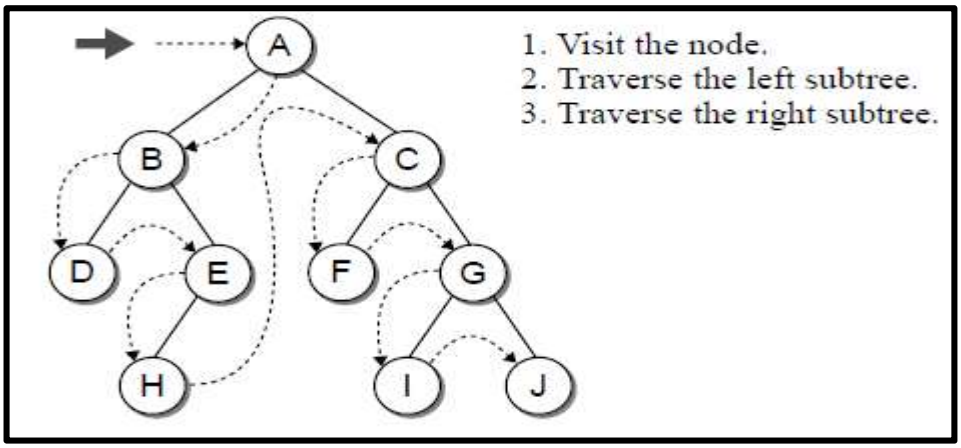
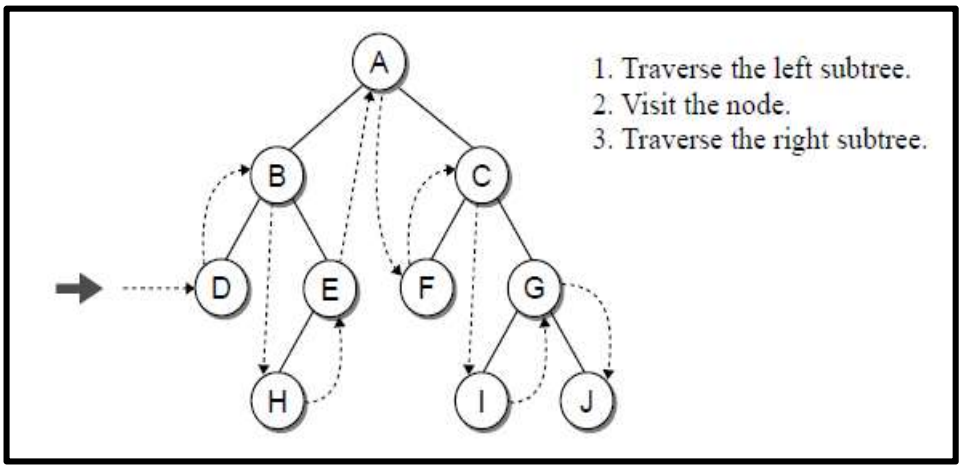
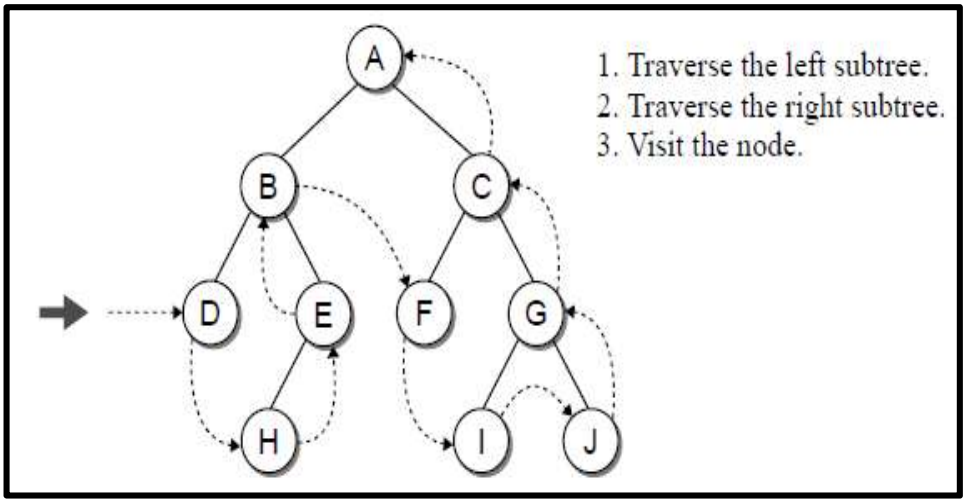
Week 12	Depth First Traversal. Breadth First Traversal. Tree Application: Expression Trees.

Depth-First Traversal

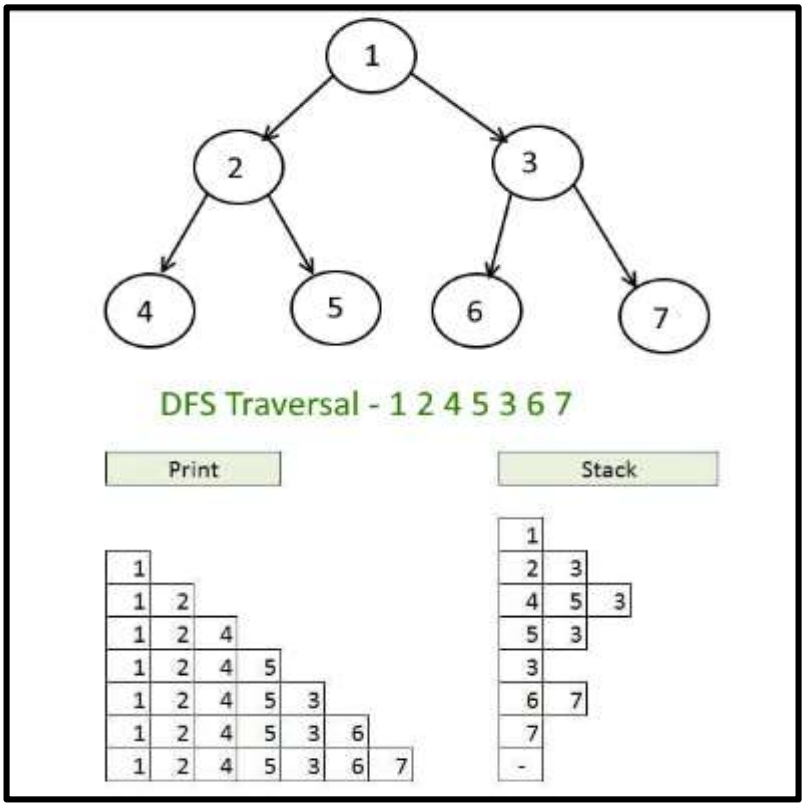
- The pre-order, in-order, and post-order traversals are all examples of a Depth First Traversal.
- That is, the nodes are traversed deeper in the tree before returning to higher-level nodes.
- Figure shows the ordering of the nodes in a Depth First Traversal of a following Binary Tree.







- 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:**

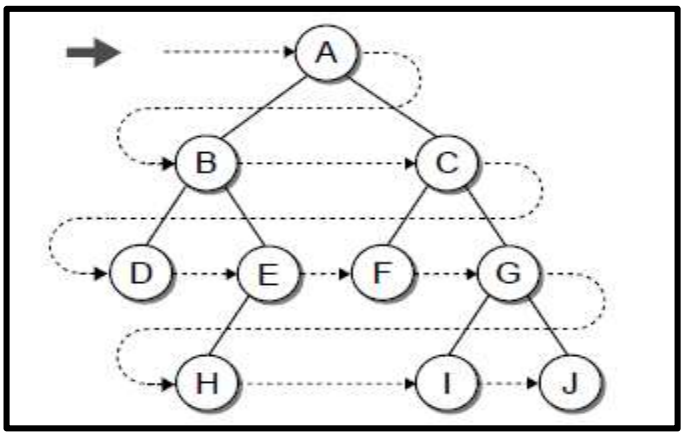


Python Function for Depth First Traversal

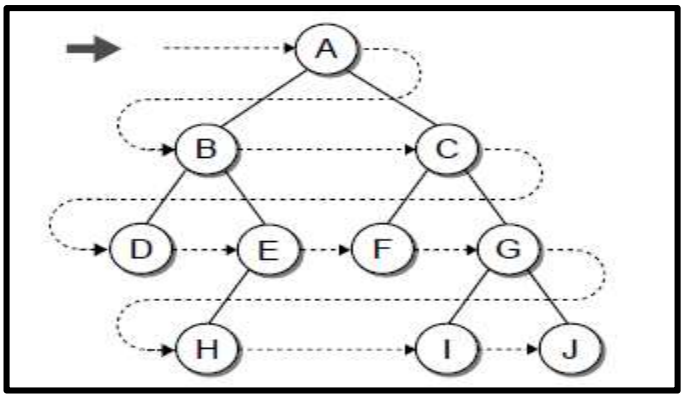
```
def DFS(root):
    S = Stack()
    S.push(root)
    while S.isEmpty() != True:
        node=S.pop()
        print(node.data,end="\t")
        if node.right is not None:
            S.push(node.right)
        if node.left is not None:
            S.push(node.left)
```

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					

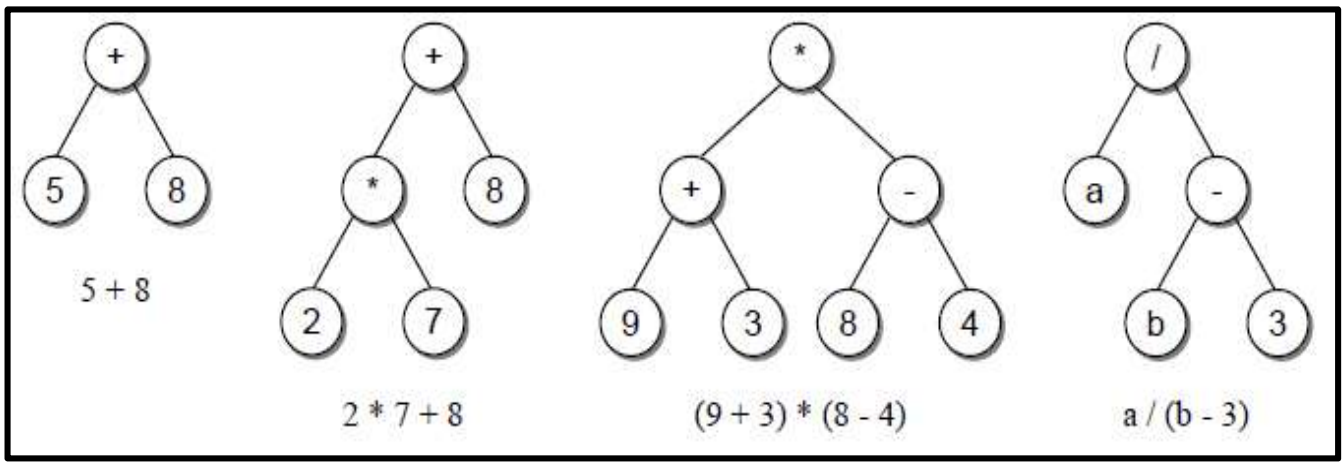
BFS Traversal: A B C D E F G H I J

Python Function for Breadth First Traversal

```
def BFS(root):
        Queue q
        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 :
```

Expression Trees

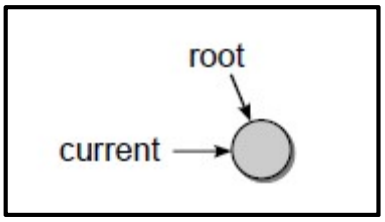
- Arithmetic expressions such as (9+3)*(8-4) can be represented using an expression tree.
- An expression tree is a binary tree in which the operators are stored in the interior nodes and the operands (the variables or constant values) are stored in the leaves.
- Once constructed, an expression tree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



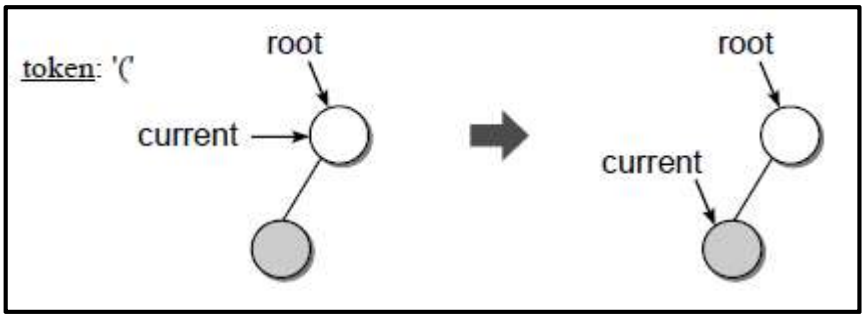
Construction of Expression Tree

- To construct Expression Tree, for simplicity, we assume the following:
  1. The expression is stored in string with no white space;
  2. The supplied expression is valid and fully parenthesized;
  3. Each operand will be a single-digit or single-letter variable; and
  4. The operators will consist of +, -, *, /, and %.

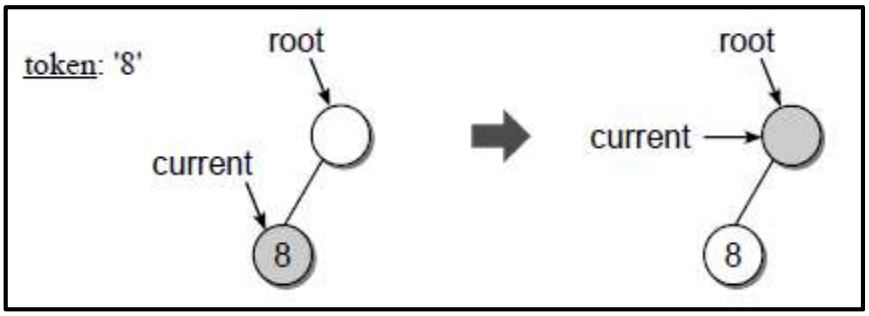
- An expression tree is constructed by accessing the individual tokens of the expression.
- The process starts with an empty root node set as the current node:



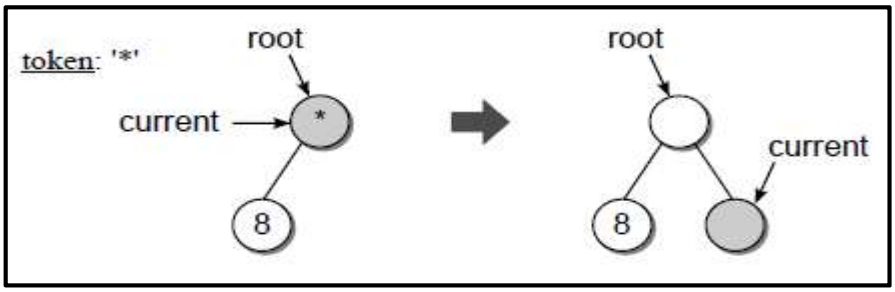
- Suppose we are building the tree for the expression (8*5).
- The First token is a left parenthesis. When a left parenthesis is encountered, a new node is created and linked into the tree as the left child of the current node. We then move down to the new node, making the left child the new current node.



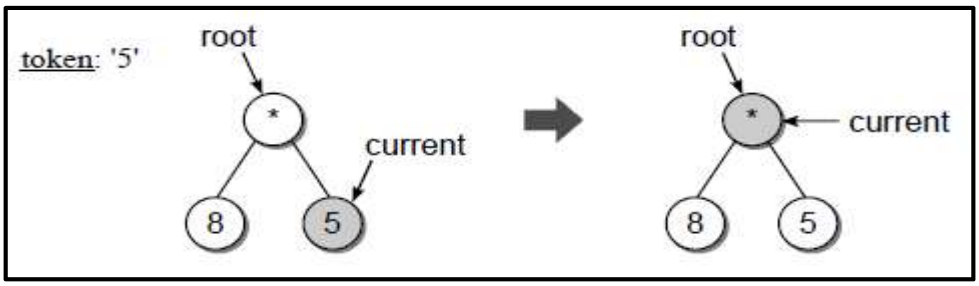
- The next token is the operand: 8. When an operand is encountered, the data value of the current node is set to contain the operand. We then move up to the parent of the current node.



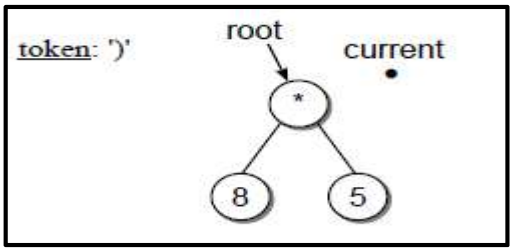
- Next comes the plus operator. When an operator is encountered, the data value of the current node is set to the operator. A new node is then created and linked into the tree as the right child of the current node. We move down to the new node.



- The second operand, 5, repeats the same action taken with the first operand.



- Finally, the right parenthesis is encountered and we move up to the parent of the current node. In this case, we have reached the end of the expression and the tree is complete.



Build the tree for the expression ((2*7)+8). The steps illustrated in the figure are described below:

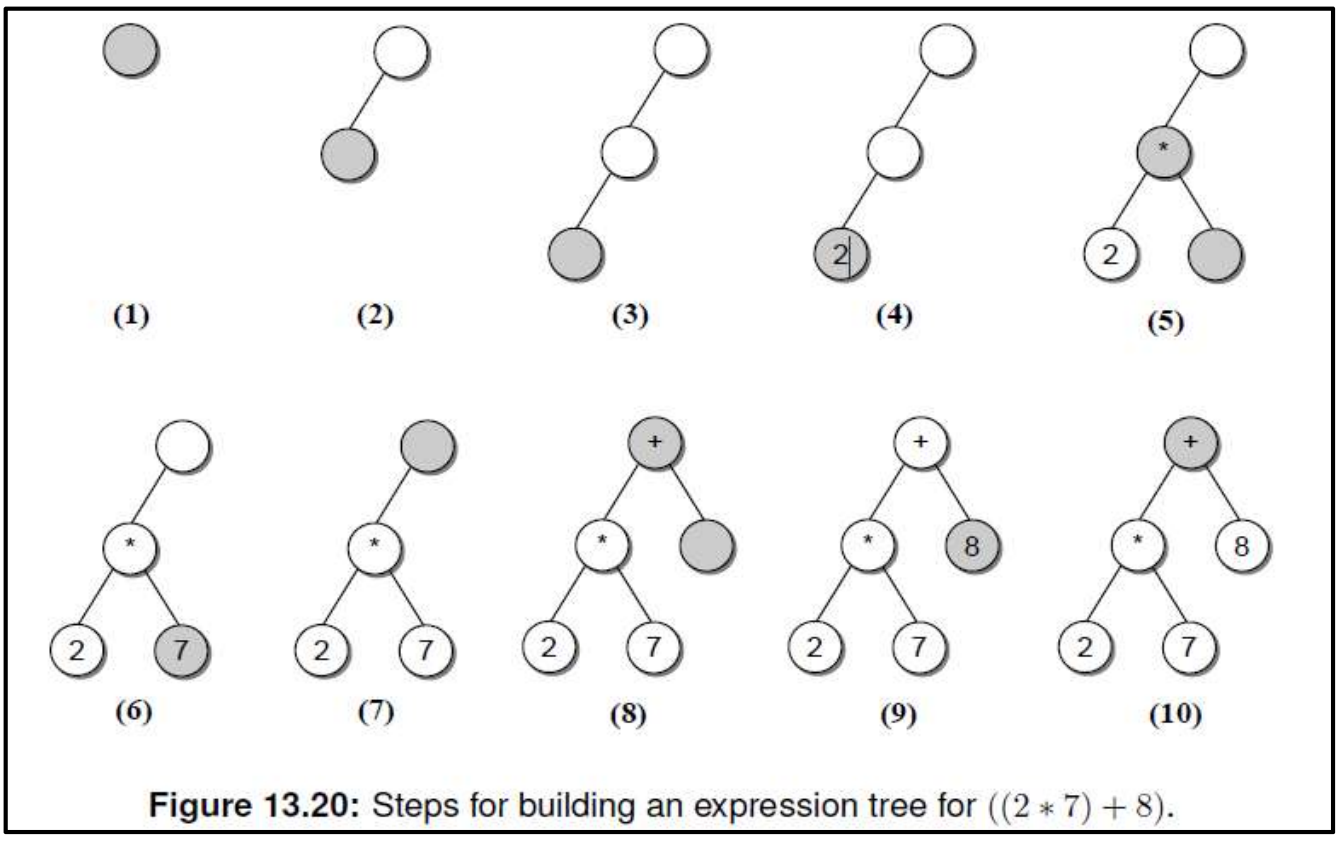


Figure 13.20: Steps for building an expression tree for $((2 * 7) + 8)$.

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
```

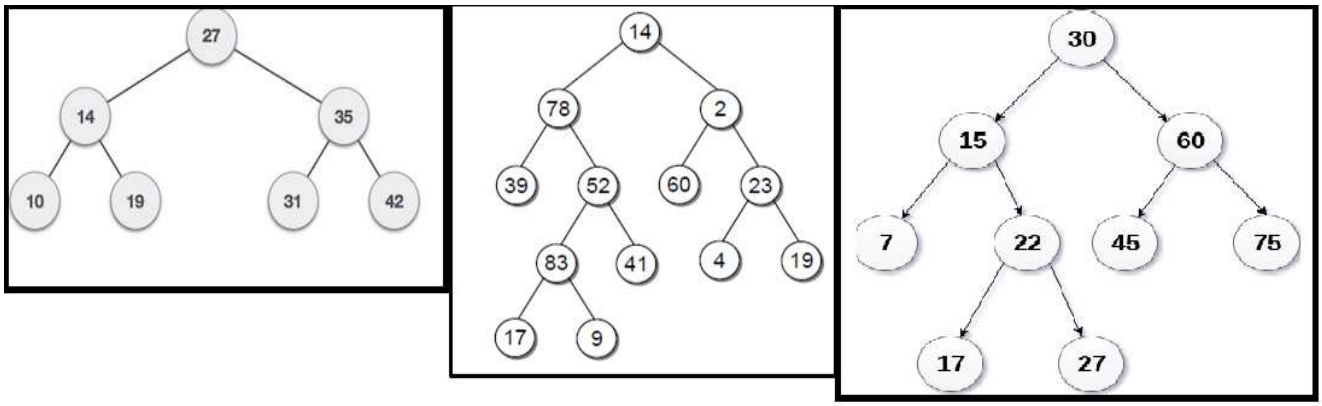
```
                break
            else:
                curNode = curNode.left
        else:
            if curNode.right is None:
                curNode.right=newNode
                break
            else:
                curNode=curNode.right

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
```

Assignment 12

1. **Consider the following binary tree:**

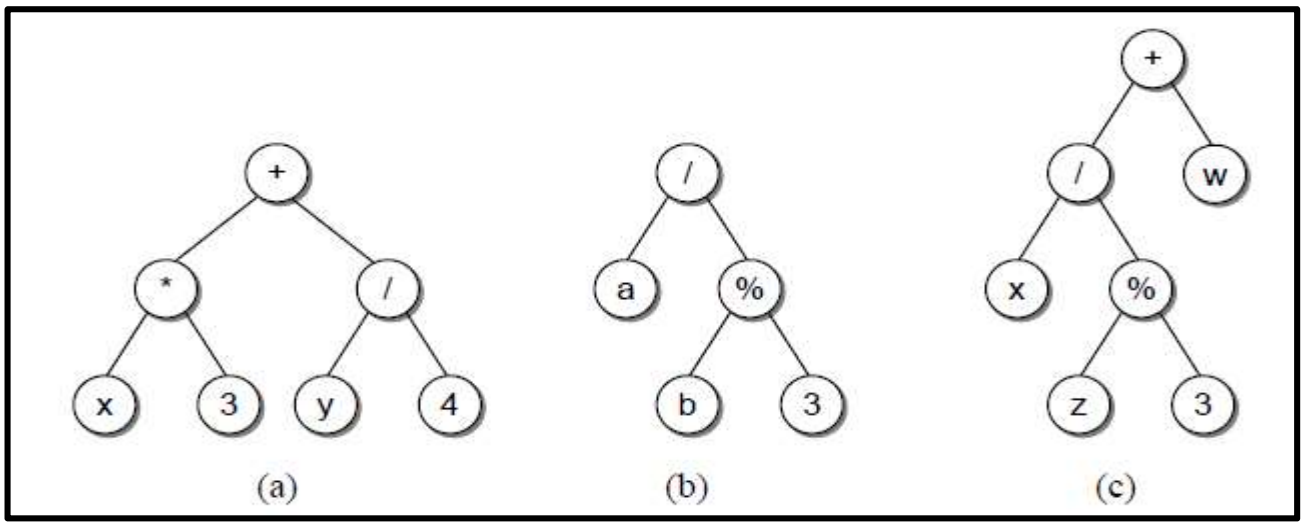


(a) Show the order the nodes will be visited in a:

i. Depth First Traversal

ii. Breadth First Traversal

2. **Determine the arithmetic expression represented by each of the following expression trees:**



3. **Build the expression tree for each of the following arithmetic expressions:**

(a) (A * B) / C

(b) A - (B * C) + D / E

(c) (X - Y) + (W * Z) / V

(d) V * W % X + Y - Z

(e) A / B * C - D + E