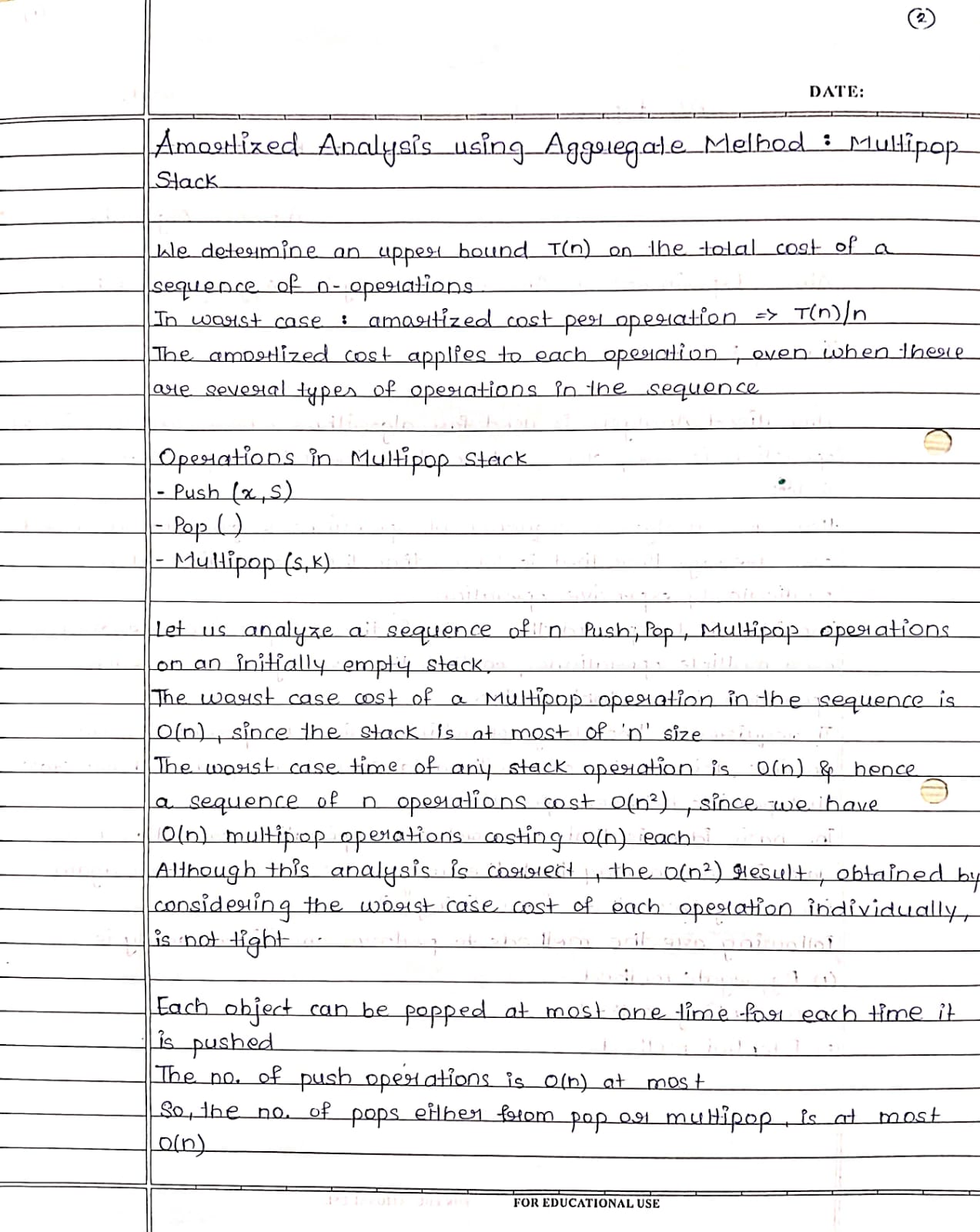
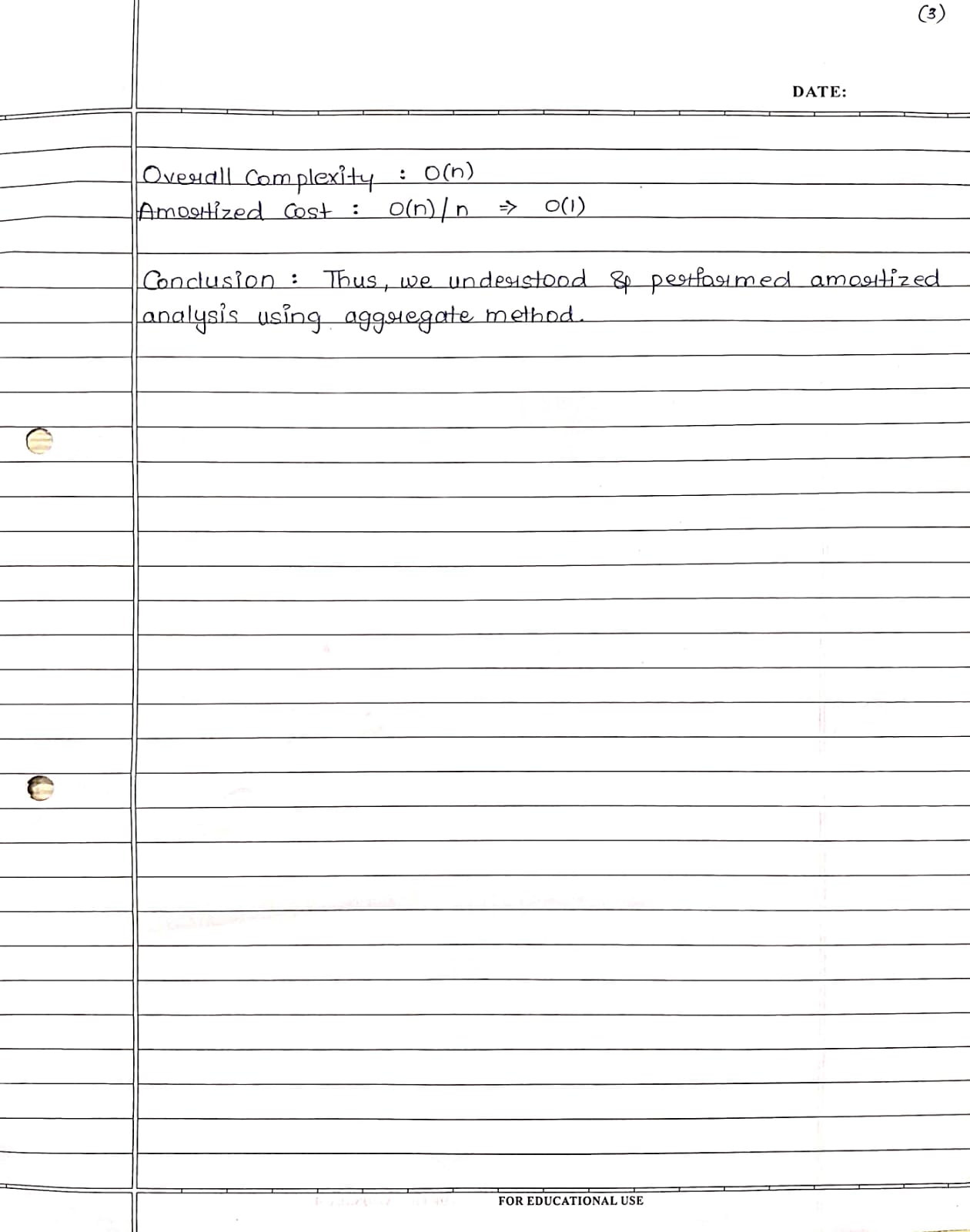
###### **Experiment 1A**





Code :

#include <bits/stdc++.h>

using namespace std;

class Multipop {

public:

int c = 0;

vector <int> stack;

void push(int a) {

stack.push\_back(a);

for (int i: stack) cout << "|" << i;

cout << "|" << endl;

c++;

}

void pop(int k) {

for (int i = 0; i < k; i++) {

if (stack.size() > 0) {

cout << stack.back() << " popped." << endl;

stack.pop\_back();

for (int i: stack) cout << "|" << i;

cout << "|" << endl;

}

c++;

}

}

void multipop(vector<int> arr) {

for (int i = 0; i < arr.size(); i++) {

if(arr[i] >= stack.size()) {

cout << arr[i] << " pushed." << endl;

push(arr[i]);

}

else {

cout << "Multi pop called " << arr[i] << " times." << endl;

pop(arr[i]);

cout << arr[i] << " pushed." << endl;

push(arr[i]);

}

}

cout << "Asymptotic cost: " << c << endl;

cout << "Amortized cost: " << c / arr.size();

}

};

int main() {

int n;

cout << "Enter array size: ";

cin >> n;

vector <int> a(n,0);

cout << "Enter array elements: " << endl;;

for (int i = 0; i < n; i++) cin >> a[i];

Multipop m;

m.multipop(a);

return 0;

}

Output:

Enter array size: 5

Enter array elements:

5

6

7

2

9

5 pushed.

|5|

6 pushed.

|5|6|

7 pushed.

|5|6|7|

Multi pop called 2 times.

7 popped.

|5|6|

6 popped.

|5|

2 pushed.

|5|2|

9 pushed.

|5|2|9|

Asymptotic cost: 7

Amortized cost: 1

###### **Experiment 1B**

Code :

def accounting(n):

size=1

total=0

dcost=0

icost=0

bank=0

totalfinal=0

print('Elements\tDoubling Cost\tInsertion Cost\tTotal Cost\tBank')

for i in range(1,n+1):

icost=1

if i>size:

size\*=2

dcost=i-1

total=icost+dcost

totalfinal=total+totalfinal

bank+=(3-total)

print(i,'\t\t\t\t',dcost,'\t\t\t\t',icost,'\t\t\t\t',total,'\t\t',bank)

icost=0

dcost=0

return totalfinal/n

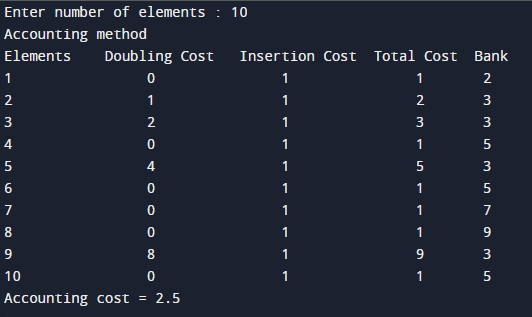
n=int(input('Enter number of elements : '))

print('Accounting method')

a=accounting(n)

print('Accounting cost =',a)

Output :



###### **Experiment 1C**

Code :

binaryStr = "0000"

binaryPrev = binaryStr

# cost to flip 0 -> 1 always = 1

setCost = 1

# cost to flip required 1s -> 0s

resetCost = 0

# total cost -> setCost + resetCost

totalCost = 0

potential = 0

potentialPrev = 0

amortized = 0

print("Element Set Reset Total Potential Amortized")

for i in range(8):

if i == 0:

setCost = 0

else:

setCost = 1

resetCost = 0

binaryPrev = binaryStr

binaryStr = bin(i)[2:].zfill(4)

for j in range(len(binaryStr)):

if binaryPrev[j] == '1' and binaryStr[j] == '0':

resetCost += 1

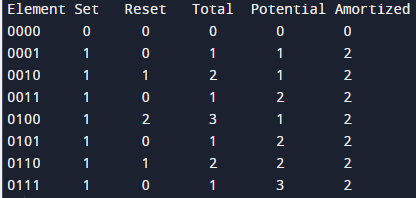
totalCost = resetCost + setCost

potentialPrev = potential

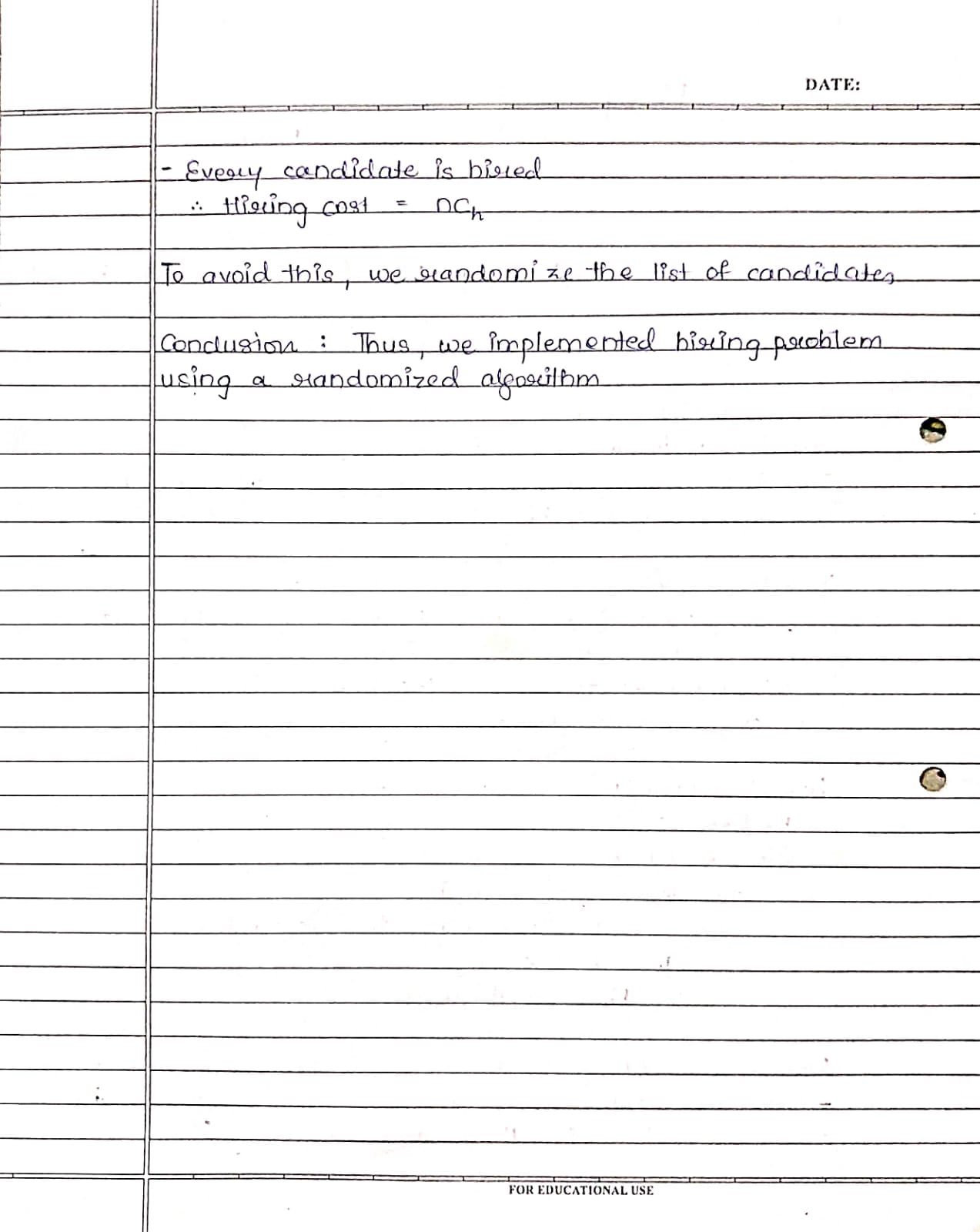
potential = potentialPrev - resetCost + setCost

amortized = totalCost + potential - potentialPrev

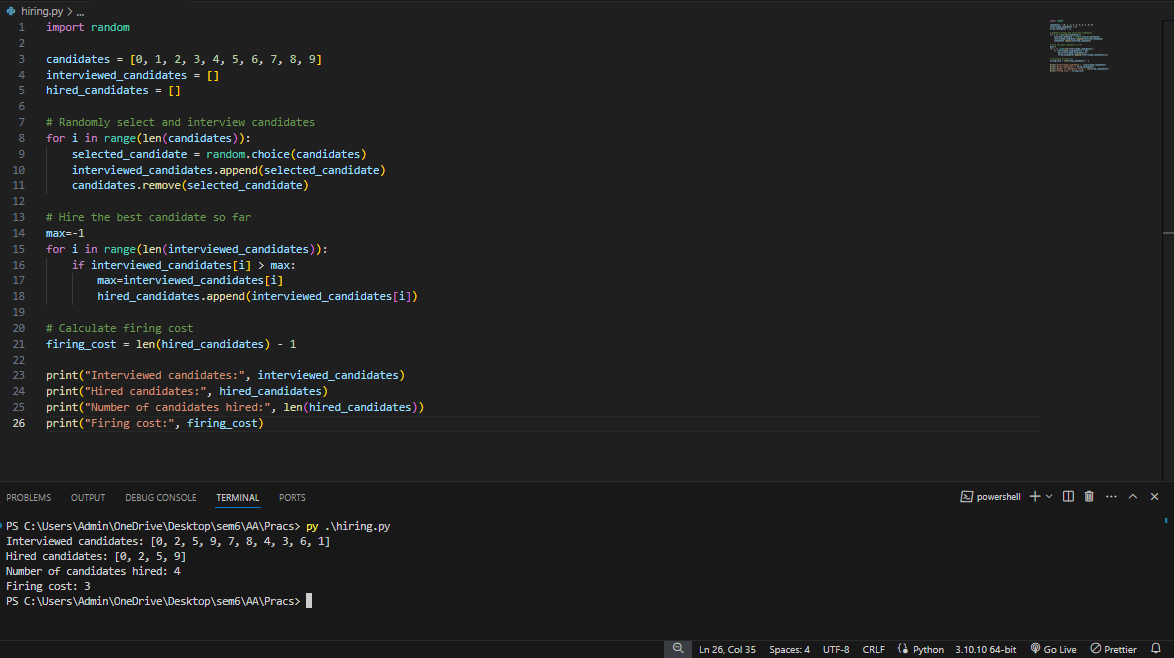
print(f"{binaryStr}\t {setCost}\t\t{resetCost}\t\t{totalCost}\t\t{potential}\t\t{amortized}")

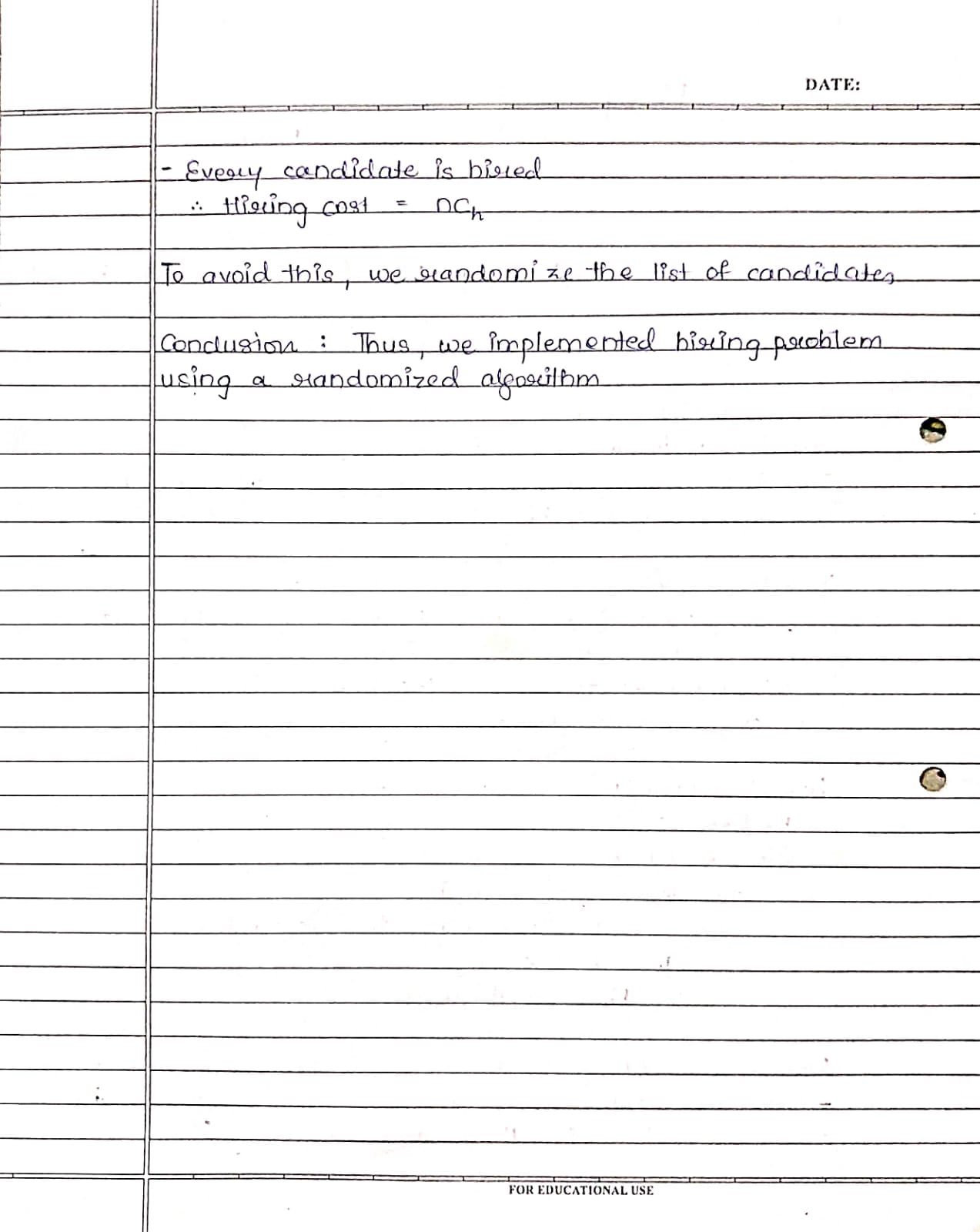


###### **Experiment 2**

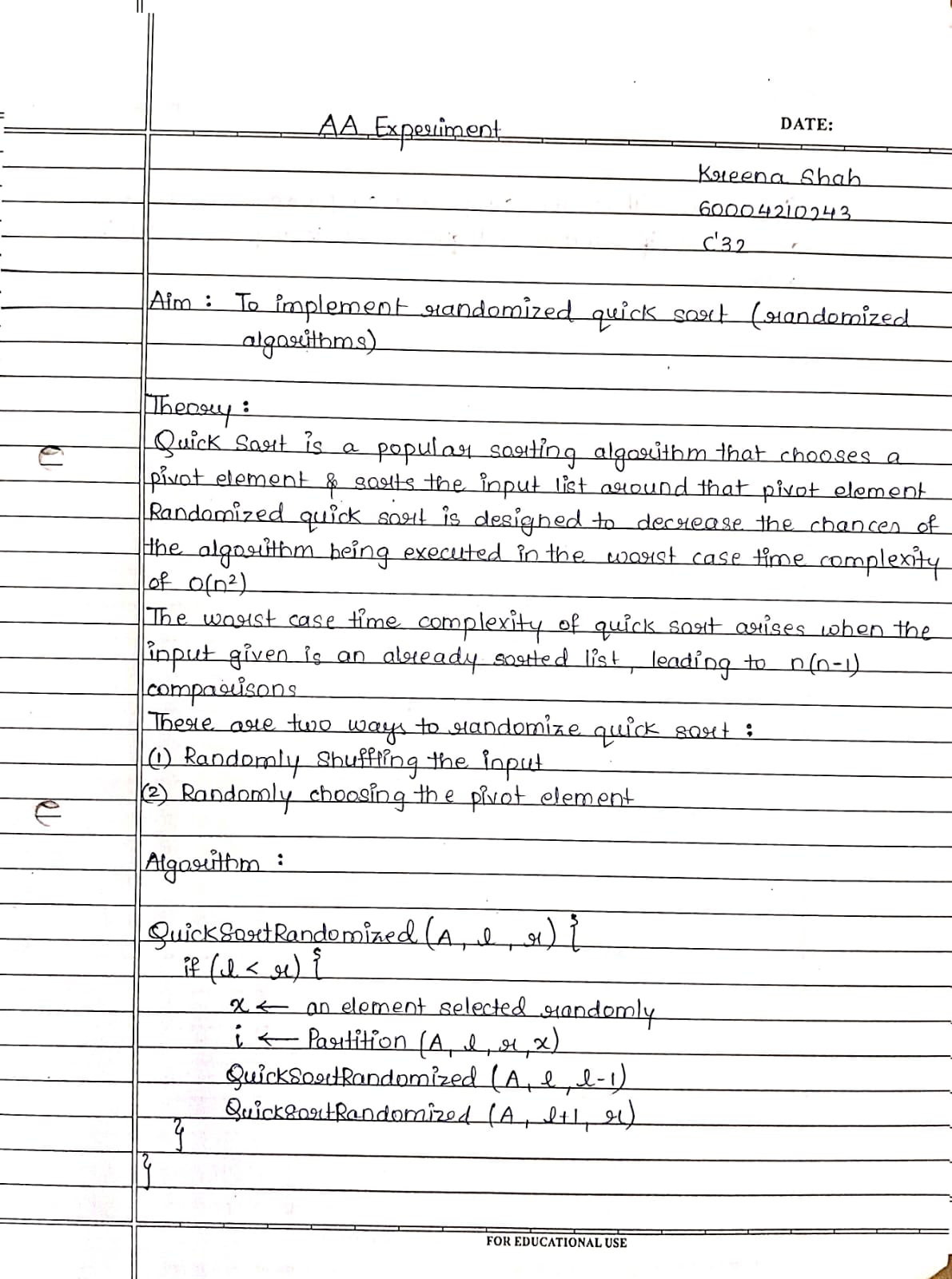
****

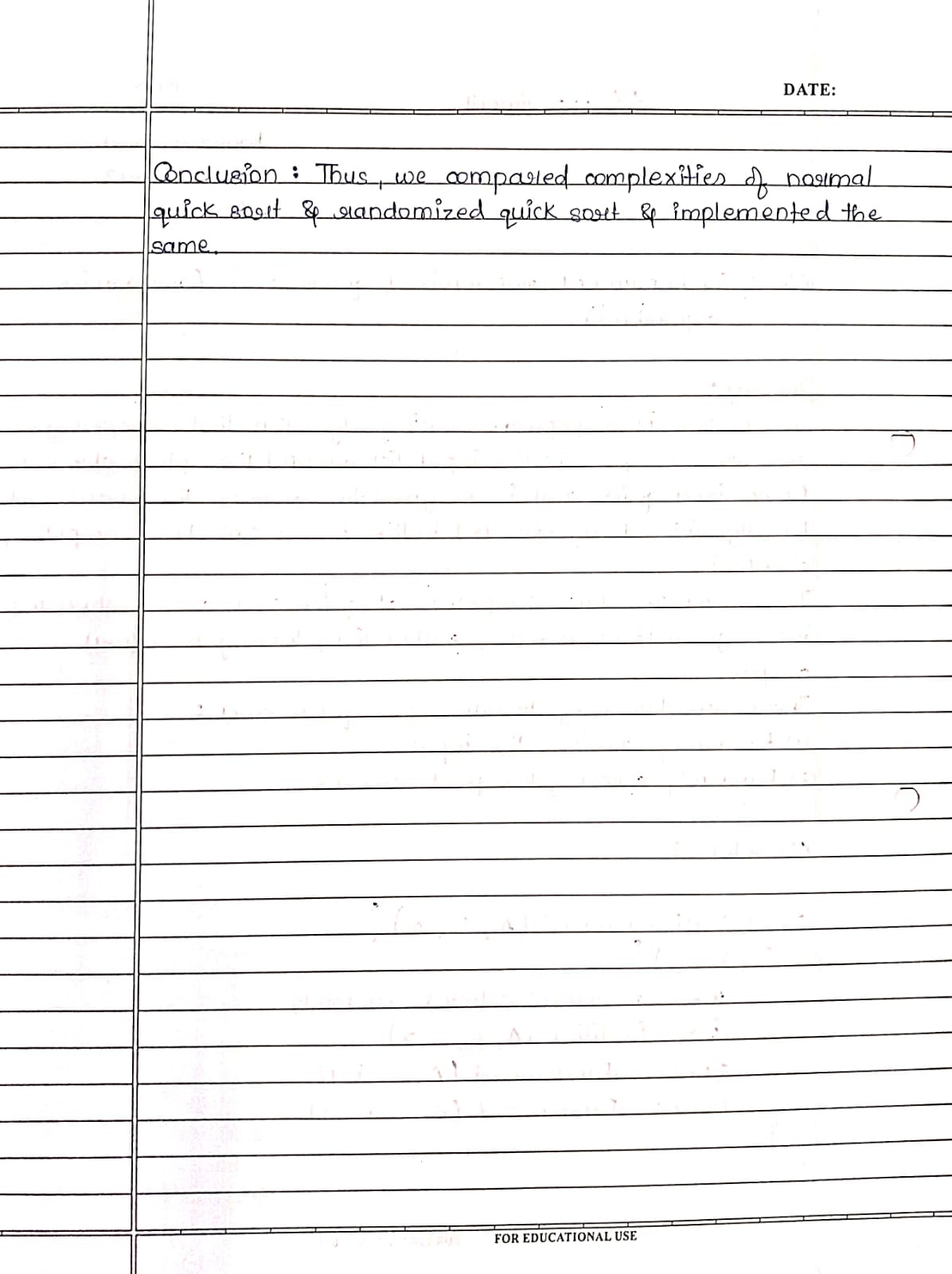
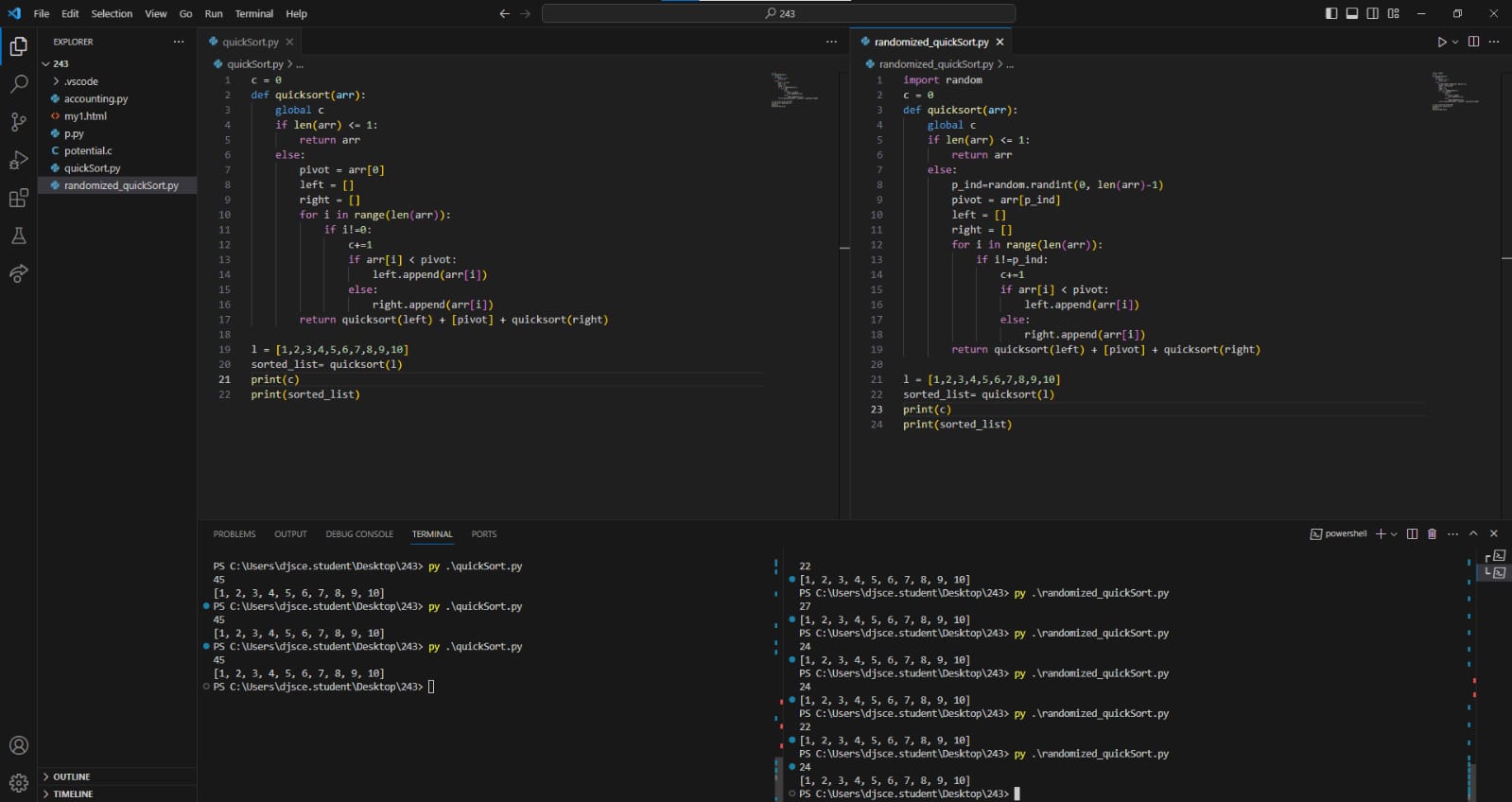
Code & Output :

****

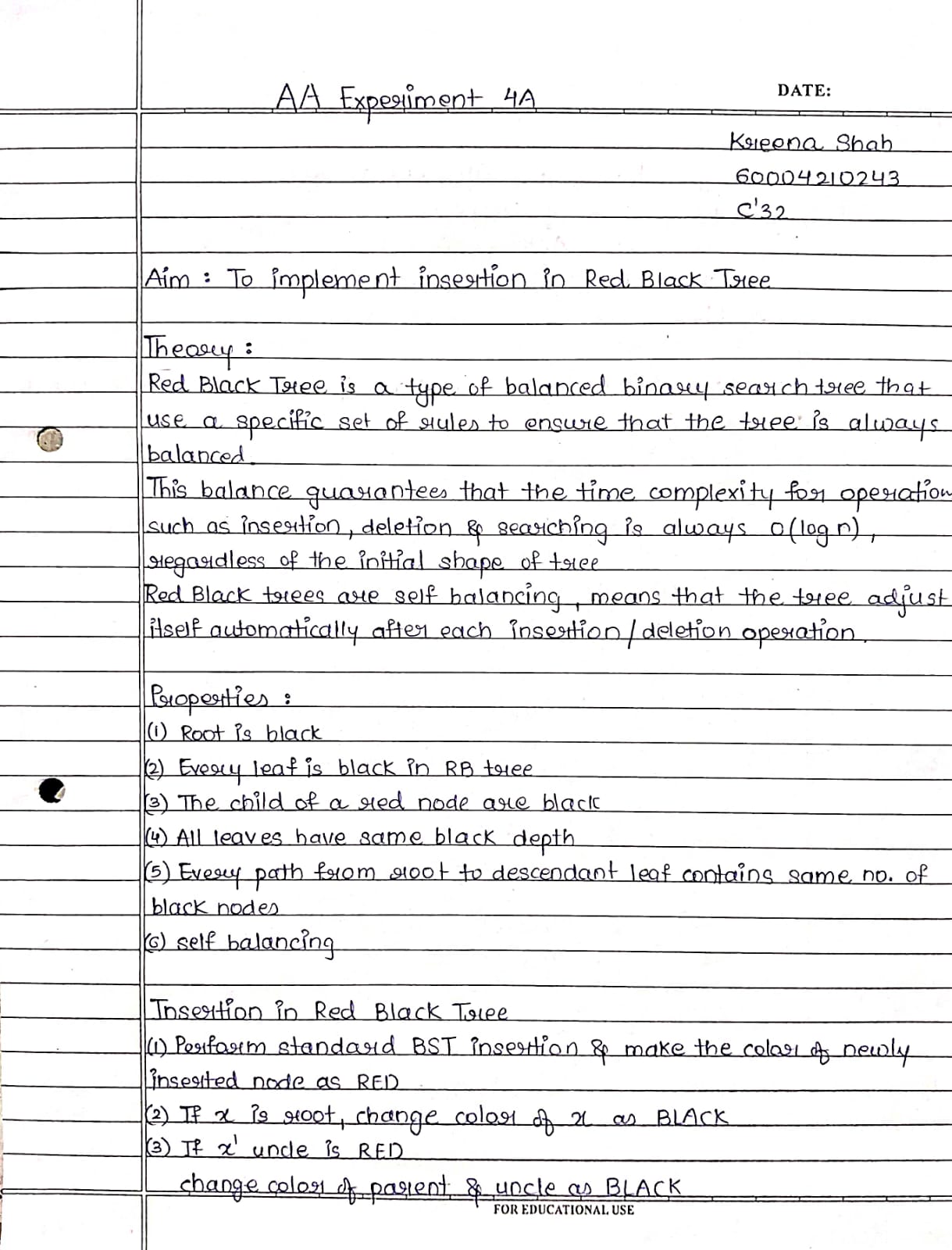
****

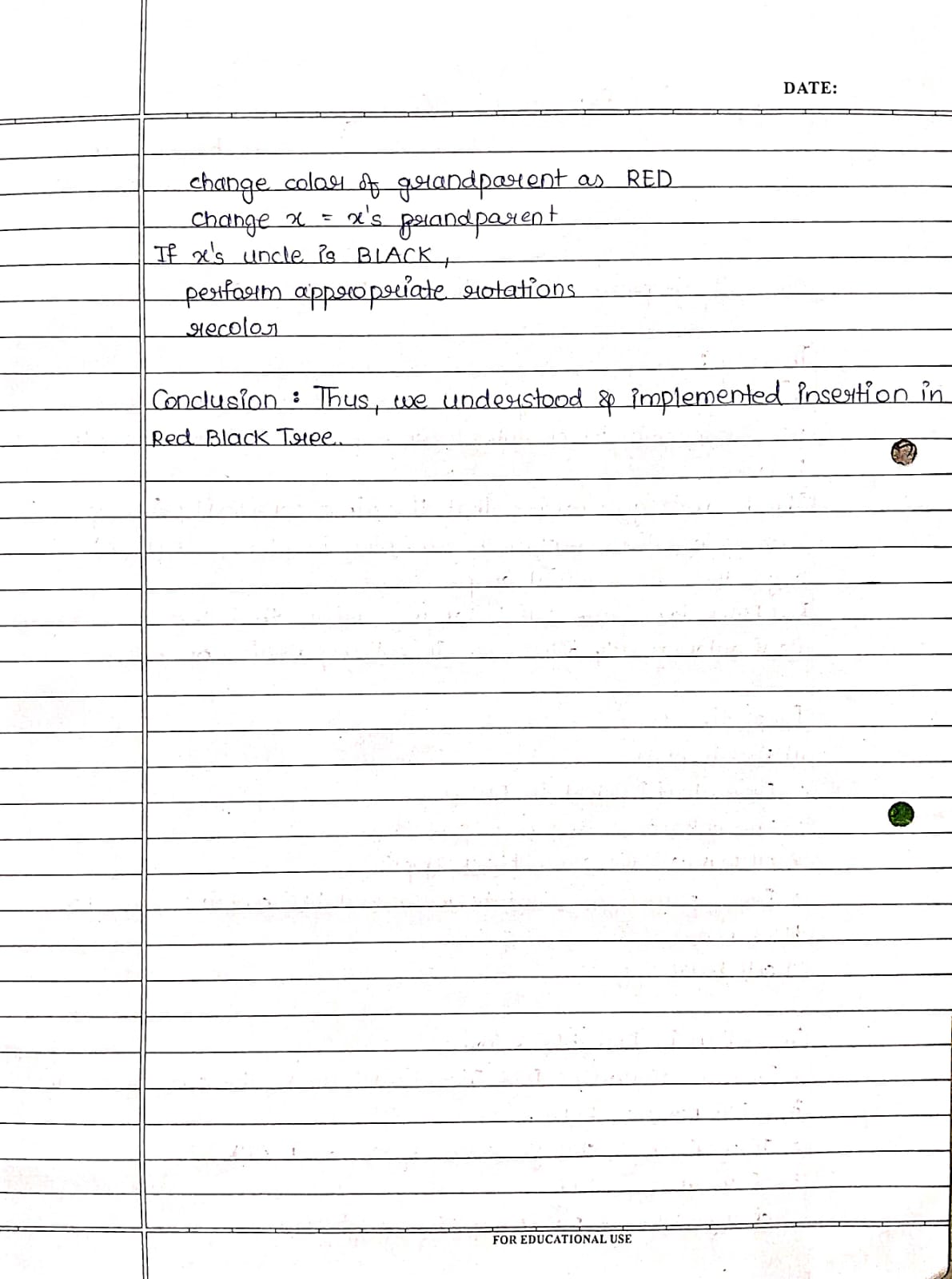
###### **Experiment 3**

****

Code & Output :****

###### **Experiment 4A**

****

****

from queue import Queue

class COLOR:

RED = 'RED'

BLACK = 'BLACK'

class Node:

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

self.val = val

self.color = COLOR.RED

self.left = None

self.right = None

self.parent = None

def uncle(self):

if self.parent is None or self.parent.parent is None:

return None

if self.parent.isOnLeft():

return self.parent.parent.right

else:

return self.parent.parent.left

def isOnLeft(self):

return self == self.parent.left

def sibling(self):

if self.parent is None:

return None

if self.isOnLeft():

return self.parent.right

else:

return self.parent.left

def moveDown(self, new\_parent):

if self.parent is not None:

if self.isOnLeft():

self.parent.left = new\_parent

else:

self.parent.right = new\_parent

new\_parent.parent = self.parent

self.parent = new\_parent

def hasRedChild(self):

return (self.left is not None and self.left.color == COLOR.RED) or \

(self.right is not None and self.right.color == COLOR.RED)

class RBTree:

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

self.root = None

def leftRotate(self, x):

new\_parent = x.right

if x == self.root:

self.root = new\_parent

x.moveDown(new\_parent)

x.right = new\_parent.left

if new\_parent.left is not None:

new\_parent.left.parent = x

new\_parent.left = x

def rightRotate(self, x):

new\_parent = x.left

if x == self.root:

self.root = new\_parent

x.moveDown(new\_parent)

x.left = new\_parent.right

if new\_parent.right is not None:

new\_parent.right.parent = x

new\_parent.right = x

def swapColors(self, x1, x2):

temp = x1.color

x1.color = x2.color

x2.color = temp

def swapValues(self, u, v):

temp = u.val

u.val = v.val

v.val = temp

def fixRedRed(self, x):

if x == self.root:

x.color = COLOR.BLACK

return

parent = x.parent

grandparent = parent.parent

uncle = x.uncle()

if parent.color != COLOR.BLACK:

if uncle is not None and uncle.color == COLOR.RED:

parent.color = COLOR.BLACK

uncle.color = COLOR.BLACK

grandparent.color = COLOR.RED

self.fixRedRed(grandparent)

else:

if parent.isOnLeft():

if x.isOnLeft():

self.swapColors(parent, grandparent)

else:

self.leftRotate(parent)

self.swapColors(x, grandparent)

self.rightRotate(grandparent)

else:

if x.isOnLeft():

self.rightRotate(parent)

self.swapColors(x, grandparent)

else:

self.swapColors(parent, grandparent)

self.leftRotate(grandparent)

def BSTreplace(self, x):

if x.left is not None and x.right is not None:

return self.successor(x.right)

if x.left is None and x.right is None:

return None

if x.left is not None:

return x.left

else:

return x.right

def levelOrder(self, x):

if x is None:

return

q = Queue()

q.put(x)

while not q.empty():

curr = q.get()

print(curr.val, end=" ")

if curr.left is not None:

q.put(curr.left)

if curr.right is not None:

q.put(curr.right)

def inorder(self, x):

if x is None:

return

self.inorder(x.left)

print(x.val, end=" ")

self.inorder(x.right)

def getRoot(self):

return self.root

def search(self, n):

temp = self.root

while temp is not None:

if n < temp.val:

if temp.left is None:

break

else:

temp = temp.left

elif n == temp.val:

break

else:

if temp.right is None:

break

else:

temp = temp.right

return temp

def insert(self, n):

newNode = Node(n)

if self.root is None:

newNode.color = COLOR.BLACK

self.root = newNode

else:

temp = self.search(n)

if temp.val == n:

return

newNode.parent = temp

if n < temp.val:

temp.left = newNode

else:

temp.right = newNode

self.fixRedRed(newNode)

def printInOrder(self):

print("Inorder:")

if self.root is None:

print("Tree is empty")

else:

self.inorder(self.root)

print()

def printLevelOrder(self):

print("Level order:")

if self.root is None:

print("Tree is empty")

else:

self.levelOrder(self.root)

print()

tree = RBTree()

tree.insert(7)

tree.insert(3)

tree.insert(18)

tree.insert(10)

tree.insert(22)

tree.insert(8)

tree.insert(11)

tree.insert(26)

tree.insert(2)

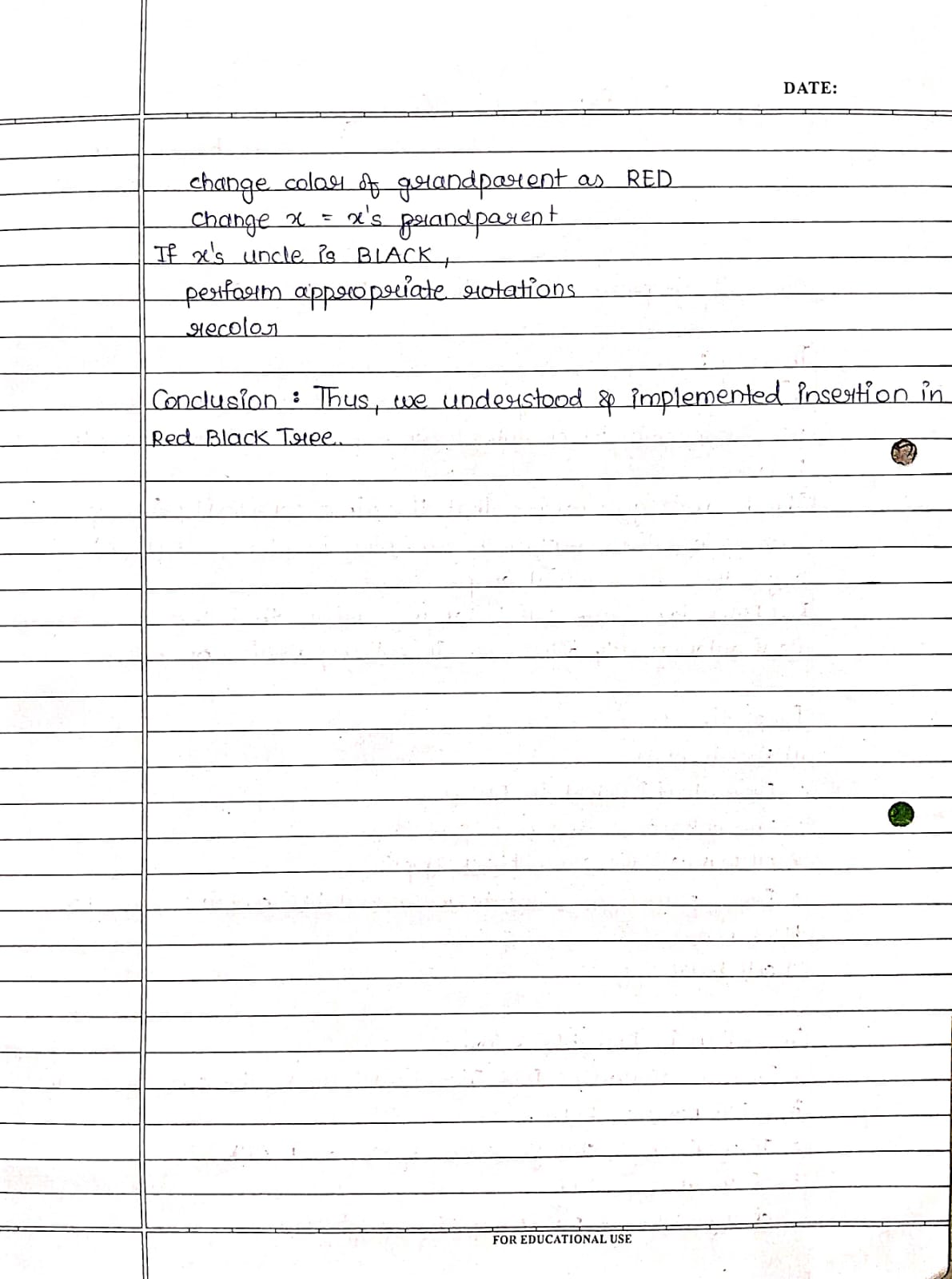
tree.insert(6)

tree.insert(13)

tree.printInOrder()

tree.printLevelOrder()



****

###### **Experiment 4B**

from queue import Queue

# Enumeration for colors

class COLOR:

RED = 'RED'

BLACK = 'BLACK'

class Node:

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

# Initialize node attributes

self.val = val

self.color = COLOR.RED # New nodes are always red

self.left = None

self.right = None

self.parent = None

def uncle(self):

# Returns the uncle of the node

if self.parent is None or self.parent.parent is None:

return None

if self.parent.isOnLeft():

return self.parent.parent.right

else:

return self.parent.parent.left

def isOnLeft(self):

# Checks if the node is the left child of its parent

return self == self.parent.left

def sibling(self):

# Returns the sibling of the node

if self.parent is None:

return None

if self.isOnLeft():

return self.parent.right

else:

return self.parent.left

def moveDown(self, new\_parent):

# Moves the node down by changing its parent

if self.parent is not None:

if self.isOnLeft():

self.parent.left = new\_parent

else:

self.parent.right = new\_parent

new\_parent.parent = self.parent

self.parent = new\_parent

def hasRedChild(self):

# Checks if the node has a red child

return (self.left is not None and self.left.color == COLOR.RED) or \

(self.right is not None and self.right.color == COLOR.RED)

class RBTree:

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

# Initialize Red-Black Tree with an empty root

self.root = None

def leftRotate(self, x):

# Performs a left rotation around the given node

new\_parent = x.right

if x == self.root:

self.root = new\_parent

x.moveDown(new\_parent)

x.right = new\_parent.left

if new\_parent.left is not None:

new\_parent.left.parent = x

new\_parent.left = x

def rightRotate(self, x):

# Performs a right rotation around the given node

new\_parent = x.left

if x == self.root:

self.root = new\_parent

x.moveDown(new\_parent)

x.left = new\_parent.right

if new\_parent.right is not None:

new\_parent.right.parent = x

new\_parent.right = x

def swapColors(self, x1, x2):

# Swaps the colors of two nodes

temp = x1.color

x1.color = x2.color

x2.color = temp

def swapValues(self, u, v):

# Swaps the values of two nodes

temp = u.val

u.val = v.val

v.val = temp

def fixRedRed(self, x):

# Fixes the red-red violation in the tree

if x == self.root:

x.color = COLOR.BLACK

return

parent = x.parent

grandparent = parent.parent

uncle = x.uncle()

if parent.color != COLOR.BLACK:

if uncle is not None and uncle.color == COLOR.RED:

# Uncle is red, perform recoloring and recurse

parent.color = COLOR.BLACK

uncle.color = COLOR.BLACK

grandparent.color = COLOR.RED

self.fixRedRed(grandparent)

else:

# Perform rotations based on the cases

if parent.isOnLeft():

if x.isOnLeft():

self.swapColors(parent, grandparent)

else:

self.leftRotate(parent)

self.swapColors(x, grandparent)

self.rightRotate(grandparent)

else:

if x.isOnLeft():

self.rightRotate(parent)

self.swapColors(x, grandparent)

else:

self.swapColors(parent, grandparent)

self.leftRotate(grandparent)

def successor(self, x):

# Finds the in-order successor of the given node

temp = x

while temp.left is not None:

temp = temp.left

return temp

def BSTreplace(self, x):

# Finds the replacement node in BST for the given node

if x.left is not None and x.right is not None:

return self.successor(x.right)

if x.left is None and x.right is None:

return None

if x.left is not None:

return x.left

else:

return x.right

def deleteNode(self, v):

# Deletes the given node from the tree

u = self.BSTreplace(v)

uvBlack = (u is None or u.color == COLOR.BLACK) and (v.color == COLOR.BLACK)

parent = v.parent

if u is None:

# Node to be deleted is a leaf or has only one child

if v == self.root:

# If the node is the root

self.root = None

else:

# Detach v from the tree and move u up

if uvBlack:

# u and v both black, fix double black at v

self.fixDoubleBlack(v)

else:

# u or v red, color u black

if v.sibling() is not None:

v.sibling().color = COLOR.RED

if v.isOnLeft():

parent.left = None

else:

parent.right = None

del v # Delete the node

return

if v.left is None or v.right is None:

# Node to be deleted has only one child

if v == self.root:

# If the node is the root

v.val = u.val

v.left = v.right = None

del u

else:

# Detach v from the tree and move u up

if v.isOnLeft():

parent.left = u

else:

parent.right = u

del v

u.parent = parent

if uvBlack:

# u and v both black, fix double black at u

self.fixDoubleBlack(u)

else:

# u or v red, color u black

u.color = COLOR.BLACK

else:

# Node to be deleted has two children, swap values with successor and recurse

self.swapValues(u, v)

self.deleteNode(u)

# Fixes the double black violation in the tree

def fixDoubleBlack(self, x):

# Reached root

if x == self.root:

return

sibling = x.sibling()

parent = x.parent

# No sibling, double black pushed up

if sibling is None:

self.fixDoubleBlack(parent)

else:

# Sibling red

if sibling.color == COLOR.RED:

parent.color = COLOR.RED

sibling.color = COLOR.BLACK

# Left case

if sibling.isOnLeft():

self.rightRotate(parent)

# Right case

else:

self.leftRotate(parent)

self.fixDoubleBlack(x)

else:

# Sibling black

if sibling.hasRedChild():

# At least 1 red child

if sibling.left is not None and sibling.left.color == COLOR.RED:

# Left Left

if sibling.isOnLeft():

sibling.left.color = sibling.color

sibling.color = parent.color

self.rightRotate(parent)

# Right Left

else:

sibling.left.color = parent.color

self.rightRotate(sibling)

self.leftRotate(parent)

else:

# Left Right

if sibling.isOnLeft():

sibling.right.color = parent.color

self.leftRotate(sibling)

self.rightRotate(parent)

# Right Right

else:

sibling.right.color = sibling.color

sibling.color = parent.color

self.leftRotate(parent)

# 2 black children

else:

sibling.color = COLOR.RED

if parent.color == COLOR.BLACK:

self.fixDoubleBlack(parent)

else:

parent.color = COLOR.BLACK

# Prints the level order traversal of the tree starting from the given node

def levelOrder(self, x):

if x is None:

return

q = Queue()

q.put(x)

while not q.empty():

curr = q.get()

print(curr.val, end=" ")

if curr.left is not None:

q.put(curr.left)

if curr.right is not None:

q.put(curr.right)

# Prints the in-order traversal of the tree starting from the given node

def inorder(self, x):

if x is None:

return

self.inorder(x.left)

print(x.val, end=" ")

self.inorder(x.right)

def getRoot(self):

return self.root

# Searches for a given value in the tree and returns the node if found

# Otherwise, returns the last node encountered while traversing

def search(self, n):

temp = self.root

while temp is not None:

if n < temp.val:

if temp.left is None:

break

else:

temp = temp.left

elif n == temp.val:

break

else:

if temp.right is None:

break

else:

temp = temp.right

return temp

# Inserts the given value into the tree

def insert(self, n):

newNode = Node(n)

# When the tree is empty

if self.root is None:

newNode.color = COLOR.BLACK

self.root = newNode

else:

temp = self.search(n)

# Value already exists, return

if temp.val == n:

return

# Connect the new node to the correct node

newNode.parent = temp

if n < temp.val:

temp.left = newNode

else:

temp.right = newNode

# Fix red-red violation if it exists

self.fixRedRed(newNode)

# Deletes the node with the given value from the tree

def deleteByVal(self, n):

# Tree is empty

if self.root is None:

return

v = self.search(n)

if v.val != n:

print(f"No node found to delete with value: {n}")

return

self.deleteNode(v)

# Prints the in-order traversal of the tree

def printInOrder(self):

print("Inorder:")

if self.root is None:

print("Tree is empty")

else:

self.inorder(self.root)

print()

# Prints the level order traversal of the tree

def printLevelOrder(self):

print("Level order:")

if self.root is None:

print("Tree is empty")

else:

self.levelOrder(self.root)

print()

# Test the RBTree

tree = RBTree()

tree.insert(7)

tree.insert(3)

tree.insert(18)

tree.insert(10)

tree.insert(22)

tree.insert(8)

tree.insert(11)

tree.insert(26)

tree.insert(2)

tree.insert(6)

tree.insert(13)

tree.printInOrder()

tree.printLevelOrder()

print("Deleting 18, 11, 3, 10, 22")

tree.deleteByVal(18)

tree.deleteByVal(11)

tree.deleteByVal(3)

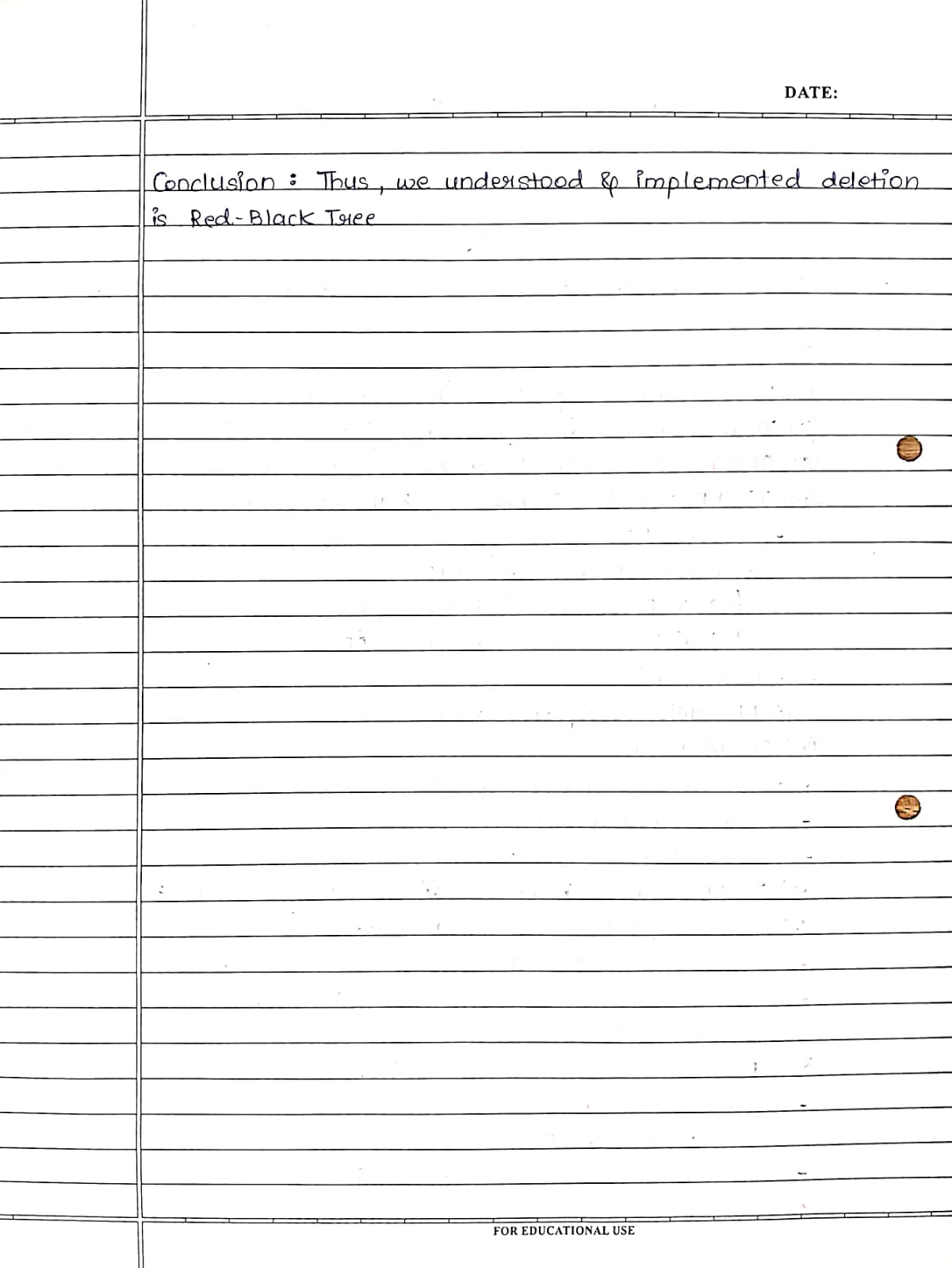
tree.deleteByVal(10)

tree.deleteByVal(22)

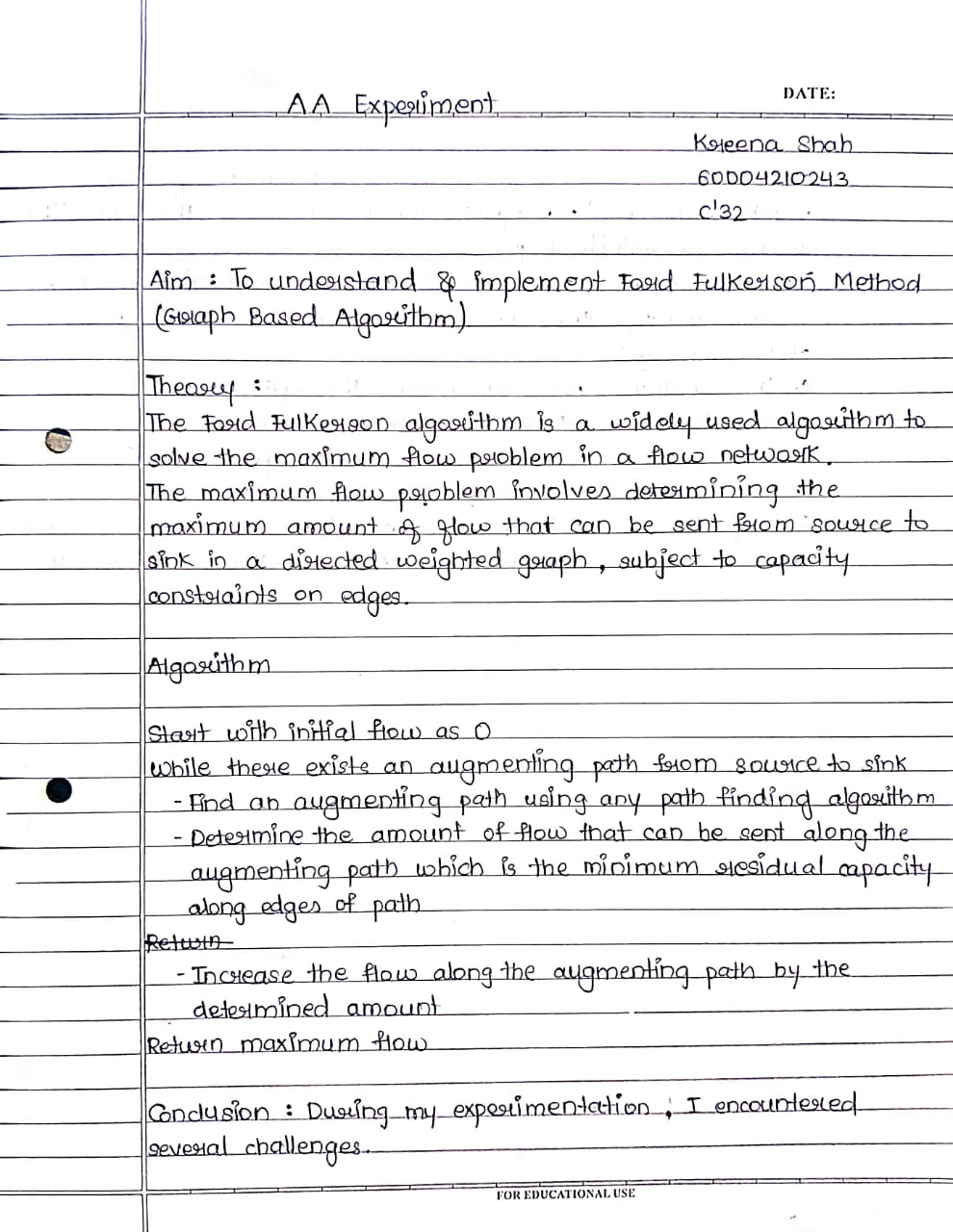
tree.printInOrder()

tree.printLevelOrder()

****

****

###### **Experiment 8**



Code :

from collections import defaultdict

class Graph:

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

self.graph = graph

self. ROW = len(graph)

def BFS(self, s, t, parent):

visited = [False]\*(self.ROW)

queue = []

queue.append(s)

visited[s] = True

while queue:

u = queue.pop(0)

for ind, val in enumerate(self.graph[u]):

if visited[ind] == False and val > 0:

queue.append(ind)

visited[ind] = True

parent[ind] = u

if ind == t:

return True

return False

def FordFulkerson(self, source, sink):

parent = [-1]\*(self.ROW)

max\_flow = 0

while self.BFS(source, sink, parent) :

path\_flow = float("Inf")

s = sink

while(s != source):

path\_flow = min (path\_flow, self.graph[parent[s]][s])

s = parent[s]

max\_flow += path\_flow

v = sink

while(v != source):

u = parent[v]

self.graph[u][v] -= path\_flow

self.graph[v][u] += path\_flow

v = parent[v]

return max\_flow

graph = [[0, 16, 13, 0, 0, 0],

[0, 0, 10, 12, 0, 0],

[0, 4, 0, 0, 14, 0],

[0, 0, 9, 0, 0, 20],

[0, 0, 0, 7, 0, 4],

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

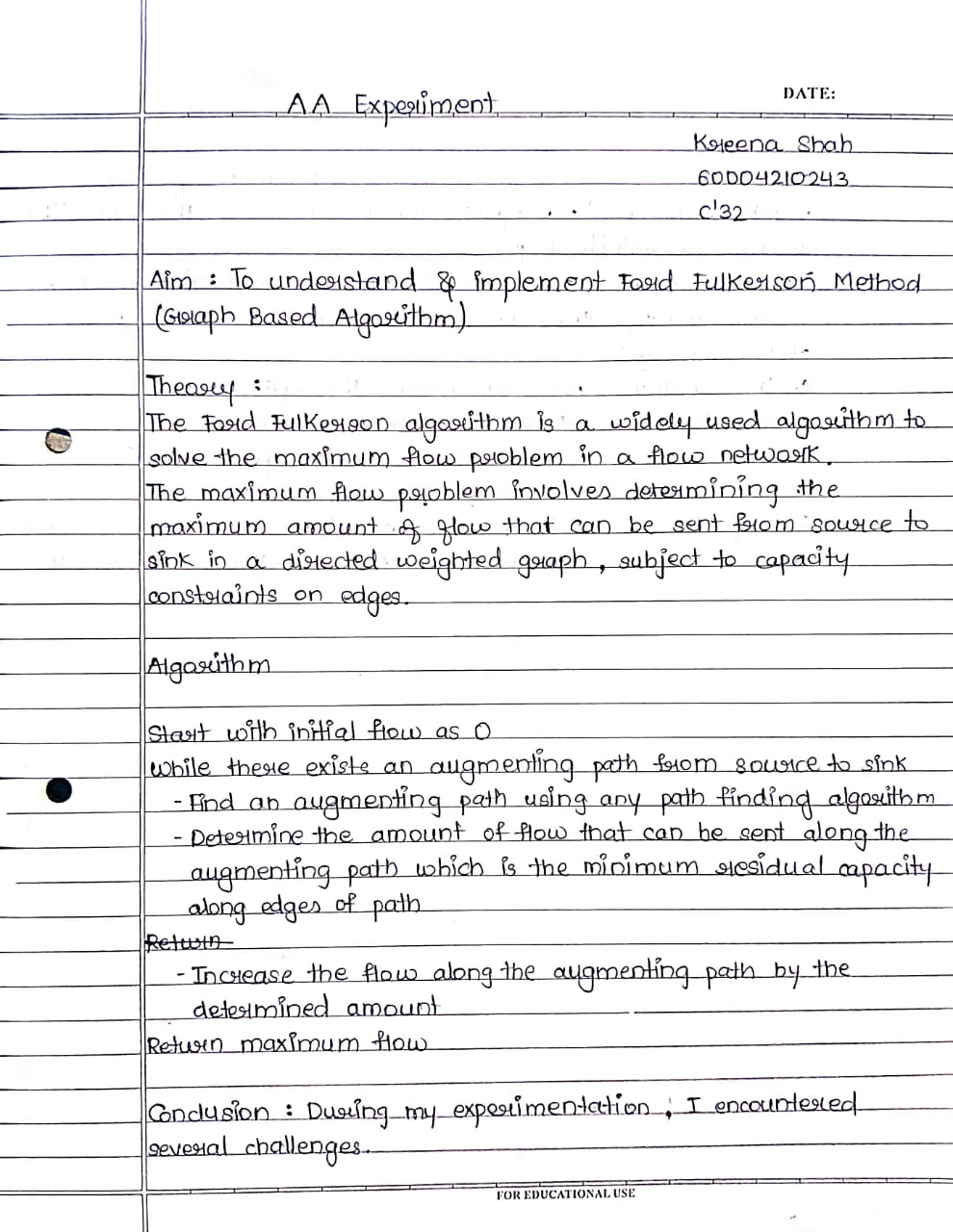
g = Graph(graph)

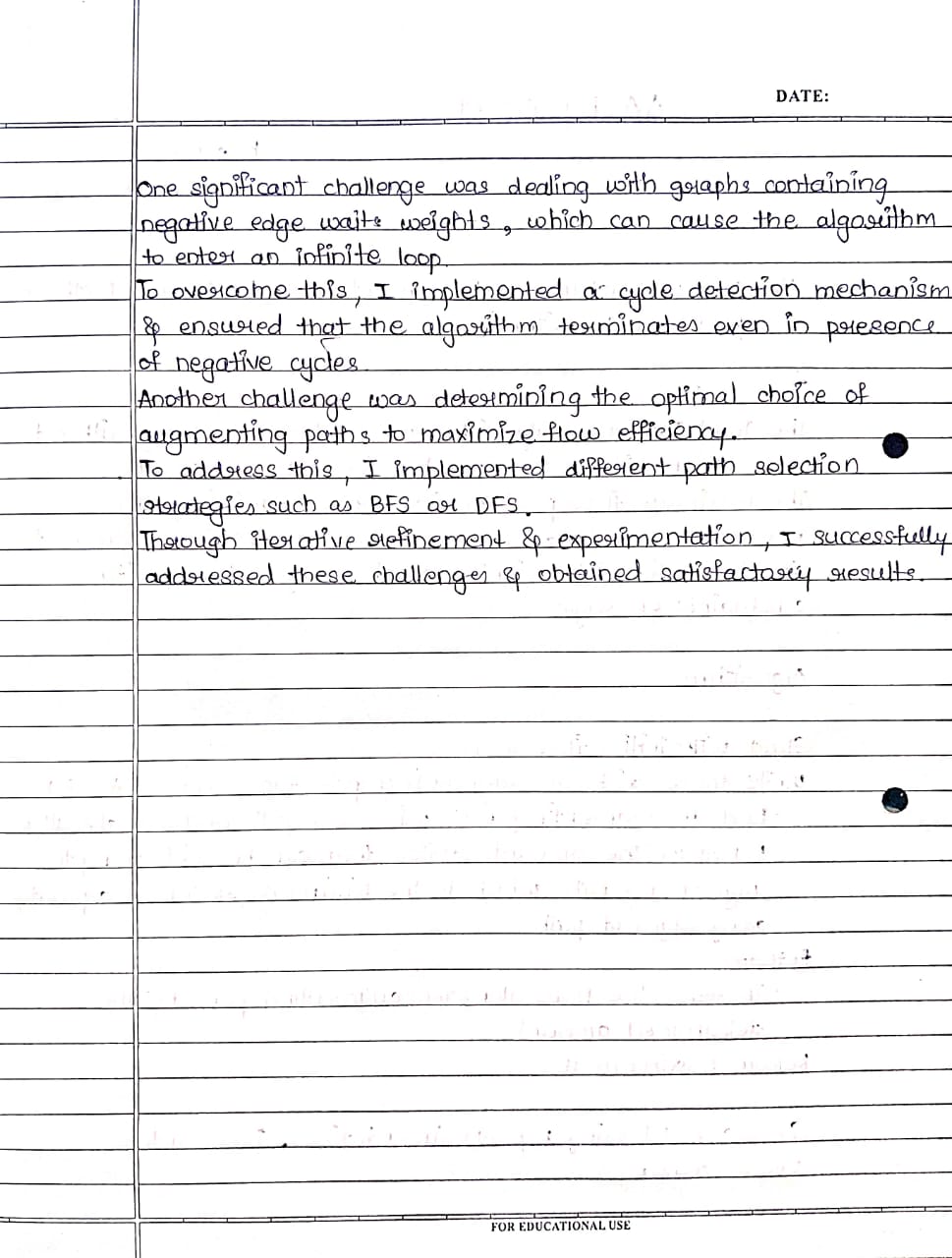
source = 0; sink = 5

print ("The maximum possible flow is %d " % g.FordFulkerson(source, sink))

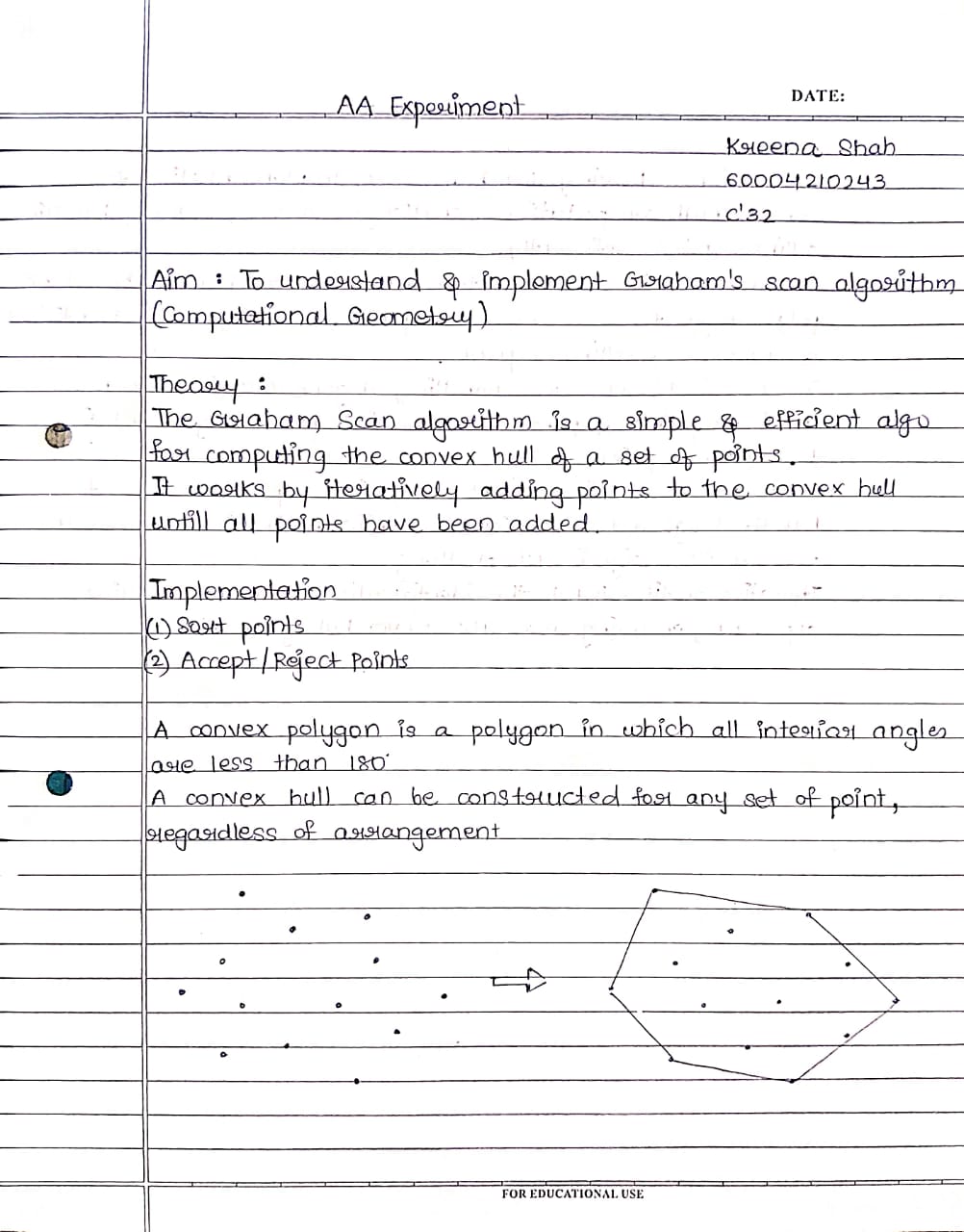
Output :







###### **Experiment 9**



Code :

from functools import cmp\_to\_key

class Point:

def \_\_init\_\_(self, x = None, y = None):

self.x = x

self.y = y

p0 = Point(0, 0)

def nextToTop(S):

return S[-2]

def distSq(p1, p2):

return ((p1.x - p2.x) \* (p1.x - p2.x) +

(p1.y - p2.y) \* (p1.y - p2.y))

def orientation(p, q, r):

val = ((q.y - p.y) \* (r.x - q.x) -

(q.x - p.x) \* (r.y - q.y))

if val == 0:

return 0

elif val > 0:

return 1

else:

return 2

def compare(p1, p2):

o = orientation(p0, p1, p2)

if o == 0:

if distSq(p0, p2) >= distSq(p0, p1):

return -1

else:

return 1

else:

if o == 2:

return -1

else:

return 1

def convexHull(points, n):

ymin = points[0].y

min = 0

for i in range(1, n):

y = points[i].y

if ((y < ymin) or

(ymin == y and points[i].x < points[min].x)):

ymin = points[i].y

min = i

points[0], points[min] = points[min], points[0]

p0 = points[0]

points = sorted(points, key=cmp\_to\_key(compare))

m = 1

for i in range(1, n):

while ((i < n - 1) and

(orientation(p0, points[i], points[i + 1]) == 0)):

i += 1

points[m] = points[i]

m += 1

if m < 3:

return

S = []

S.append(points[0])

S.append(points[1])

S.append(points[2])

for i in range(3, m):

while ((len(S) > 1) and

(orientation(nextToTop(S), S[-1], points[i]) != 2)):

S.pop()

S.append(points[i])

while S:

p = S[-1]

print("(" + str(p.x) + ", " + str(p.y) + ")")

S.pop()

input\_points = [(0, 3), (1, 1), (2, 2), (4, 4),

(0, 0), (1, 2), (3, 1), (3, 3)]

points = []

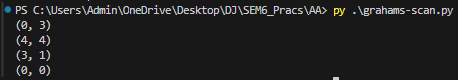
for point in input\_points:

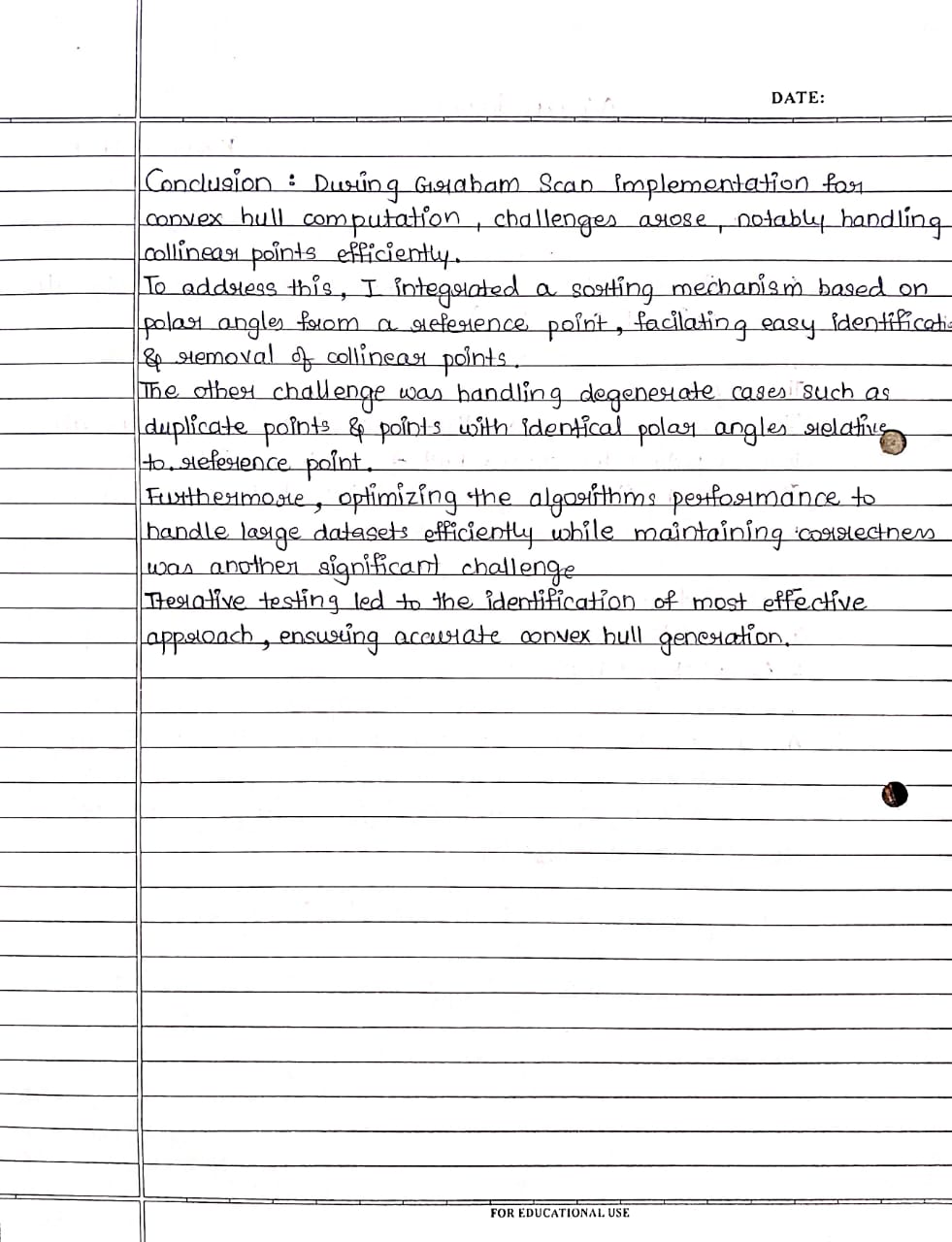
points.append(Point(point[0], point[1]))

n = len(points)

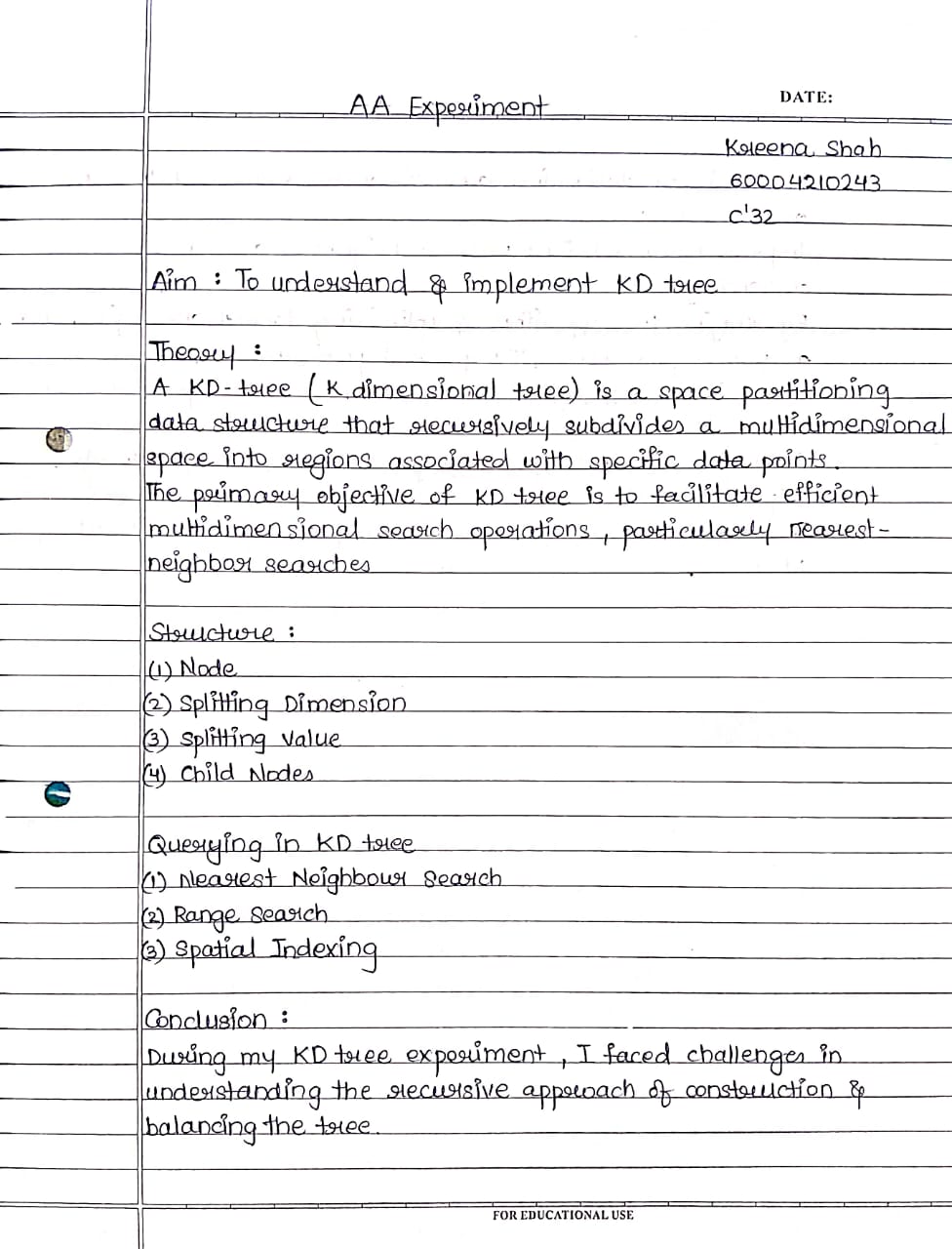
convexHull(points, n)

Output :





###### **Experiment 10**



Code :

from sklearn.neighbors import KDTree

import numpy as np

data = np.array([[2, 3], [5, 4], [9, 6], [4, 7], [8, 1], [7, 2]])

kdtree = KDTree(data, leaf\_size=30)

query\_point = np.array([[9, 2]])

distances, indices = kdtree.query(query\_point, k=2)

print("Query Point:", query\_point)

print("Nearest Neighbors:")

for i, idx in enumerate(indices[0]):

print(f"Neighbor {i + 1}: {data[idx]}, Distance: {distances[0][i]}")

Output :

