Tree Data Structure:

Introduction to Tree Data Structure
Terminologies of Tree
Binary Tree (BT)
BT Implementation
Binary Search Tree (BST)
BST Implementation
AVL Trees

==> if we want to represent the data in the form of Hierarchical relationship.
==> it is non-linear data structure
==> Parent & Child relationship
==> Best suitable for Search Operations
==> Ex: Binary Search Algorithm

L search : O(n)
B search : O(logn)

Tree is a finiate set of one or more nodes such that

1) There is specially designed node called as Root.
2) Remaining nodes are partitioned into n>=0 disjoin sets.

Tree Data Structure Terminologies

Root:
-----

unique node, having only out going edges.

Ex: A

Node:

Fundamental element of a tree, each node contains the following three fields.

1. data field (actual info)
2. left pointer pointing to the left child
3. right pointer pointing to the right child

Edges:

It is also a fundamental part of tree, used to connect two nodes.

AB, AC, AD, BE, BF, CG, DH, EI

Path:
-----

an ordered list of nodes that are connected by edges are called as path.

A to I ----> AB, BE, EI

leaf nodes:

The nodes which are not having any children are called as leaf nodes. i.e. without out going edges.

Ex:
I, F, G, H

height of the tree
----

height of the tree is sum of edges on the longest path between root and leaf node.

AI ---> AB, BE, EI
AF ---> AB, BF
AG ---> AC, CG
AH ---> AD, DH

height of tree: 3

level of the tree:

The level of node/tree is number of edges on the path from root to that node.

Ex:

```
      A ---------> 0
      B,C,D -----> 1
      E,F,G,H ---> 2
      I ---------> 3
```

parent:

Node is parent of all the children that are linked by out going edges.

Ex: A, B, C, D, E

Children:

Nodes that are having incoming edges are children.

B, C, D, E, F, G, H, I

Siblings:

Nodes in the tree that are children of same parent are called as siblings.

Ex:
      BCD, EF

degree of node:
----
total number of sub-trees attached to that node is called as degree of node

Ex:

```
      A ---> 3
      B ---> 2
      C ---> 1
      D ---> 1
      E ---> 1
      F ---> 0
```

degree of a tree:
----
max degree in the tree is called as degree of the tree

Ex:
      3

ancestor:
----
a node reachable through repeated moving from child to parent.

Ex:
      I ----> E, B, A

predessor:
----

while displaying the nodes in the tree, if node comes before another node that node is called as predessor.

Ex:

E is predessor of I

sucessor:
----

while displaying the nodes in the tree, if node comes after another node that node is called as successor.

Ex:

F is successor of I

Binary Tree (BT)

Binary tree is a type of tree in which each node has atmost two children (0,1,2) which are reffered as left child and right child.

In BT each node will have data field and two pointer field, pointing to the left and right sub trees.

BT implementation:

There are two ways are there to represent binary trees.

1. sequential representation.
2. linked list representation.

1. sequential representation.

In this sequential representation if a node is present at nth location.

```
Parent ----> N
left C ----> 2N+1
right C ---> 2N+2

Ex: [1, 2, 3, 4, 5, 6, 7]
     0  1  2  3  4  5  6

n=0 ---> l(0) = 2x0+1=0+1 ==> 2, r(0) = 2x0+2 = 2 ==> 3
n=1 ---> l(1) = 2x1+1=2+1===> 4, r(1) = 2x1+2=4 ==> 5
n=2 ---> l(2) = 2x2+1=5 ====> 6, r(2) = 2x2+2=6 ==> 7
```

2. linked list representation.
----

we can use double linked list to represent tree data structure in linked list.

binary tree traversals:
----

traversing means visiting each node once in a tree. it is also called as displaying each node present in the tree. the following are the three ways to display the elements in the Tree/BT/BST/AVL/RBT/B+.

```
Root---> D
Left --> L
Right -> R
```

1. Inorder ----> LDR ---> Left, Root, Right
2. Preorder ---> DLR ---> Root, Left, Right
3. Postorder --> LRD ---> Left, Right, Root

```
Ex1:
----
Inorder ----> BAC
Preorder ---> ABC
Postorder --> BCA

Ex2:
----
Inorder -----> DBEAC
Preorder ----> ABDEC
Postorder ---> DEBCA

Ex3:
----


Inorder ---> 4 2 5 1 6 3 7
Preorder --> 1 2 4 5 3 6 7
Postorder -> 4 5 2 6 7 3 1

creation of a BT using list
---------------------------
case1:
------
class BTree:

      class Node:

            def __init__(self,data,left=None,right=None):
                  self.data = data
                  self.left = left
                  self.right = right

      def __init__(self):
            self.root = None

      def createtree(self,L):
            self.root = self.createtree_util(L,0)

      def createtree_util(self,L,start):
            size = len(L)
            curr = self.Node(L[start])
            l = 2*start + 1
            r = 2*start + 2
            if l<size:
                  curr.left = self.createtree_util(L,l)
            if r<size:
                  curr.right = self.createtree_util(L,r)
            return curr
      def inorder(self):
            self.inorder_util(self.root)
            print()

      def inorder_util(self,node):
            if node!=None:
                  self.inorder_util(node.left)
                  print(node.data,end=" ")
                  self.inorder_util(node.right)

      def preorder(self):
            self.preorder_util(self.root)
            print()
      def preorder_util(self,node):
            if node!=None:
                  print(node.data,end=" ")
                  self.preorder_util(node.left)
                  self.preorder_util(node.right)

      def postorder(self):
            self.postorder_util(self.root)
            print()

      def postorder_util(self,node):
            if node!=None:
                  self.postorder_util(node.left)
                  self.postorder_util(node.right)
                  print(node.data,end=" ")


L = [1,2,3,4,5,6,7]
#    0 1 2 3 4 5 6
obj = BTree()
obj.createtree(L)
obj.inorder()
obj.preorder()
obj.postorder()
```

```
C:\DSAP>py BT1.py
4 2 5 1 6 3 7
1 2 4 5 3 6 7
4 5 2 6 7 3 1
```

```
case2:
------
L = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
obj = BTree()
obj.createtree(L)
obj.inorder()
obj.preorder()
obj.postorder()
```

```
C:\DSAP>py BT1.py
8 4 9 2 10 5 11 1 12 6 13 3 14 7 15
1 2 4 8 9 5 10 11 3 6 12 13 7 14 15
8 9 4 10 11 5 2 12 13 6 14 15 7 3 1
```

```
BT implementation without using list
-------------------------------------
class BTree:

      class Node:

            def __init__(self,data,left=None,right=None):
                  self.data = data
                  self.left = left
                  self.right = right

      def __init__(self):
            self.root = None

      def inorder(self):
            self.inorder_util(self.root)
            print()
```

```
    def inorder_util(self,node):
        if node!=None:
            self.inorder_util(node.left)
            print(node.data,end=" ")
            self.inorder_util(node.right)

    def preorder(self):
        self.preorder_util(self.root)
        print()

    def preorder_util(self,node):
        if node!=None:
            print(node.data,end=" ")
            self.preorder_util(node.left)
            self.preorder_util(node.right)

    def postorder(self):
        self.postorder_util(self.root)
        print()

    def postorder_util(self,node):
        if node!=None:
            self.postorder_util(node.left)
            self.postorder_util(node.right)
            print(node.data,end=" ")


obj = BTree()
obj.root = obj.Node(10)
obj.root.left = obj.Node(5)
obj.root.right = obj.Node(12)
obj.root.right.left = obj.Node(11)
obj.root.right.right = obj.Node(13)
obj.inorder()
obj.preorder()
obj.postorder()
```

```
C:\DSAP>py BT1.py
5 10 11 12 13
10 5 12 11 13
5 11 13 12 10
```

level order traversing
count nodes
sum of nodes
find max element
find min element
search
num of leaf nodes
print all paths

```
import sys

class BTree:

    class Node:

        def __init__(self,data,left=None,right=None):
            self.data = data
            self.left = left
            self.right = right
```

```
def __init__(self):
      self.root = None

def createtree(self,L):
      self.root = self.createtree_util(L,0)

def createtree_util(self,L,start):
      size = len(L)
      curr = self.Node(L[start])
      l = 2*start + 1
      r = 2*start + 2
      if l<size:
            curr.left = self.createtree_util(L,l)
      if r<size:
            curr.right = self.createtree_util(L,r)
      return curr
def inorder(self):
      self.inorder_util(self.root)
      print()

def inorder_util(self,node):
      if node!=None:
            self.inorder_util(node.left)
            print(node.data,end=" ")
            self.inorder_util(node.right)

def preorder(self):
      self.preorder_util(self.root)
      print()

def preorder_util(self,node):
      if node!=None:
            print(node.data,end=" ")
            self.preorder_util(node.left)
            self.preorder_util(node.right)

def postorder(self):
      self.postorder_util(self.root)
      print()

def postorder_util(self,node):
      if node!=None:
            self.postorder_util(node.left)
            self.postorder_util(node.right)
            print(node.data,end=" ")

def level_order(self):
      if self.root == None:
            return None
      q = []
      q.append(self.root)
      q.append(None)
      while len(q)!=0:
            temp = q.pop(0)
            if temp==None:
                  print()
                  if len(q)==0:
                        break
                  else:
                        q.append(None)
            else:
                  print(temp.data,end=" ")
                  if temp.left!=None:
                        q.append(temp.left)
```

```
                if temp.right!=None:
                    q.append(temp.right)

    def countnodes(self,node):
        if node==None:
            return 0
        lc = self.countnodes(node.left)
        rc = self.countnodes(node.right)
        return lc+rc+1

    def sumofnodes(self,node):
        if node==None:
            return 0
        ls = self.sumofnodes(node.left)
        rs = self.sumofnodes(node.right)
        return ls + rs + node.data

    def search(self,node,key):
        if node==None:
            return False
        if node.data == key:
            return True
        if self.search(node.left,key):
            return True
        if self.search(node.right,key):
            return True
        return False

    def maxElement(self,node):
        if node==None:
            return -1
        m = node.data
        l = self.maxElement(node.left)
        r = self.maxElement(node.right)
        return max(m,l,r)

    def minElement(self,node):
        if node==None:
            return sys.maxsize
        m = node.data
        l = self.minElement(node.left)
        r = self.minElement(node.right)
        return min(m,l,r)

    def numofleaf(self,node):
        if node==None:
            return 0
        if node.left==None and node.right==None:
            return 1
        return self.numofleaf(node.left)+self.numofleaf(node.right)

    def printallpaths(self):
        l = []
        self.printallpaths_util(self.root,l)

    def printallpaths_util(self,node,l):
        if node==None:
            return
        l.append(node.data)
        if node.left==None and node.right==None:
            print(l)
            l.pop()
            return
        self.printallpaths_util(node.left,l)
```

```
            self.printallpaths_util(node.right,l)
            l.pop()


obj = BTree()
'''obj.root = obj.Node(3)
obj.root.left = obj.Node(7)
obj.root.left.left = obj.Node(1)
obj.root.right = obj.Node(5)
obj.root.right.left = obj.Node(2)
obj.root.right.right = obj.Node(4)'''
L = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
obj.createtree(L)
obj.level_order()
obj.printallpaths()


equality
---------
import sys

class BTree:

      class Node:

            def __init__(self,data,left=None,right=None):
                  self.data = data
                  self.left = left
                  self.right = right

      def __init__(self):
            self.root = None

      def createtree(self,L):
            self.root = self.createtree_util(L,0)

      def createtree_util(self,L,start):
            size = len(L)
            curr = self.Node(L[start])
            l = 2*start + 1
            r = 2*start + 2
            if l<size:
                  curr.left = self.createtree_util(L,l)
            if r<size:
                  curr.right = self.createtree_util(L,r)
            return curr
      def inorder(self):
            self.inorder_util(self.root)
            print()

      def inorder_util(self,node):
            if node!=None:
                  self.inorder_util(node.left)
                  print(node.data,end=" ")
                  self.inorder_util(node.right)

      def preorder(self):
            self.preorder_util(self.root)
            print()

      def preorder_util(self,node):
            if node!=None:
                  print(node.data,end=" ")
                  self.preorder_util(node.left)
            self.preorder_util(node.right)

    def postorder(self):
        self.postorder_util(self.root)
        print()

    def postorder_util(self,node):
        if node!=None:
            self.postorder_util(node.left)
            self.postorder_util(node.right)
            print(node.data,end=" ")

    def level_order(self):
        if self.root == None:
            return None
        q = []
        q.append(self.root)
        q.append(None)
        while len(q)!=0:
            temp = q.pop(0)
            if temp==None:
                print()
                if len(q)==0:
                    break
                else:
                    q.append(None)
            else:
                print(temp.data,end=" ")
                if temp.left!=None:
                    q.append(temp.left)
                if temp.right!=None:
                    q.append(temp.right)

    def countnodes(self,node):
        if node==None:
            return 0
        lc = self.countnodes(node.left)
        rc = self.countnodes(node.right)
        return lc+rc+1

    def sumofnodes(self,node):
        if node==None:
            return 0
        ls = self.sumofnodes(node.left)
        rs = self.sumofnodes(node.right)
        return ls + rs + node.data

    def search(self,node,key):
        if node==None:
            return False
        if node.data == key:
            return True
        if self.search(node.left,key):
            return True
        if self.search(node.right,key):
            return True
        return False

    def maxElement(self,node):
        if node==None:
            return -1
        m = node.data
        l = self.maxElement(node.left)
        r = self.maxElement(node.right)
        return max(m,l,r)

    def minElement(self,node):
        if node==None:
            return sys.maxsize
        m = node.data
        l = self.minElement(node.left)
        r = self.minElement(node.right)
        return min(m,l,r)

    def numofleaf(self,node):
        if node==None:
            return 0
        if node.left==None and node.right==None:
            return 1
        return self.numofleaf(node.left)+self.numofleaf(node.right)

    def printallpaths(self):
        l = []
        self.printallpaths_util(self.root,l)

    def printallpaths_util(self,node,l):
        if node==None:
            return
        l.append(node.data)
        if node.left==None and node.right==None:
            print(l)
            l.pop()
            return
        self.printallpaths_util(node.left,l)
        self.printallpaths_util(node.right,l)
        l.pop()

    def equal(self,node1,node2):
        if node1==None and node2==None:
            return True
        if node1==None or node2==None:
            return False
        return self.equal(node1.left,node2.left) and
self.equal(node1.right,node2.right) and node1.data==node2.data

    def copytree(self,node):
        if node!=None:
            temp = self.Node(node.data)
            temp.left = self.copytree(node.left)
            temp.right = self.copytree(node.right)
            return temp
        return None




obj1 = BTree()
obj1.root = obj1.Node(2)
obj1.root.left = obj1.Node(1)
obj1.root.right = obj1.Node(3)

obj2 = BTree()
obj2.root = obj2.Node(2)
obj2.root.left = obj2.Node(1)
obj2.root.right = obj2.Node(4)

obj3 = BTree()
obj3.root = obj3.Node(2)
```

```
obj3.root.left = obj3.Node(1)
obj3.root.right = obj3.Node(3)

print(obj1.equal(obj1.root,obj2.root)) #False
print(obj1.equal(obj1.root,obj3.root)) #True


copy tree
---------
import sys

class BTree:

      class Node:

            def __init__(self,data,left=None,right=None):
                  self.data = data
                  self.left = left
                  self.right = right

      def __init__(self):
            self.root = None

      def createtree(self,L):
            self.root = self.createtree_util(L,0)

      def createtree_util(self,L,start):
            size = len(L)
            curr = self.Node(L[start])
            l = 2*start + 1
            r = 2*start + 2
            if l<size:
                  curr.left = self.createtree_util(L,l)
            if r<size:
                  curr.right = self.createtree_util(L,r)
            return curr
      def inorder(self):
            self.inorder_util(self.root)
            print()

      def inorder_util(self,node):
            if node!=None:
                  self.inorder_util(node.left)
                  print(node.data,end=" ")
                  self.inorder_util(node.right)

      def preorder(self):
            self.preorder_util(self.root)
            print()

      def preorder_util(self,node):
            if node!=None:
                  print(node.data,end=" ")
                  self.preorder_util(node.left)
                  self.preorder_util(node.right)

      def postorder(self):
            self.postorder_util(self.root)
            print()

      def postorder_util(self,node):
            if node!=None:
                  self.postorder_util(node.left)
                  self.postorder_util(node.right)
                  print(node.data,end=" ")

      def level_order(self):
            if self.root == None:
                  return None
            q = []
            q.append(self.root)
            q.append(None)
            while len(q)!=0:
                  temp = q.pop(0)
                  if temp==None:
                        print()
                        if len(q)==0:
                              break
                        else:
                              q.append(None)
                  else:
                        print(temp.data,end=" ")
                        if temp.left!=None:
                              q.append(temp.left)
                        if temp.right!=None:
                              q.append(temp.right)

      def countnodes(self,node):
            if node==None:
                  return 0
            lc = self.countnodes(node.left)
            rc = self.countnodes(node.right)
            return lc+rc+1

      def sumofnodes(self,node):
            if node==None:
                  return 0
            ls = self.sumofnodes(node.left)
            rs = self.sumofnodes(node.right)
            return ls + rs + node.data

      def search(self,node,key):
            if node==None:
                  return False
            if node.data == key:
                  return True
            if self.search(node.left,key):
                  return True
            if self.search(node.right,key):
                  return True
            return False

      def maxElement(self,node):
            if node==None:
                  return -1
            m = node.data
            l = self.maxElement(node.left)
            r = self.maxElement(node.right)
            return max(m,l,r)

      def minElement(self,node):
            if node==None:
                  return sys.maxsize
            m = node.data
            l = self.minElement(node.left)
            r = self.minElement(node.right)
            return min(m,l,r)
      def numofleaf(self,node):
            if node==None:
                  return 0
            if node.left==None and node.right==None:
                  return 1
            return self.numofleaf(node.left)+self.numofleaf(node.right)

      def printallpaths(self):
            l = []
            self.printallpaths_util(self.root,l)

      def printallpaths_util(self,node,l):
            if node==None:
                  return
            l.append(node.data)
            if node.left==None and node.right==None:
                  print(l)
                  l.pop()
                  return
            self.printallpaths_util(node.left,l)
            self.printallpaths_util(node.right,l)
            l.pop()

      def equal(self,node1,node2):
            if node1==None and node2==None:
                  return True
            if node1==None or node2==None:
                  return False
            return self.equal(node1.left,node2.left) and
self.equal(node1.right,node2.right) and node1.data==node2.data

      def copytree(self,node):
            if node!=None:
                  temp = self.Node(node.data)
                  temp.left = self.copytree(node.left)
                  temp.right = self.copytree(node.right)
                  return temp
            return None




obj1 = BTree()
obj1.root = obj1.Node(1)
obj1.root.left = obj1.Node(2)
obj1.root.right = obj1.Node(3)
obj1.level_order()

obj2 = BTree()
obj2.root = obj1.copytree(obj1.root)
obj2.level_order()
```

```
C:\DSAP>py BT.py
1
2 3
1
2 3
```

Binary Search Trees:
~~~~~~~~~~~~~~~~~~~~
Binary Search Tree (BST) is a kind of tree with the following properties.
~~~~~~~~~~~~~~~~~~~~

1. elements which are existed in the left sub-tree of BST is < root node value.
2. elements which are existed in the right sub-tree of BST is > root node value
3. no duplicates are allowed

Ex:

```
class BSTree:

      class Node:

            def __init__(self,data,left=None,right=None):
                  self.data = data
                  self.left = left
                  self.right = right

      def __init__(self):
            self.root = None

      def insert(self,data):
            self.root = self.insert_util(self.root,data)

      def insert_util(self,node,data):
            if node==None:
                  node = self.Node(data)
            else:
                  if node.data > data:
                        node.left = self.insert_util(node.left,data)
                  else:
                        node.right = self.insert_util(node.right,data)
            return node

      def search(self,key):
            curr = self.root
            while curr!=None:
                  if curr.data == key:
                        return True
                  elif curr.data > key:
                        curr = curr.left
                  else:
                        curr = curr.right
            return False

      def findmin(self):
            node = self.root
            if node==None:
                  return "tree is not there"
            while node.left!=None:
                  node = node.left
            return node.data

      def findmax(self):
            node = self.root
            if node==None:
                  return "tree is not there"
            while node.right!=None:
                  node = node.right
            return node.data

      def remove_all_leafnodes(self):
            self.root = self.remove_all_leafnodes_util(self.root)

      def remove_all_leafnodes_util(self,node):
            if node==None:
                  return None
            if node.left==None and node.right==None:
                  return None
            node.left = self.remove_all_leafnodes_util(node.left)
            node.right = self.remove_all_leafnodes_util(node.right)
            return node

      def print_root_to_leaf(self):
            self.print_root_to_leaf_util(self.root,[])

      def print_root_to_leaf_util(self,node,L):
            if node==None:
                  return
            L.append(node.data)
            if node.left==None and node.right==None:
                  self.print_path(L)
            self.print_root_to_leaf_util(node.left,L)
            self.print_root_to_leaf_util(node.right,L)
            L.pop(-1)

      def print_path(self,L):
            for i in L:
                  print(i,end=' ')
            print()




      def level_order(self):
            if self.root == None:
                  return None
            q = []
            q.append(self.root)
            q.append(None)
            while len(q)!=0:
                  temp = q.pop(0)
                  if temp==None:
                        print()
                        if len(q)==0:
                              break
                        else:
                              q.append(None)
                  else:
                        print(temp.data,end=" ")
                        if temp.left!=None:
                              q.append(temp.left)
                        if temp.right!=None:
                              q.append(temp.right)


T = BSTree()
T.insert(6)
T.insert(4)
T.insert(8)
T.insert(1)
T.insert(5)
T.insert(9)
T.insert(10)
```

```
T.level_order()
print()
T.print_root_to_leaf()
```

creation
insert
max/min
traversal
root to leaf
removing leaf

delete operation on BST
----
case 1: deletion of a node which is having no children, directly we can del
case 2: deletion of a node which is having one child.

delete that node and make its left/right child as parent

case 3: deletion of a node which is having two chilren

```
class BSTree:

    class Node:

        def __init__(self,data,left=None,right=None):
            self.data = data
            self.left = left
            self.right = right

    def __init__(self):
        self.root = None

    def insert(self,data):
        self.root = self.insert_util(self.root,data)

    def insert_util(self,node,data):
        if node==None:
            node = self.Node(data)
        else:
            if node.data > data:
                node.left = self.insert_util(node.left,data)
            else:
                node.right = self.insert_util(node.right,data)
        return node

    def search(self,key):
        curr = self.root
        while curr!=None:
            if curr.data == key:
                return True
            elif curr.data > key:
                curr = curr.left
            else:
                curr = curr.right
        return False

    def findmin(self):
        node = self.root
        if node==None:
            return "tree is not there"
        while node.left!=None:
            node = node.left
        return node.data

    def findmax(self):
        node = self.root
        if node==None:
            return "tree is not there"
        while node.right!=None:
            node = node.right
        return node.data

    def findmax_util(self,node):
        if node==None:
            return None
        while node.right!=None:
            node = node.right
        return node


    def remove_all_leafnodes(self):
        self.root = self.remove_all_leafnodes_util(self.root)

    def remove_all_leafnodes_util(self,node):
        if node==None:
            return None
        if node.left==None and node.right==None:
            return None
        node.left = self.remove_all_leafnodes_util(node.left)
        node.right = self.remove_all_leafnodes_util(node.right)
        return node

    def print_root_to_leaf(self):
        self.print_root_to_leaf_util(self.root,[])

    def print_root_to_leaf_util(self,node,L):
        if node==None:
            return
        L.append(node.data)
        if node.left==None and node.right==None:
            self.print_path(L)
        self.print_root_to_leaf_util(node.left,L)
        self.print_root_to_leaf_util(node.right,L)
        L.pop(-1)

    def print_path(self,L):
        for i in L:
            print(i,end=' ')
        print()

    def level_order(self):
        if self.root == None:
            return None
        q = []
        q.append(self.root)
        q.append(None)
        while len(q)!=0:
            temp = q.pop(0)
            if temp==None:
                print()
                if len(q)==0:
                    break
                else:
                    q.append(None)
```

```
                  else:
                        print(temp.data,end=" ")
                        if temp.left!=None:
                              q.append(temp.left)
                        if temp.right!=None:
                              q.append(temp.right)
      def delete(self,data):
            self.root = self.delete_util(self.root,data)

      def delete_util(self,node,data):
            temp = None
            if node!=None:
                  if node.data == data:
                        #case1
                        if node.left==None and node.right==None:
                              return None
                        else:
                              #case2
                              if node.left==None:
                                    temp = node.right
                                    return temp
                              if node.right==None:
                                    teemp = node.left
                                    return temp
                              #case3
                              maxnode = self.findmax_util(node.left)
                              node.data = maxnode.data
                              node.left =
self.delete_util(node.left,maxnode.data)
                  else:
                        if node.data > data:
                              node.left = self.delete_util(node.left,data)
                        else:
                              node.right = self.delete_util(node.right,data)
            return node

T = BSTree()
T.insert(6)
T.insert(4)
T.insert(8)
T.insert(1)
T.insert(5)
T.insert(7)
T.insert(10)
T.insert(9)
T.insert(12)
T.insert(2)
T.insert(13)
T.level_order()
T.delete(6)
T.level_order()
```

We have to balance the height of the tree to over come the problems with BST.

1. AVL trees
2. RED BLACK tress
3. B Trees
4. B+ Trees

AVL Tree:

==> It is a BST.
==> It is called as self-balancing binary search tree.
==> Adelson Velsky Landis (AVL)

==> All the operations in AVL tree will take O(logN)
==> Balance Factor for every node in the BST
==> Balance Factor BF = height of left sub-tree - height of right sub-tree
==> The balance factor for AVL tree must be -1 or 0 or +1
==> If we are getting other than these values, then that is not a AVL tree
==> Rotations on the tree to make it as AVL tree

LL Rotation
RR Rotation
LR Rotation
RL Rotation

```
class AVL:

      class Node:

            def __init__(self,data,left=None,right=None):
                  self.data = data
                  self.left = left
                  self.right = right
                  self.height = 1

      def __init__(self):
            self.root = None

      def insert(self,data):
            self.root = self.insert_util(self.root,data)

      def insert_util(self,node,data):
            if node==None:
                  node = self.Node(data)
            else:
                  if node.data > data:
                        node.left = self.insert_util(node.left,data)
                  else:
                        node.right = self.insert_util(node.right,data)

            node.height = 1 + 
max(self.getheight(node.left),self.getheight(node.right))
            balance = self.getbalance(node)

            #LL case
            if balance > 1 and data < node.left.data:
                  return self.rightrotate(node)

            #RR case
            if balance < -1 and data > node.right.data:
                  return self.leftrotate(node)

            #LR case
            if balance > 1 and data > node.left.data:
                  node.left = self.leftrotate(node.left)
                  return self.rightrotate(node)

            #RL case
            if balance < -1 and data < node.right.data:
                  node.right = self.rightrotate(node.right)
                  return self.leftrotate(node)
            return node

      def findmax_util(self,node):
            if node==None:
                  return None
```

```
        while node.right!=None:
            node = node.right
        return node

    def delete(self,data):
        self.root = self.delete_util(self.root,data)

    def delete_util(self,node,data):
        temp = None
        if node!=None:
            if node.data == data:
                #case1
                if node.left==None and node.right==None:
                    return None
                else:
                    #case2
                    if node.left==None:
                        temp = node.right
                        return temp
                    if node.right==None:
                        teemp = node.left
                        return temp
                    #case3
                    maxnode = self.findmax_util(node.left)
                    node.data = maxnode.data
                    node.left =
self.delete_util(node.left,maxnode.data)
            else:
                if node.data > data:
                    node.left = self.delete_util(node.left,data)
                else:
                    node.right = self.delete_util(node.right,data)
        node.height = 1 +
max(self.getheight(node.left),self.getheight(node.right))
        balance = self.getbalance(node)

        #LL case
        if balance > 1 and data < node.left.data:
            return self.rightrotate(node)

        #RR case
        if balance < -1 and data > node.right.data:
            return self.leftrotate(node)

        #LR case
        if balance > 1 and data > node.left.data:
            node.left = self.leftrotate(node.left)
            return self.rightrotate(node)

        #RL case
        if balance < -1 and data < node.right.data:
            node.right = self.rightrotate(node.right)
            return self.leftrotate(node)

        return node

    #left rotation code
    def leftrotate(self,z):
        y = z.right
        t2 = y.left
        y.left = z
        z.right = t2
        z.height = 1 + max(self.getheight(z.left),self.getheight(z.right))
        y.height = 1 + max(self.getheight(y.left),self.getheight(y.right))
        return y

    #right rotation code
    def rightrotate(self,z):
        y = z.left
        t3 = y.right
        y.right = z
        z.left = t3
        z.height = 1 + max(self.getheight(z.left),self.getheight(z.right))
        y.height = 1 + max(self.getheight(y.left),self.getheight(y.right))
        return y

    def getheight(self,node):
        if node==None:
            return 0
        return node.height

    def getbalance(self,node):
        if node==None:
            return 0
        return self.getheight(node.left) - self.getheight(node.right)

    def level_order(self):
        if self.root == None:
            return None
        q = []
        q.append(self.root)
        q.append(None)
        while len(q)!=0:
            temp = q.pop(0)
            if temp==None:
                print()
                if len(q)==0:
                    break
                else:
                    q.append(None)
            else:
                print(temp.data,end=" ")
                if temp.left!=None:
                    q.append(temp.left)
                if temp.right!=None:
                    q.append(temp.right)

T = AVL()
T.insert(40)
T.insert(20)
T.insert(10)
T.insert(25)
T.insert(30)
T.insert(22)
T.insert(50)
T.level_order()
T.delete(25)
T.level_order()
```