DAY 5 - NON LINEAR DATA STRUCTURES

BINARY TREE:A node can have maximum of two children

class Binarytreenode:

def \_\_init\_\_(self,data):

self.data=data

self.leftchild=None

self.rightnode=None

node1=Binarytreenode(50)

node2=Binarytreenode(20)

node3=Binarytreenode(45)

node4=Binarytreenode(11)

node5=Binarytreenode(15)

node6=Binarytreenode(30)

node7=Binarytreenode(78)

node1.leftchild=node2

node1.rightchild=node3

node2.leftchild=node4

node2.rightchild=node5

node3.leftchild=node6

node3.rightchild=node7

print("root node is:")

print(node1.data)

print("leftchild of root node is:")

print(node1.leftchild.data)

print("rightchild of root node is:")

print(node1.rightchild.data)

print("node is:")

print(node2.data)

print("leftchild of node2 is:")

print(node2.leftchild.data)

print("rightchild of node2 is:")

print(node2.rightchild.data)

print("node is:")

print(node3.data)

BINARY TREE TRAVERSAL:

\* INORDER : left-root-right >LRR >LDR

\*PREORDER: root-left-right > RLR >DLR

\*POST ORDER : left -right- root >LRR>LRD

class node:

def \_\_init\_\_(self,key):

self.left=None

self.right=None

self.val=key

def printinorder(root):

if root:

printinorder(root.left)

print(root.val,end='')

printinorder(root.right)

def printpostorder(root):

if root:

printpostorder(root.left)

printpostorder(root.right)

print(root.val,end='')

def printpreorder(root):

if root:

print(root.val,end='')

printpreorder(root.left)

printpreorder(root.right)

root=node(1)

root.left=node(2)

root.right=node(3)

root.left.left=node(4)

root.right.right=node(5)

print("IN ORDER")

printinorder(root)

print()

print("POST ORDER")

printpostorder(root)

print()

print("PRE ORDER")

printpreorder(root)

print()

TYPES OF BINARY TREE

$ FULL BINARY TREE: All the nodes have 0 or 2 children

$ PATHELOGICAL TREE:((degenerate tree))

$ SCREW TREE : single side

Basedc on the level we classified as >>>>>

\*complete binary tree :1. every level should be full r complete

2. in last level if it is incomplete nodes should present at extreme left side

\*\* perefect binary tree:1.all internal nodes having two childrens and leaf nodes should be at the same level

\*\*Balanced binary Tree: for all the nodes height of left subtree minus height of right subtree can be 0 r 1(height of leftsubtree -height of right subtree =0 r 1)

BINARY EARCH TREE :: all the left side element should be lesser than its parent and all the right side element should be greeater than its parent

class node:

def \_\_init\_\_(self,key):

self.left=None

self.right=None

self.val=key

#a new node with the given key

def insert(root,key):

if root is None :

return node(key)

else:

if root .val== key:

return root

elif root.val < key:

root.right=insert(root.right,key)

else:

root.left = insert(root.left,key)

return root

def printinorder(root):

if root:

printinorder(root.left)

print(root.val,end=' ')

printinorder(root.right)

r= node(50)

r=insert(r,30)

r=insert(r,60)

r=insert(r,10)

r=insert(r,25)

r=insert(r,60)

r=insert(r,75)

printinorder(r)

DELETION OF BINARY SEARCH TREE

# Python program to demonstrate delete operation

# in binary search tree

# A Binary Tree Node

class Node:

# Constructor to create a new node

def \_\_init\_\_(self, key):

self.key = key

self.left = None

self.right = None

# A utility function to do inorder traversal of BST

def inorder(root):

if root is not None:

inorder(root.left)

print(root.key, end=" ")

inorder(root.right)

# A utility function to insert a

# new node with given key in BST

def insert(node, key):

# If the tree is empty, return a new node

if node is None:

return Node(key)

# Otherwise recur down the tree

if key < node.key:

node.left = insert(node.left, key)

else:

node.right = insert(node.right, key)

# return the (unchanged) node pointer

return node

# Given a non-empty binary

# search tree, return the node

# with minimum key value

# found in that tree. Note that the

# entire tree does not need to be searched

def minValueNode(node):

current = node

# loop down to find the leftmost leaf

while(current.left is not None):

current = current.left

return current

# Given a binary search tree and a key, this function

# delete the key and returns the new root

def deleteNode(root, key):

# Base Case

if root is None:

return root

# If the key to be deleted

# is smaller than the root's

# key then it lies in left subtree

if key < root.key:

root.left = deleteNode(root.left, key)

# If the kye to be delete

# is greater than the root's key

# then it lies in right subtree

elif(key > root.key):

root.right = deleteNode(root.right, key)

# If key is same as root's key, then this is the node

# to be deleted

else:

# Node with only one child or no child

if root.left is None:

temp = root.right

root = None

return temp

elif root.right is None:

temp = root.left

root = None

return temp

# Node with two children:

# Get the inorder successor

# (smallest in the right subtree)

temp = minValueNode(root.right)

# Copy the inorder successor's

# content to this node

root.key = temp.key

# Delete the inorder successor

root.right = deleteNode(root.right, temp.key)

return root

root = None

root = insert(root, 50)

root = insert(root, 30)

root = insert(root, 20)

root = insert(root, 40)

root = insert(root, 70)

root = insert(root, 60)

root = insert(root, 80)

print("Inorder traversal of the given tree")

inorder(root)

print("\nDelete 60")

root = deleteNode(root, 60)

print("Inorder traversal of the modified tree")

inorder(root)

print("\nDelete 70")

root = deleteNode(root, 70)

print("Inorder traversal of the modified tree")

inorder(root)

print("\nDelete 50")

root = deleteNode(root, 50)

print("Inorder traversal of the modified tree")

inorder(root)

# This code is contributed by Nikhil Kumar Singh(nickzuck\_007)