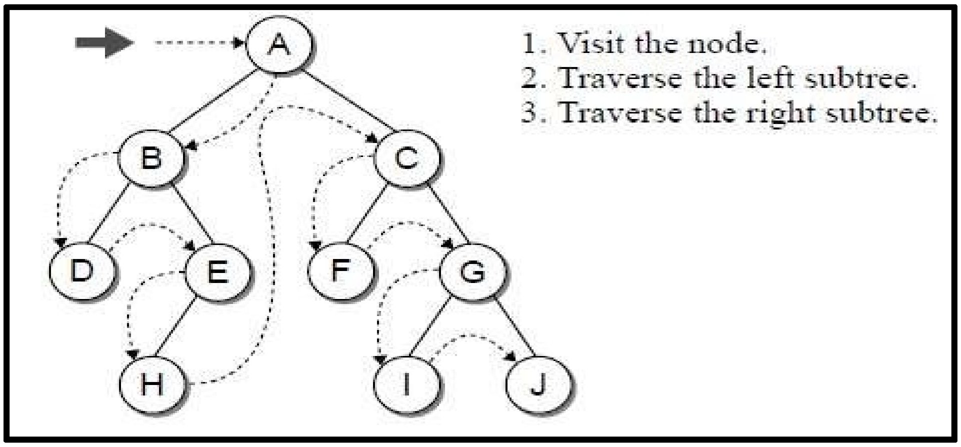
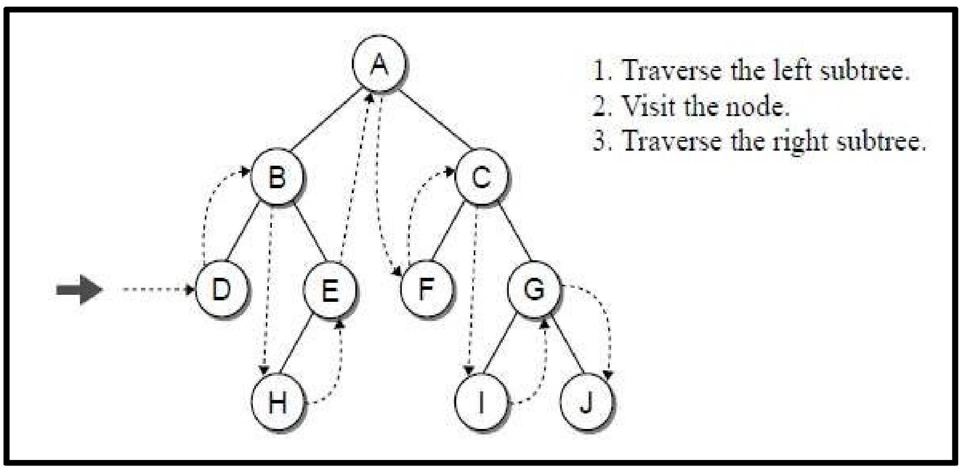
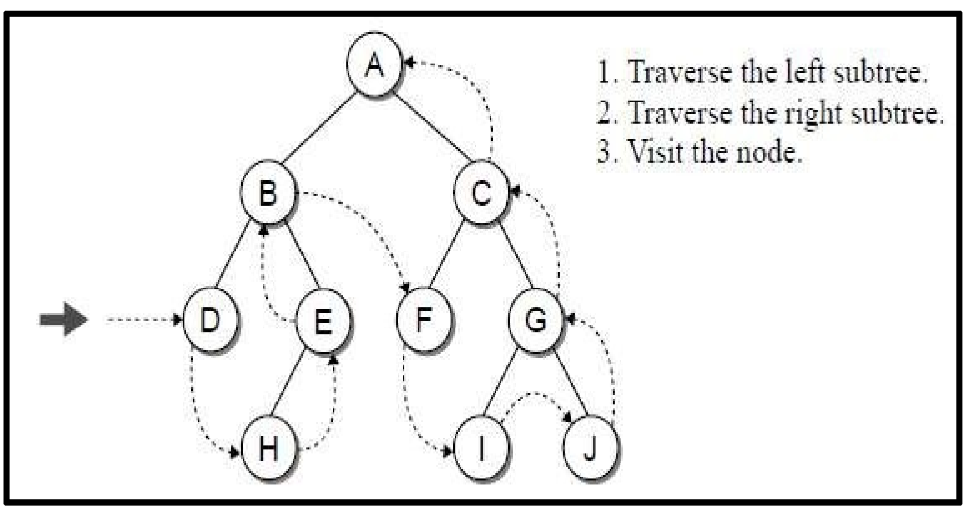
Week 12	Depth First Traversal. Breadth First Traversal. Tree Application: Expression Trees.

Depth-First Traversal

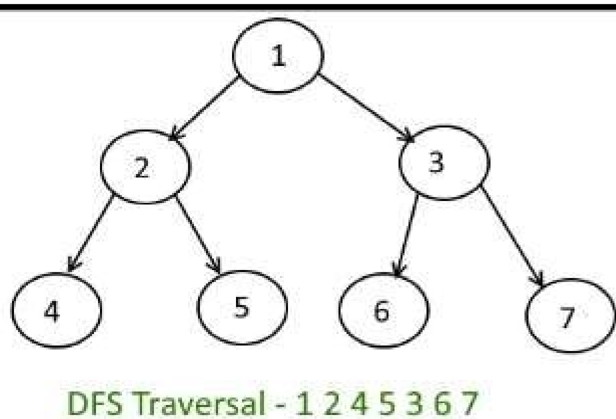
- The pre-order, in-order, and post-order traversals are all examples of a Depth First Traversal.
- That is, the nodes are traversed deeper in the tree before returning to higher-level nodes.
- Figure shows the ordering of the nodes in a Depth First Traversal of a following Binary Tree.







- Stack data structure is used to implement the Depth First Traversal.
- First add the add root to the Stack.
- Pop out an element from Stack, visit it and add its right and left children to stack.
- Pop out an element, visit it and add its children.
- Repeat the above two steps until the Stack is empty.
- **Example:**



Print						
1						
1	2					
1	2	4				
1	2	4	5			
1	2	4	5	3		
1	2	4	5	3	6	
1	2	4	5	3	6	7

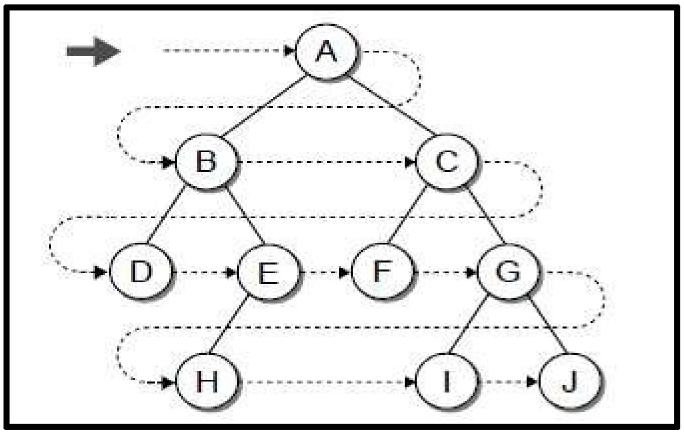
Stack		
1		
2	3	
4	5	3
5	3	
3		
6	7	
7		
-		

Python Function for Depth First Traversal

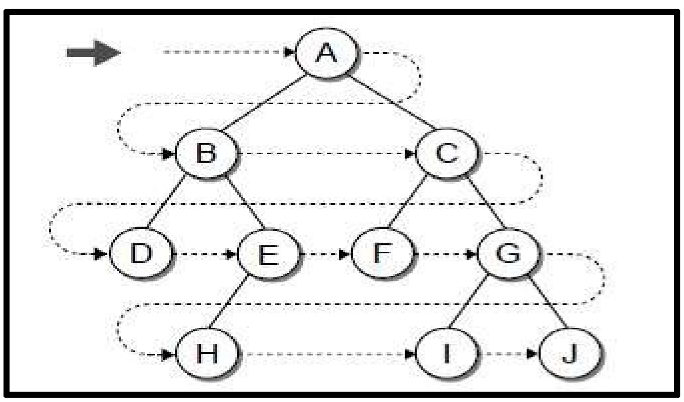
```
def DFS(root):
    S = Stack()
    S.push(root)
    while S.isEmpty() != True:
      node=S.pop()
      print(node.data,end="\t")
      if node.right is not None:
         S.push(node.right)
      if node.left is not None:
         S.push(node.left)
```

Breadth-First Traversal

- Another type of traversal that can be performed on a binary tree is the Breadth First Traversal.
- In a Breadth First Traversal, the nodes are visited by level, from left to right.
- Figure shows the ordering of the nodes in a Breadth First Traversal of a following Binary Tree.



- Queue data structure is used to implement the Breadth First Traversal.
- The process starts by adding root node to the queue.
- Next, we visit the node at the front end of the queue, remove it and insert all its child nodes.
- This process is repeated till queue becomes empty.
- **Example:**



Queue

A					
B	C				
C	D	E			
D					
E	F	G			
F	G	H			
G	H				
H	I	J			
I	J				
J					

BFS Traversal: A B C D E F G H I J

Python Function for Breadth First Traversal

```
def BFS(root):
    Q = Queue()
    Q.Enqueue(root)
    while Q.IsEmpty() != True:
        node=Q.Dequeue()
        print(node.data,end="\t")
        if node.left is not None:
            Q.Enqueue(node.left)
        if node.right is not None:
            Q.Enqueue(node.right)
```

Program #1: Python Program to implement Depth First Traversal.

```
class Stack:
   def__init_(self):
      self.items=list()

   def push(self,value):
      self.items.append(value)

   def pop(self):
      if len(self.items) != 0:
         return self.items.pop()

   def isEmpty(self):
      if len(self.items) == 0:
         return True
      else:
         return False

class Node:
   def__init_(self,value):
      self.data = value
      self.left = None
      self.right =None

class BST:
   def__init_(self):
      self.root=None

   def insert(self,value):
      newNode=Node(value)
      if self.root is None:
         self.root = newNode
      else:
       curNode = self.root
       while curNode is not None:
         if value < curNode.data:
            if curNode.left is None:
               curNode.left=newNode
```

```
                    break
                else:
                    curNode = curNode.left
            else:
                if curNode.right is None:
                    curNode.right=newNode
                    break
                else:
                    curNode=curNode.right

def DFS(root):
        S = Stack()
        S.push(root)
        while S.isEmpty() != True:
            node=S.pop()
            print(node.data,end="\t")
            if node.right is not None:
                S.push(node.right)
            if node.left is not None:
                 S.push(node.left)

BT = BST()

ls = [25,10,35,20,5,30,40]
for i in ls:
    BT.insert(i)
print("DFS Traversal")
DFS(BT.root)
```

Output #1:

```
DFS Traversal
25	10	5	20	35	30	40
```

Program #2: Python Program to implement Breadth First Traversal.

```
class Queue:
  def__init_(self):
    self.items=list()

  def enqueue(self,value):
    self.items.append(value)

  def dequeue(self):
    if len(self.items) != 0
      return self.items.pop(0)

  def isEmpty(self):
    if len(self.items) == 0:
      return True
    else:
      return False

class Node:
  def__init_(self,value):
    self.data = value
    self.left = None
    self.right =None

class BST:
  def__init_(self):
    self.root=None

  def insert(self,value):
    newNode=Node(value)
    if self.root is None:
      self.root = newNode
    else:
     curNode = self.root
     while curNode is not None:
      if value < curNode.data:
        if curNode.left is None:
          curNode.left=newNode
```

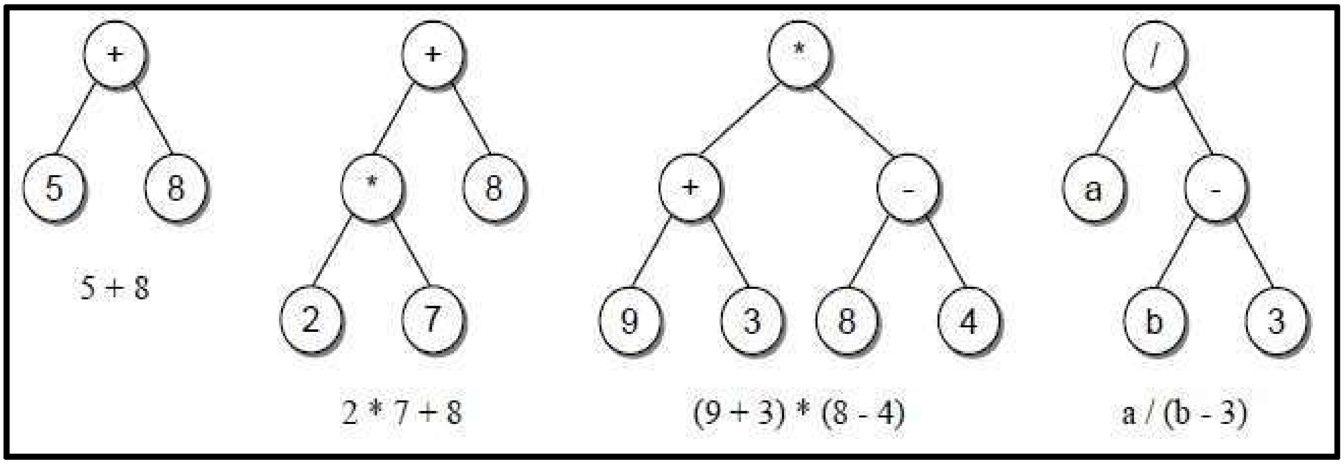
```
                break
            else:
                curNode = curNode.left
        else:
            if curNode.right is None:
                curNode.right=newNode
                break
            else:
                curNode=curNode.right

def BFS(root):
    Q = Queue()
    Q.enqueue(root)
    while Q.isEmpty() != True:
        node=Q.dequeue()
        print(node.data,end="\t")
        if node.left is not None:
            Q.enqueue(node.left)
        if node.right is not None:
            Q.enqueue(node.right)

BT = BST()

ls = [25,10,35,20,5,30,40]
for i in ls:
    BT.insert(i)

print("BFS Traversal")
BFS(BT.root)
```

Output #1:

```
BFS Traversal
25	10	35	5	20	30	40
```

Expression Trees

- Arithmetic expressions such as (9+3)*(8-4) can be represented using an expression tree.
- An expression tree is a binary tree in which the operators are stored in the interior nodes and the operands (the variables or constant values) are stored in the leaves.
- Once constructed, an expression tree can be used to evaluate the expression or for converting an infix expression to either prefix or postfix notation.



- Sample Arithmetic Expression Trees:

Construction of Expression Tree:

To construct an expression tree manually, follow these steps:

* Start with the given arithmetic expression in infix notation.
* Convert the infix expression to postfix notation. Manually apply the rules of operator precedence and associativity.
* Once you have the expression in postfix notation, begin constructing the expression tree.
  1. Initialize an empty stack to hold the nodes of the expression tree.
  2. Iterate through the postfix expression from left to right.
  3. For each symbol encountered:
  4. If the symbol is an operand (number), create a new node with the operand as its value and push it onto the stack.
  5. If the symbol is an operator:
  6. Create a new node with the operator as its value.
  7. Pop two nodes from the stack and assign them as the right and left children of the new node.
  8. Push the new node onto the stack.
  9. After processing all symbols in the postfix expression, the stack will contain only the root node of the expression tree.
  10. Pop the root node from the stack, and the construction of the expression tree is complete.

Let's construct an expression tree for the infix expression: (4 + 2) * 3 - 5

Convert infix to postfix: 4 2 + 3 * 5 -

Start constructing the expression tree:

- Symbol '4' is an operand, so create a node with value 4 and push it onto the stack.
  Stack: [4]

- Symbol '2' is an operand, so create a node with value 2 and push it onto the stack.
  Stack: [4, 2]

- Symbol '+' is an operator. Pop two nodes from the stack ('2' and '4') and create a new node with the operator '+' and assign the popped nodes as its right and left children respectively. Push the new node onto the stack.
  Stack: [+]

- Symbol '3' is an operand, so create a node with value 3 and push it onto the stack.
  Stack: [+ , 3]

- Symbol * is an operator. Pop two nodes from the stack ('3' and '+') and create a new node with the operator * and assign the popped nodes as its right and left children. Push the new node onto the stack.
  Stack: [*]

- Symbol '5' is an operand, so create a node with value 5 and push it onto the stack.
  Stack: [* , 5]

- Symbol '-' is an operator. Pop two nodes from the stack ('5' and '*') and create a new node with the operator '-' and assign the popped nodes as its right and left children. Push the new node onto the stack.

  Stack: [-]

```
        -
       / \
      *   5
     / \
    +   3
   / \
  4   2
```

The stack now contains the root node of the expression tree for the given infix expression

```
        -
       / \
      *   5
     / \
    +   3
   / \
  4   2
```