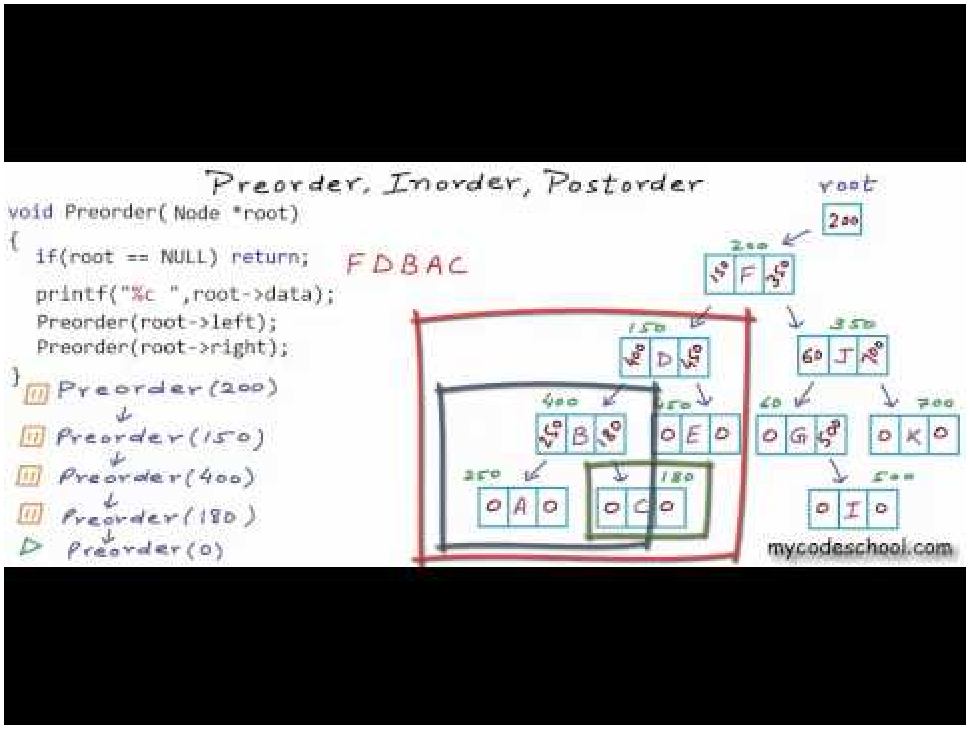
(c) Post-order (Left, Right, Root) : `4 5 2 3 1`

```
Algorithm Postorder(root)
    1. Traverse the left subtree, i.e., call Postorder(root.left)
    2. Traverse the right subtree, i.e., call Postorder(root.right)
    3. print root.key
```

Post-order traversal is used to delete the tree (not in java). Post-order traversal is also useful to get the postfix expression of an expression tree.



Video explanation of DFT:



```
void Preorder( Node *root)
{
  if(root == NULL) return;
  printf("%c ",root->data);
  Preorder(root->left);
  Preorder(root->right);
}
```

Time Complexity: O(n)

Complexity function T(n): for all problem where tree traversal is involved — can be defined as:

```
T(n) = T(k) + T(n – k – 1) + c
```
Where k is the number of nodes on one side of root and n-k-1 on the other side.

Iterative DFT approach

1. **Iterative In-order Traversal**

   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
     a) Pop the top item from stack.
     b) Print the popped item, set current = popped_item->right
     c) Go to step 3
5) If current is NULL and stack is empty then algorithm completes
```

2. **Iterative Pre-order Traversal**

Algorithm:

```
1) Create an empty stack S and push root node to stack.
2) Do following while S is not empty.
    a) Pop an item from stack and print it.
    b) Push right child of popped item to stack
    c) Push left child of popped item to stack
```

3. **Iterative Post-order Traversal**

Iterative post-order traversal is more complex than the other two traversals (due to its nature of non-tail recursion i.e. there is statements after the final recursive call to itself). Post-order traversal can easily be done using two stacks. The idea is to push reverse post-order traversal to a stack. Once we have the reversed post-order traversal in a stack, we can just pop all items one by one from the stack and print them; this order of printing will be in post-order because of the LIFO property of stacks. To get reversed post-order elements in a stack – the second stack is used for this purpose. If take a closer look at this sequence, we can observe that this sequence is very similar to the pre-order traversal, the only difference is that the right child is visited before left child, and therefore the sequence is "root-right-left" instead of "root-left-right".

So, we can do something like iterative pre-order traversal with the following differences:
a) Instead of printing an item, we push it to a stack.
b) We push the left subtree before the right subtree.

Following is the complete algorithm. After step 2, we get the reverse of a post-order traversal in the second stack. We use the first stack to get the correct order.

Algorithm:

```
1. Push root to first stack.
2. Loop while first stack is not empty
   a) Pop a node from first stack and push it to second stack
   b) Push left and right children of the popped node to first stack
3. Print contents of second stack
```

Step wise walk-through:

```
        1
      /   \
     2      3
   /  \    /  \
  4    5  6    7

1. Push (1) to 1st stack
      1st stack: 1
      2nd stack: Empty

2. Pop (1) from 1st stack and push it to 2nd stack.
   Push left and right children of (1) to first stack
      1st stack: 2, 3
      2nd stack: 1

3. Pop (3) from 1st stack and push it to 2nd stack.
   Push left and right children of (3) to first stack
      1st stack: 2, 6, 7
      2nd stack: 1, 3

4. Pop (7) from 1st stack and push it to 2nd stack.
      1st stack: 2, 6
      2nd stack: 1, 3, 7

5. Pop (6) from 1st stack and push it to 2nd stack.
      1st stack: 2
      2nd stack: 1, 3, 7, 6

6. Pop (2) from 1st stack and push it to 2nd stack.
   Push left and right children of (2) to 1st stack
      1st stack: 4, 5
      2nd stack: 1, 3, 7, 6, 2

7. Pop (5) from 1st stack and push it to 2nd stack.
      1st stack: 4
      2nd stack: 1, 3, 7, 6, 2, 5

8. Pop (4) from 1st stack and push it to 2nd stack.
      1st stack: Empty
      2nd stack: 1, 3, 7, 6, 2, 5, 4

The algorithm stops here since there are no more items in the first stack.
Observe that the contents of second stack are in post-order fashion. Print them.
```