***BUDDI.AI***

**ASSIGNMENT 1**

**Reg.No: 210420243060 J. VINU BALAN - AI & DS**

**1) Postfix to numeric expression**

class evalpostfix:

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

self.stack =[]

self.top =-1

def pop(self):

if self.top ==-1:

return

else:

self.top-= 1

return self.stack.pop()

def push(self, i):

self.top+= 1

self.stack.append(i)

#function to calculate postfix expressions

def evaluate(self, postfix):

for i in postfix:

try:

self.push(int(i))

except:

val1 = self.pop()

val2 = self.pop()

s ={'+':val2 + val1, '-':val2-val1, '\*':val2 \* val1, '^':val2\*\*val1,'/':val2/val1}

self.push(s.get(i))

return float(self.pop())

str1 ='10 25 + 2 / 3 \* 6 + 3 %'

#numeric form = (((10 + 25) /2)\*3+6)%3 = ((35/2)\*3+6)%3 = (17.5\*3+6)%3 = (52.5+6)%3 = 58.5%3 = 1

str1 = str1.split(' ')

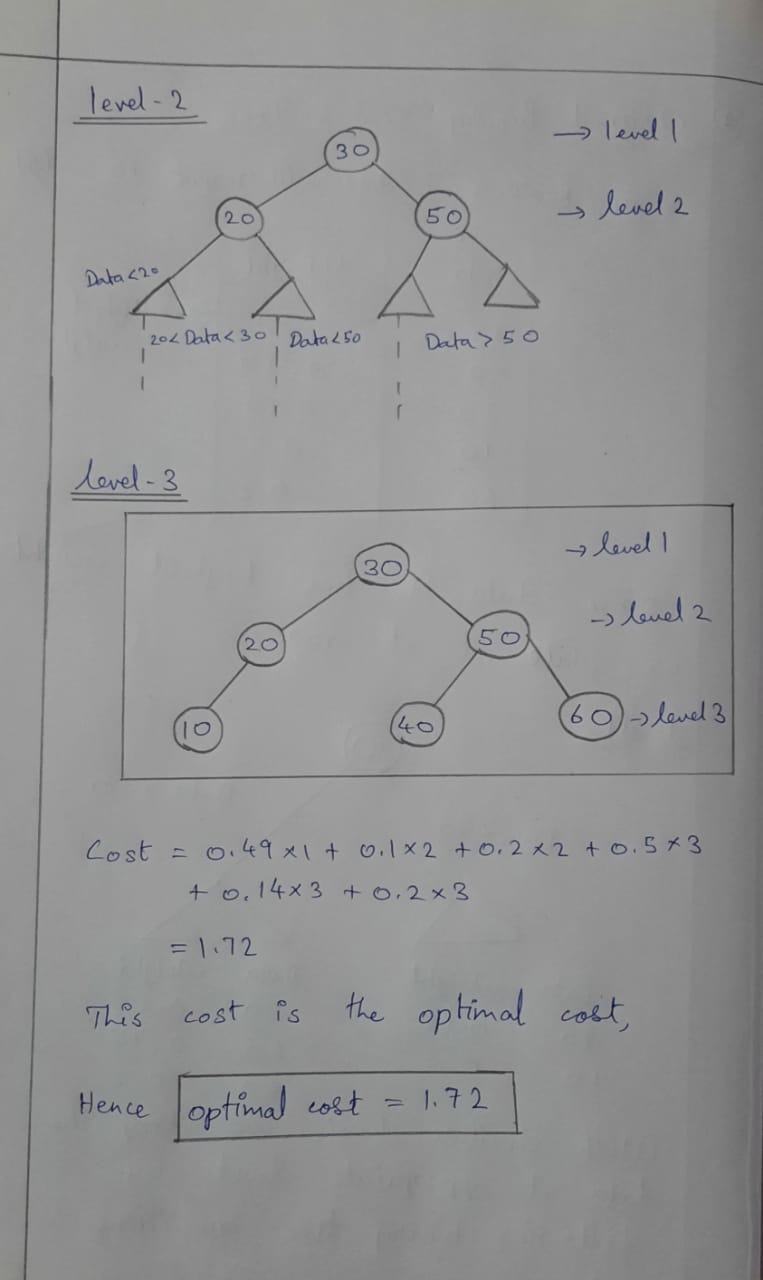
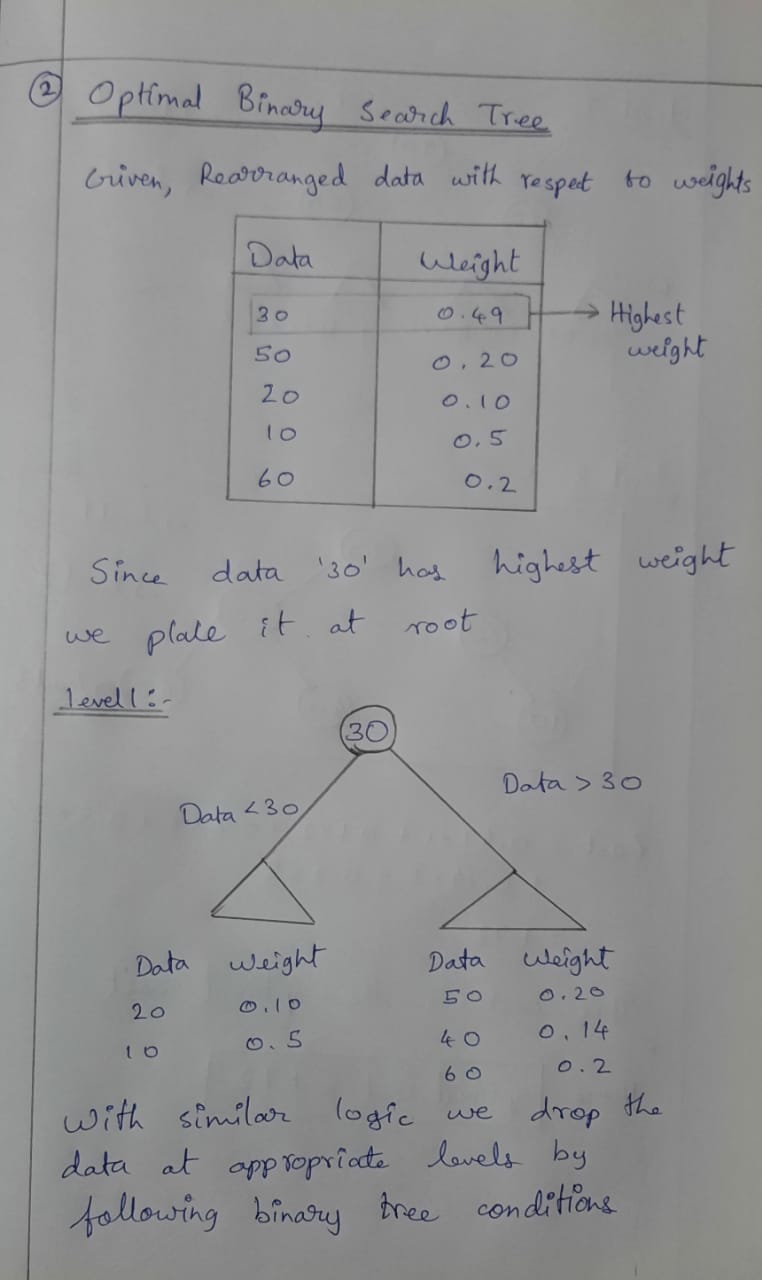
obj = evalpostfix()

print(obj.evaluate(str1))

OUTPUT:

1

**2) Consider the data in Table 1. Construct an optimal BST that will minimize the average cost of performing searches.**



**3)RESISTANCE CALCULATIONS PROGRAM**

def calResistance(ring1,ring2,ring3):

color\_digit = {'black': '0',

'brown': '1',

'red': '2',

'orange': '3',

'yellow': '4',

'green' : '5',

'blue' : '6',

'violet' : '7',

'grey' : '8',

'white': '9'}

multiplier = {'black': '1',

'brown': '10',

'red': '100',

'orange': '1000',

'yellow': '10000',

'green' : '100000',

'blue' : '1000000',

'violet' : '10000000',

'grey' : '100000000',

}

if ring1 in color\_digit and ring2 in color\_digit and ring3 in multiplier:

x = color\_digit[ring1]

x += color\_digit[ring2]

x = int(x)

x2 = int(multiplier[ring3])

r = x\*x2

if r>=1000000:

return str(r//1000000)+"M Ohms"

elif r>=1000:

return str(r//1000)+"K Ohms"

else:

return str(r)+" Ohms"

else:

print("Invalid Colors")

return

print(calResistance("brown","black","yellow"))

**4)exp(x):**

def exp(x):

n=10

s = 0

for i in range(n,0,-1):

s+ = 1 + x\*s/i

return s

val = exp(5)

print("Sum of the series:",val)

OUTPUT:

Sum of the series: 143.68945656966488

TIME COMPLEXITY:

O(n)

**5) MyBigInteger (addition):**

class MyBigInteger:

def \_\_init\_\_(self,a=0):

a = int(bin(int(a)).replace("0b",""))

self.val = ['0' for x in range(1024)] #binary value

i=-1

#storing the bits in the 1024 sized array

while a>0:

self.val[i] = str(a%10)

a = a//10

i-=1

self.dval = int("".join(self.val),2) #decimal value

def add(a,b):

def add(a,b):

x = "".join(a.val)

y = "".join(b.val)

max\_len = len(x) if len(x)>len(y) else len(y)

#binary addition of two binary numbers

result = ''

carry = 0

for i in range(max\_len - 1, -1, -1):

r = carry

r += 1 if x[i] == '1' else 0

r += 1 if y[i] == '1' else 0

result = ('1' if r % 2 == 1 else '0') + result

carry = 0 if r < 2 else 1

if carry != 0:

result = '1' + result

return list(result),int(result,2)

if \_\_name\_\_=='\_\_main\_\_':

a = MyBigInteger("18446744073709551615023454774323254365489") #134 bits

b = MyBigInteger("21343534564699384802523754676785685676754375") #144 bits

s = MyBigInteger()

s.val,s.dval = add(a,b)

print(s.dval)

**OUTPUT:**

21361981308773094354138778131560008931119864

**6) MyBigInteger (multiply):**

class MyBigInteger:

def \_\_init\_\_(self,a=0):

a = int(bin(int(a)).replace("0b",""))

self.val = ['0' for x in range(1024)]

i=-1

#storing the bits in the 1024 sized array

while a>0:

self.val[i] = str(a%10)

a = a//10

i-=1

self.dval = int("".join(self.val),2)

def multiply(a,b):

x = a.val

y = b.val

if len(x)>len(y):

a1 = x

b1 = y

else:

a1 = y

b1 = x

fresult = ['0' for x in range(2\*len(a.val))] #new array with size 2048

#long multuplication method

k=0

fres = '0'

r=''

for i in range(len(a1)-1,-1,-1):

r = ''

res = ''

for j in range(len(b1)-1,-1,-1):

r = str(int(a1[i])\*int(b1[j])) + r

res = r + k\*'0'

k+=1

fres = bin(int(fres,2)+int(res,2))[2:]

i=-1

fresval = int(fres)

#storing the bits in the 2048 sized array

while fresval>0:

fresult[i] = fresval%10

fresval = fresval//10

i-=1

return fresult,int(fres,2)

if \_\_name\_\_=='\_\_main\_\_':

a = MyBigInteger("18446744073709551615023454774323254365489") #134 bits

b = MyBigInteger("21343534564699384802523754676785685676754375") #144 bits

m = MyBigInteger()

m.val,m.dval = multiply(a,b)

print(m.dval)

**OUTPUT:**

393718719743383351053392725288315167327173515936836726927443189020613281047529764375

**7) Given a set of strings S, write a function that would construct a balanced binary search tree. The height of the constructed BST should be O(log n), where n = |S|, the cardinality of the input set S:**

class Node:

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

self.left = None

self.right = None

self.val = 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 inorder(root):

if root:

inorder(root.left)

print(root.val)

inorder(root.right)

r = Node("bob")

r = insert(r, "john")

r = insert(r, "apple")

r = insert(r, "tom")

# Print inoder traversal of the BST

inorder(r)

**OUTPUT:**

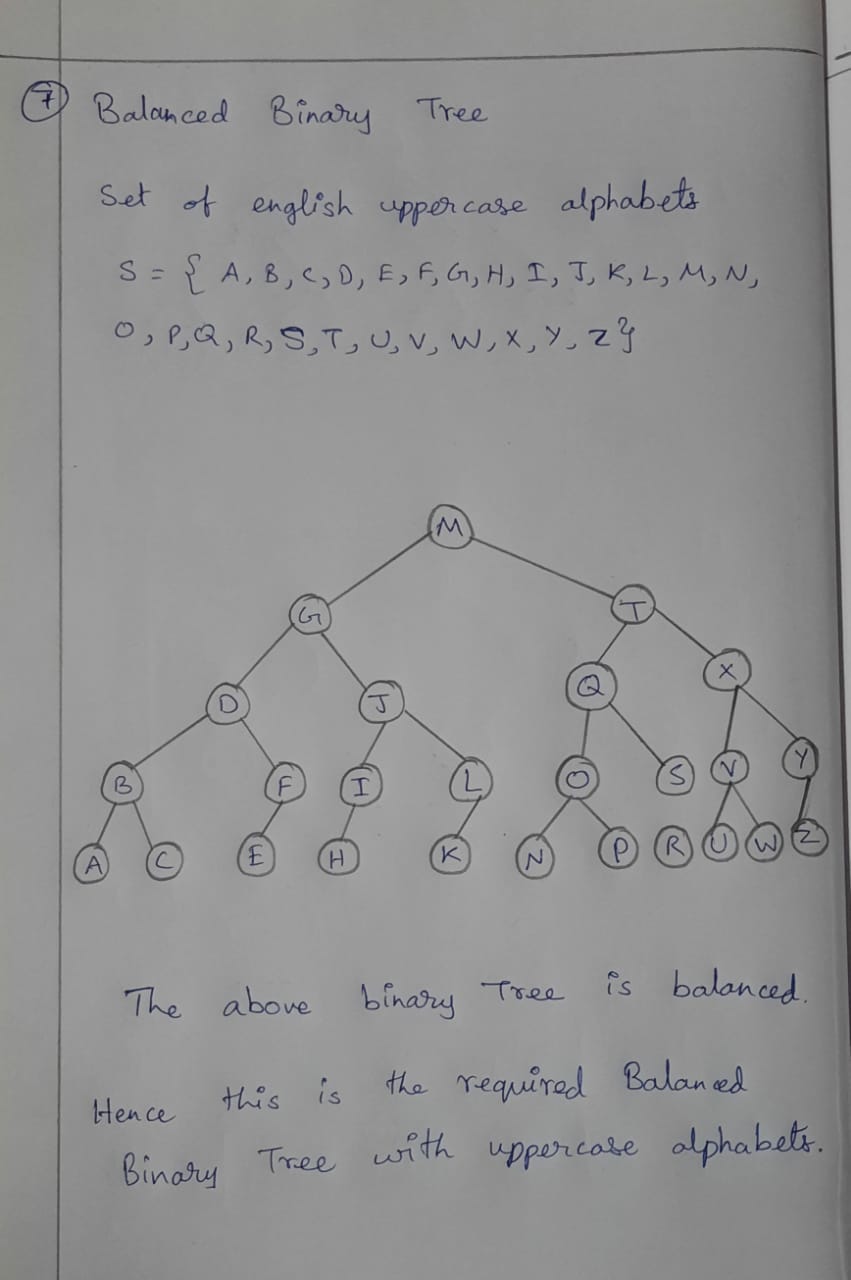
apple

bob

john

tom

**8) Given the English alphabet as a set of symbols, construct a balanced BST by hand:**

****

**9) Given a graph G(V, E), check if it is a clique. Additionally, can you identify sub-graphs of G that are cliques with a minimum of 3 vertices:**

import numpy as np

MAX = 100;

# Stores the vertices

store = [0]\* MAX;

# Graph

graph = np.zeros((MAX, MAX));

# Degree of the vertices

d = [0] \* MAX;

def is\_clique(b) :

for i in range(1, b) :

for j in range(i + 1, b) :

if (graph[store[i]][store[j]] == 0) :

return False;

return True;

# Function to print the clique

def print\_cli(n) :

for i in range(1, n) :

print(store[i], end = " ");

print(",", end=" ");

# Function to find all the cliques of size s>=3

def findCliques(i, l, s) :

for j in range( i + 1, n -(s - l) + 1) :

if (d[j] >= s - 1) :

store[l] = j;

if (is\_clique(l + 1)) :

if (l < s) :

findCliques(j, l + 1, s)

else :

print\_cli(l + 1)

# Driver code

if \_\_name\_\_ == "\_\_main\_\_" :

edges = [ [ 1, 2 ],

[ 2, 3 ],

[ 3, 1 ],

[ 4, 3 ],

[ 4, 5 ],

[ 5, 3 ] ]

k = 3

size = len(edges)

n = 5

for i in range(size) :

graph[edges[i][0]][edges[i][1]] = 1

graph[edges[i][1]][edges[i][0]] = 1

d[edges[i][0]] += 1

d[edges[i][1]] += 1

findCliques(0, 1, k)

**OUTPUT:**

1 2 3 , 3 4 5

**10)** **Given a chess board of 8x8, write a algorithm that would place a maximum number of color-insensitive horses without killing each other:**

b1 = []

b2 = []

n=8

#creating the 8x8 chessboard

for i in range(n):

x =[]

for j in range(n):

x.append(0)

b1.append(x)

b2.append(x.copy())

#placing the knights

for i in range(n):

for j in range(n):

if (i+j)%2==0:

b1[i][j]=1

else:

b2[i][j]=1

#maximum knights placed can also be calculated using the below formula

**N = n\*n/2**  #for all n is a multiple of 4 and n>=4 (It’s an Emprical Proof)

print("Maximum number of knights placed:",N)

print("Solution 1:")

for i in b1:

print(i)

print("Solution 2:")

for i in b2:

print(i)

print("Here 1 represents the knights and 0 represents the empty boxes")

**OUTPUT:**

Maximum number of knights placed: 32

Solution 1:

[1, 0, 1, 0, 1, 0, 1, 0]

[0, 1, 0, 1, 0, 1, 0, 1]

[1, 0, 1, 0, 1, 0, 1, 0]

[0, 1, 0, 1, 0, 1, 0, 1]

[1, 0, 1, 0, 1, 0, 1, 0]

[0, 1, 0, 1, 0, 1, 0, 1]

[1, 0, 1, 0, 1, 0, 1, 0]

[0, 1, 0, 1, 0, 1, 0, 1]

Solution 2:

[0, 1, 0, 1, 0, 1, 0, 1]

[1, 0, 1, 0, 1, 0, 1, 0]

[0, 1, 0, 1, 0, 1, 0, 1]

[1, 0, 1, 0, 1, 0, 1, 0]

[0, 1, 0, 1, 0, 1, 0, 1]

[1, 0, 1, 0, 1, 0, 1, 0]

[0, 1, 0, 1, 0, 1, 0, 1]

[1, 0, 1, 0, 1, 0, 1, 0]

Here 1 represents the knights and 0 represents the empty boxes

**11)Let A n×n be a matrix, In be an identity matrix and k be a scalar. Implement a method to evaluate the expression A × kI. You should be faster than O(n 3 ):**

a = []

n=3

#A nxn

for i in range(n):

x =[]

for j in range(n):

x.append(1)

a.append(x)

#Scalar k

k = 2

#AxKI

for i in range(n):

for j in range(n):

a[i][j]\*=k

for i in a:

print(i)

OUTPUT:

[2, 2, 2]

[2, 2, 2]

[2, 2, 2]

**TMIE COMPLEXITY:**

O(n^2)

**12) Given the weights wi and values vi of n items, you need to put these items in a knapsack of capacity W to maximize the total value in the knapsack:**

# Structure for an item which stores weight and

# corresponding value of Item

class Item:

def \_\_init\_\_(self, value, weight):

self.value = value

self.weight = weight

def fractionalKnapsack(W, arr):

arr.sort(key=lambda x: (x.value/x.weight), reverse=True)

finalvalue = 0.0

for item in arr:

if item.weight <= W:

W -= item.weight

fval += item.value

else:

flval += item.value \* W / item.weight

break

return fval

W = 50

arr = [Item(50, 10), Item(80, 20), Item(60, 30)]

max\_val = fractionalKnapsack(W, arr)

print(max\_val)

**Output:**

170

**13) Implement Fibonacci number def fibonacci(n) with memoization:**

def fibonacci(n):

if n in [0, 1]:

return n

return fibonacci(n - 1) + fibonacci(n - 2)

print([fibonacci(n) for x in range(15)])

**Output:**

[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377]

**TIME COMPLEXITY:**

O(n)

**14) logstar 2(x):**

import math

class Value:

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

self.n = 0

def logstar\_2(x):

if x==1:

v.n+=1

return v.n

else:

v.n+=1

return logstar\_2(math.log2(x))

v = Value()

x = 65536

print(logstar\_2(x))

**Output:**

5

**15) dicrete derivative of sin(x):**

import math

#discrete derivative of sin(X)

def dsin(x):

dx = 0.0000000001 #lowest dx

r = (math.sin(x+dx)-math.sin(x))/dx

return r

#mean squared error between cos(x) and discrete derivative of sin(x)

def mse(a,b):

r=0

for i in range(a,b+1):

r+=(math.cos(i)-dsin(i))\*\*2

return r/(b-a+1)

print("Mean Squared Error:")

print(mse(0,360)) # x E (0,2pi)

**OUTPUT:**

5.023694735806502e-09