09/10/2021

Data Structure and Algorithms

Applied Prep Course

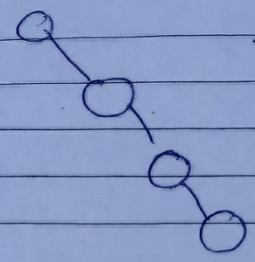
SATYAM SETH

PART-3

AVL Tree (Adelson-Velsky-Landis)

Page No.: 210
Date: / /

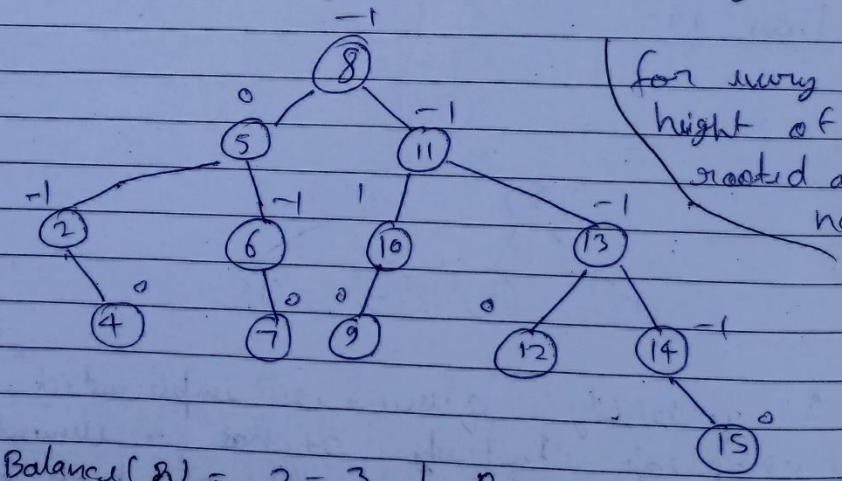
- Major problem with Binary Search Tree worst case insertion / search and deletion is O(n).



- AVL guarantee - Insert / delete / search → Θ(log n)
- AVL is self-balancing Binary Search Tree.

Balance(n) = height (LST of n) - height (RST of n)

Eg.



for every node height of the tree rooted at that node.

Balance(8) = 2 - 3.
Balance(5) = 1 - 1 = 0

Balance(14) = -1 - 0
= -1

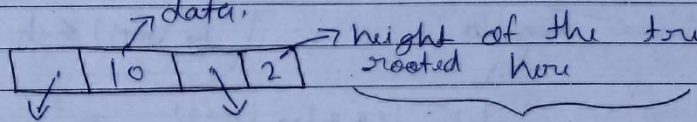
- Every node balance should be -1, 0 or 1.

Balance(n) ∈ {-1, 0, 1} ∀ nodes n in T

Page No.: 211
Date: / /

Height of a null tree = -1
Height of a single node Tree = 0

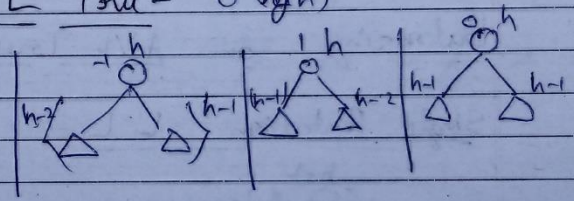
Node of an AVL Tree-



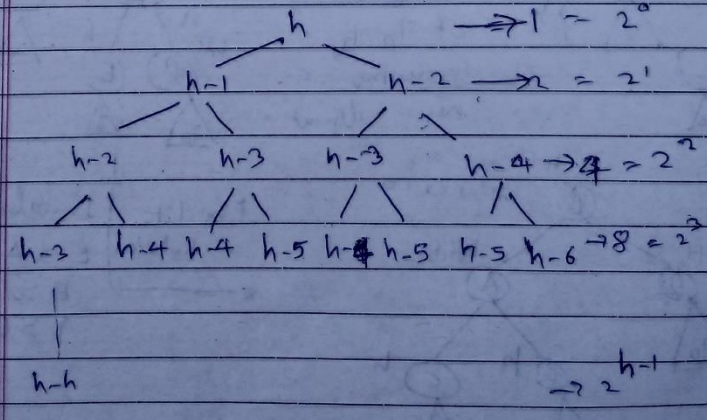
It helps us to compute Balance very fast

Height of AVL Tree- O(lgn)

n(h) = number of node in an AVL tree



$n(h) = n(h-1) + n(h-2) + 1$



$\rightarrow 1 = 2^0$

$\rightarrow 2 = 2^1$

$\rightarrow 4 = 2^2$

$\rightarrow 8 = 2^3$

$\rightarrow 2^{h-1}$

$2^0 + 2^1 + 2^2 + 2^3 + \cdots + 2^{h-1} = 2^h - 1 = n(h)$

height $h = \theta(\lg n)$ ← n nodes

Page No.: 212
Date: 12/11/20

$n(h) = n(h-1) + n(h-2) + 1$

Base Case: $n(0) = 1$, $n(1) = 2$

$Fib(n) = Fib(n-1) + Fib(n-2)$

$Fib(0) = 0$

$Fib(1) = 1$

AVL

n-nodes.

$\log_2(n+1) \leq h \leq c \cdot \log_2(n+2) + b$

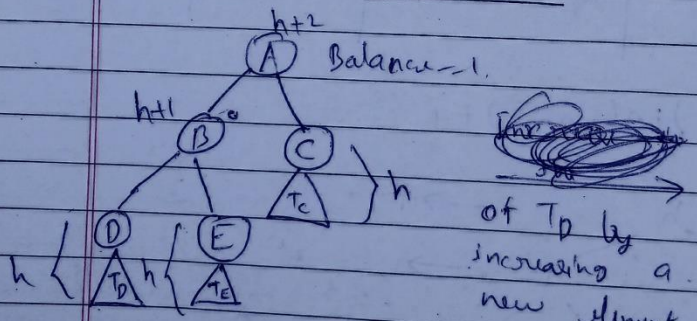
≈ 1.44

$h \leq 1.44 \log_2(n+2)$

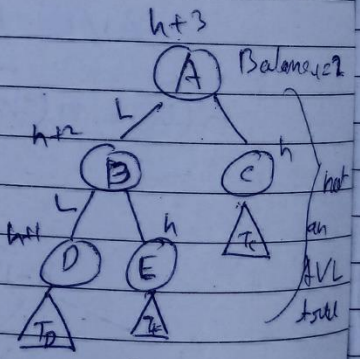
$\theta(h) = \theta(\log n)$.

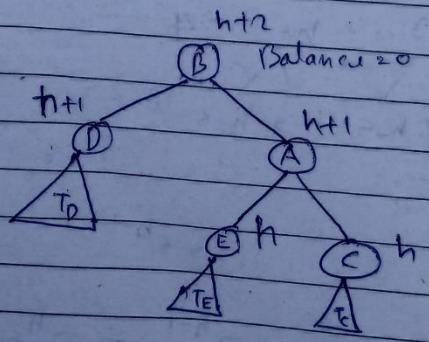
Balancing an AVL Tree -

Single Rotation LL -



of T_D by increasing a new element





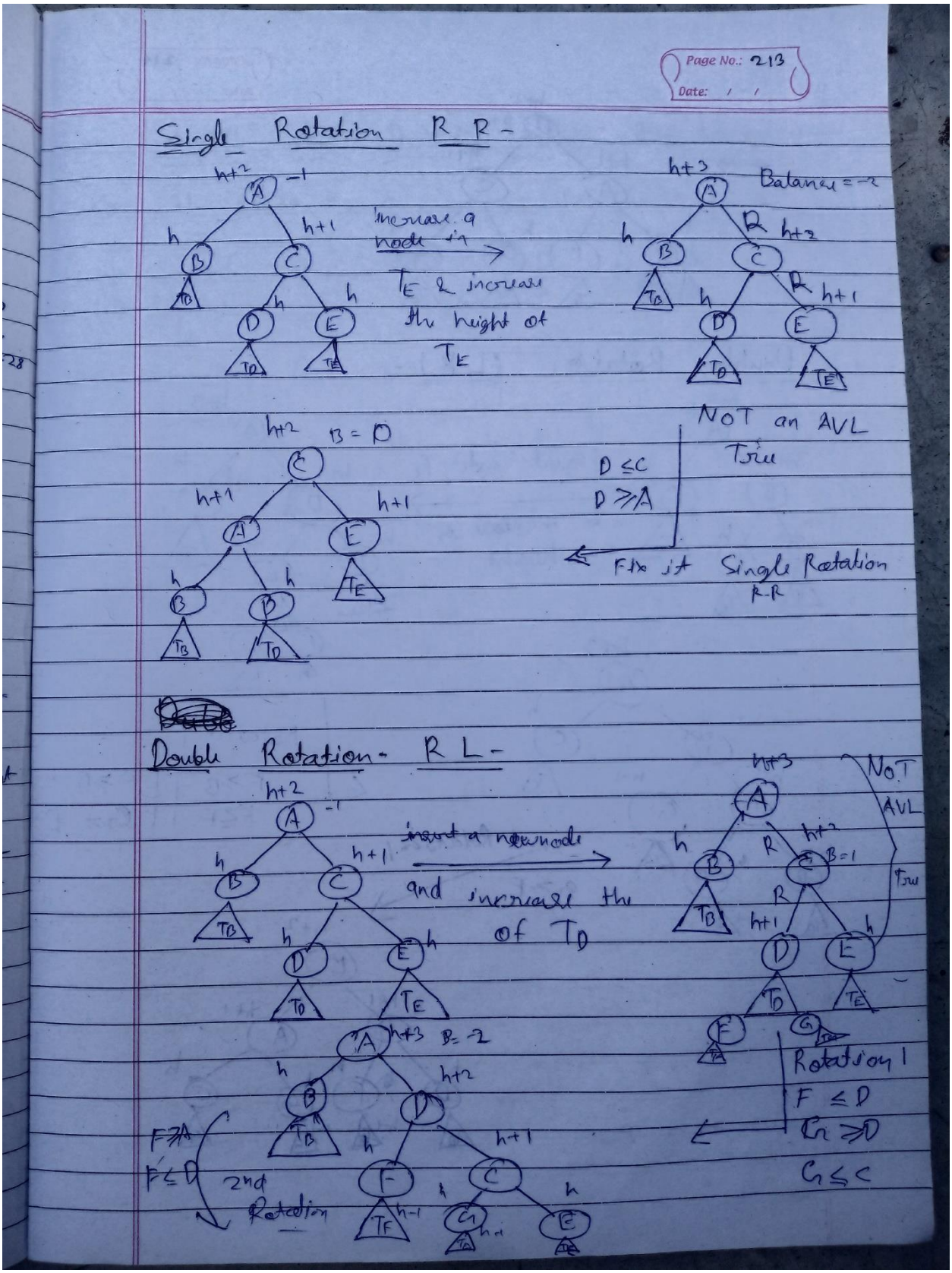
Fix LL Rotation

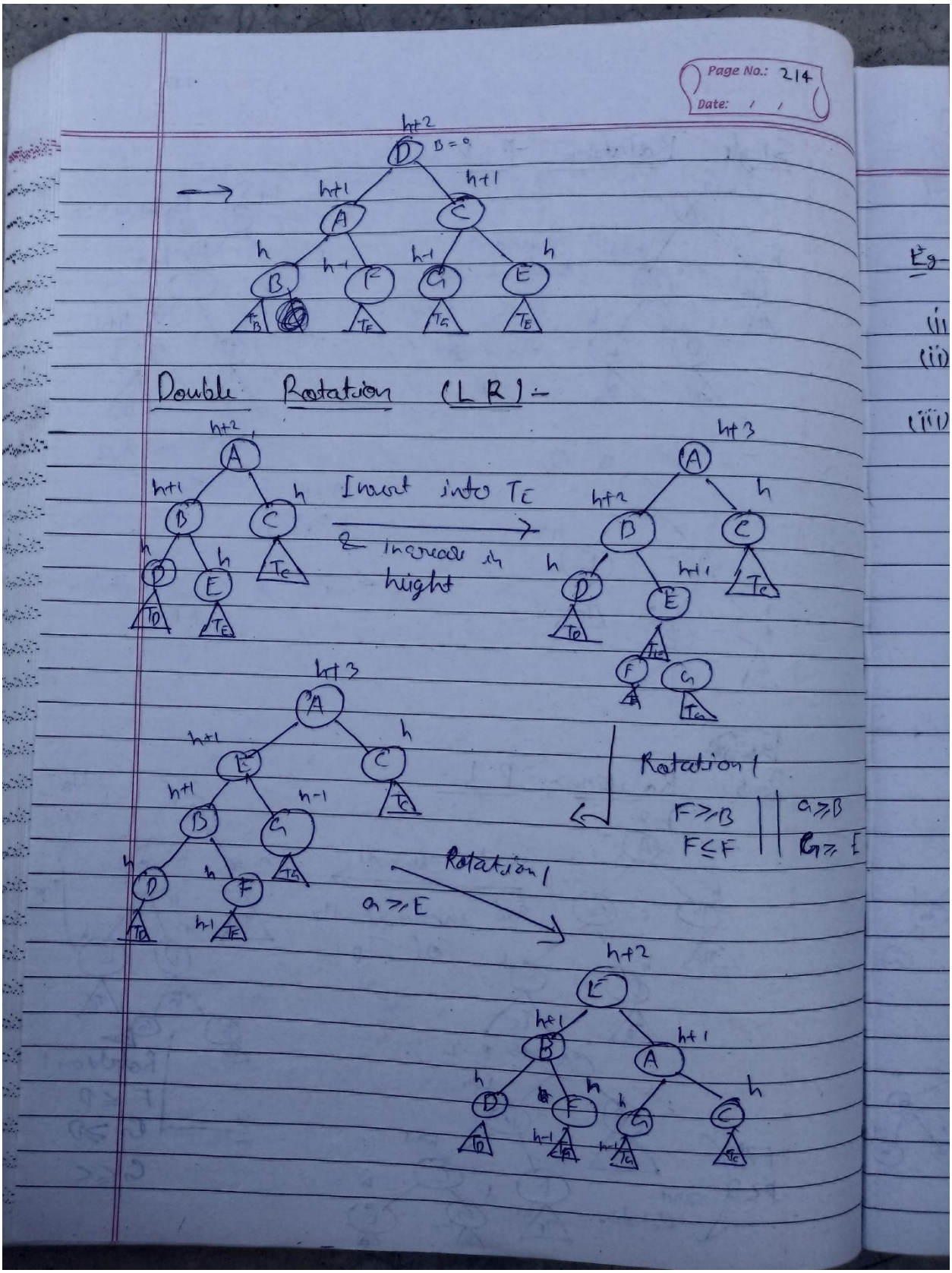
Imbalance

$E \leq A$

$E \geq B$

$A \geq B$


Page No.: 213
Date: / /
Single Rotation R R -
Increase a node in TE & increase the height of TE
Balance = -2
NOT an AVL Tree
D ≤ C
D ≥ A
Fix it Single Rotation R-R
B = D
Double Rotation - R L -
insert a new node and increase the of TD
Rotation 1
F ≤ D
G ≥ D
G ≤ C
2nd Rotation
F ≥ A
F ≤ D
NOT AVL Tree


Page No.: 214
Date: / /
Double Rotation (LR) :-
Insert into T_E
& increase in hight
Rotation 1
$F \geq B$ || $a \geq B$
$F \leq F$ || $G \geq E$
Rotation 1
$a \geq E$

Page No.: 215
Date: / /

AVL Tree :- Build using Insertions -

Eg- 21, 26, 30, 9, 4, 14, 28, 18, 15, 10, 2, 3, 7

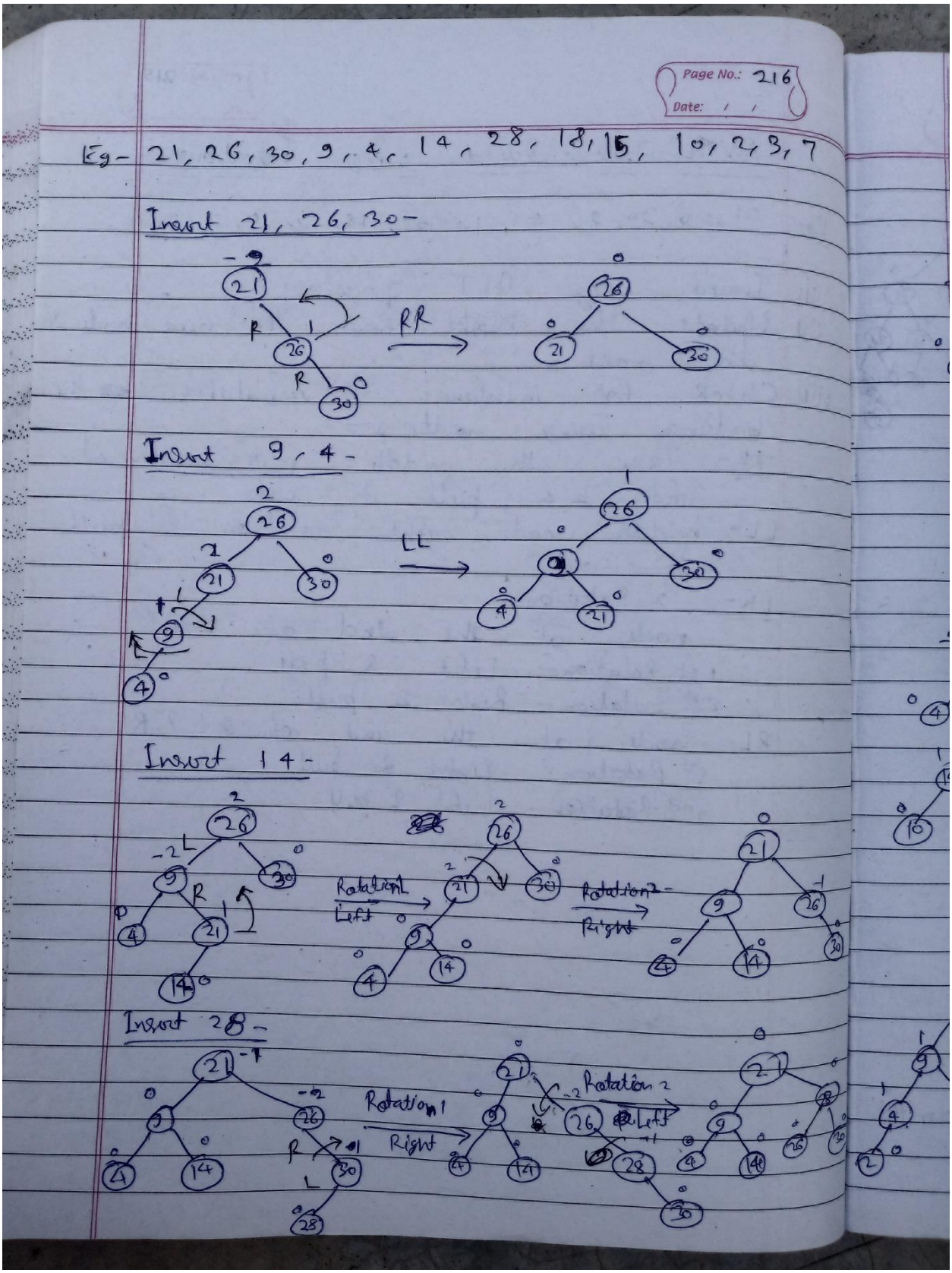
(i) Insert using BST process.
(ii) Update the height from the new node to the root
(iii) Check for imbalance if imbalance the balance using rotations :-

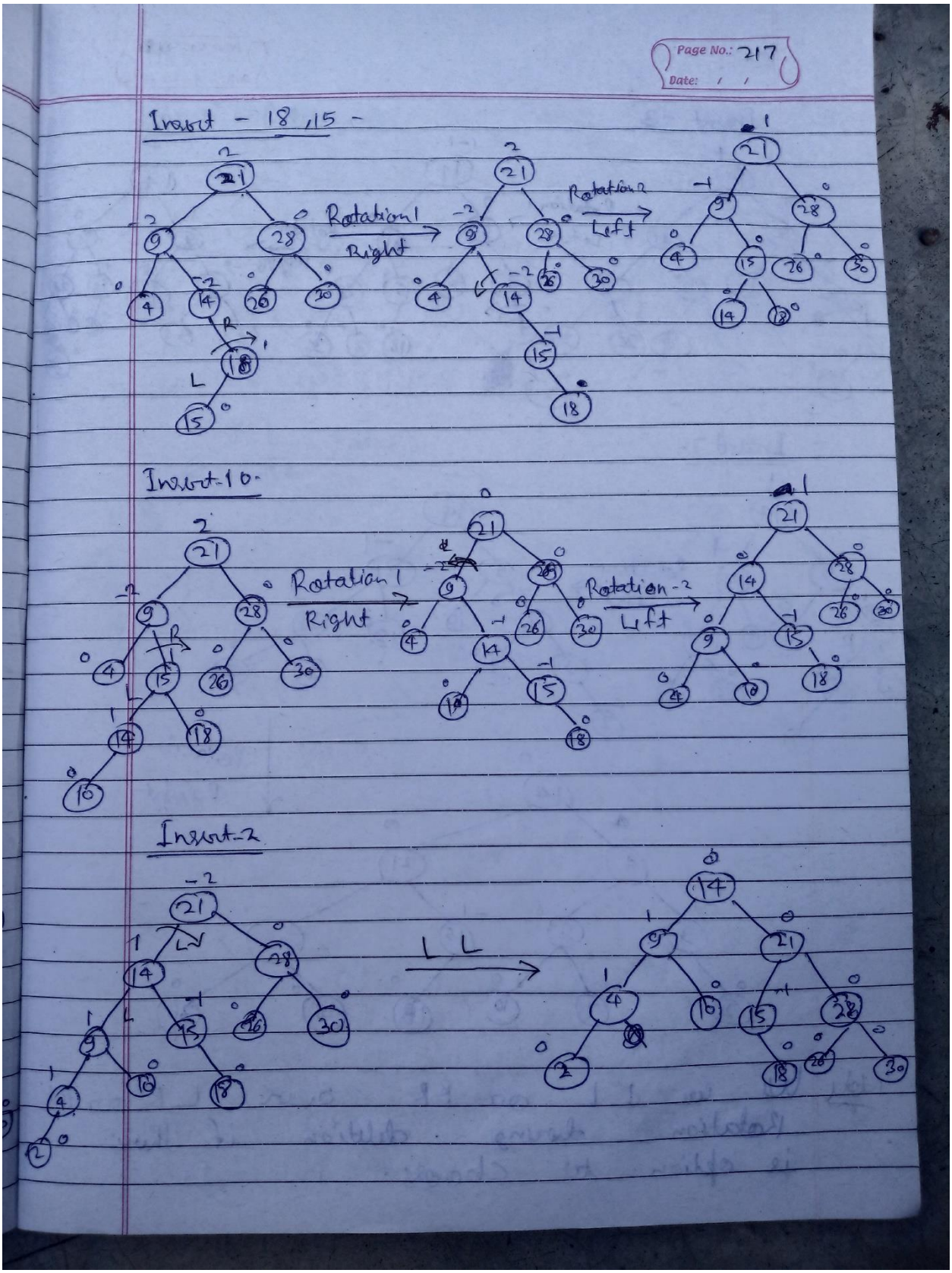
RR- Take the middle node, rotate it left & pull it out.

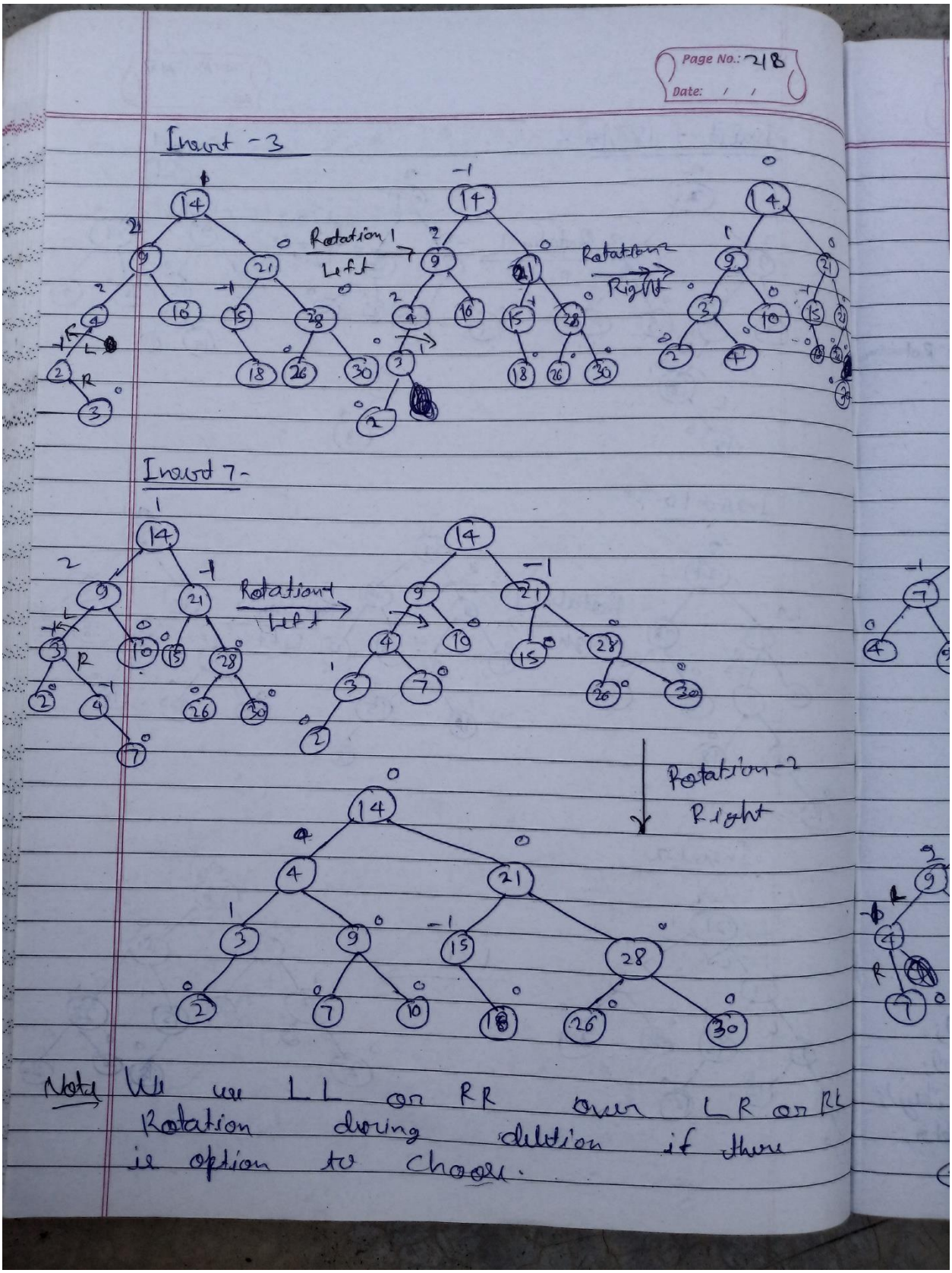
LL- middle node, right rotation & pull it out.

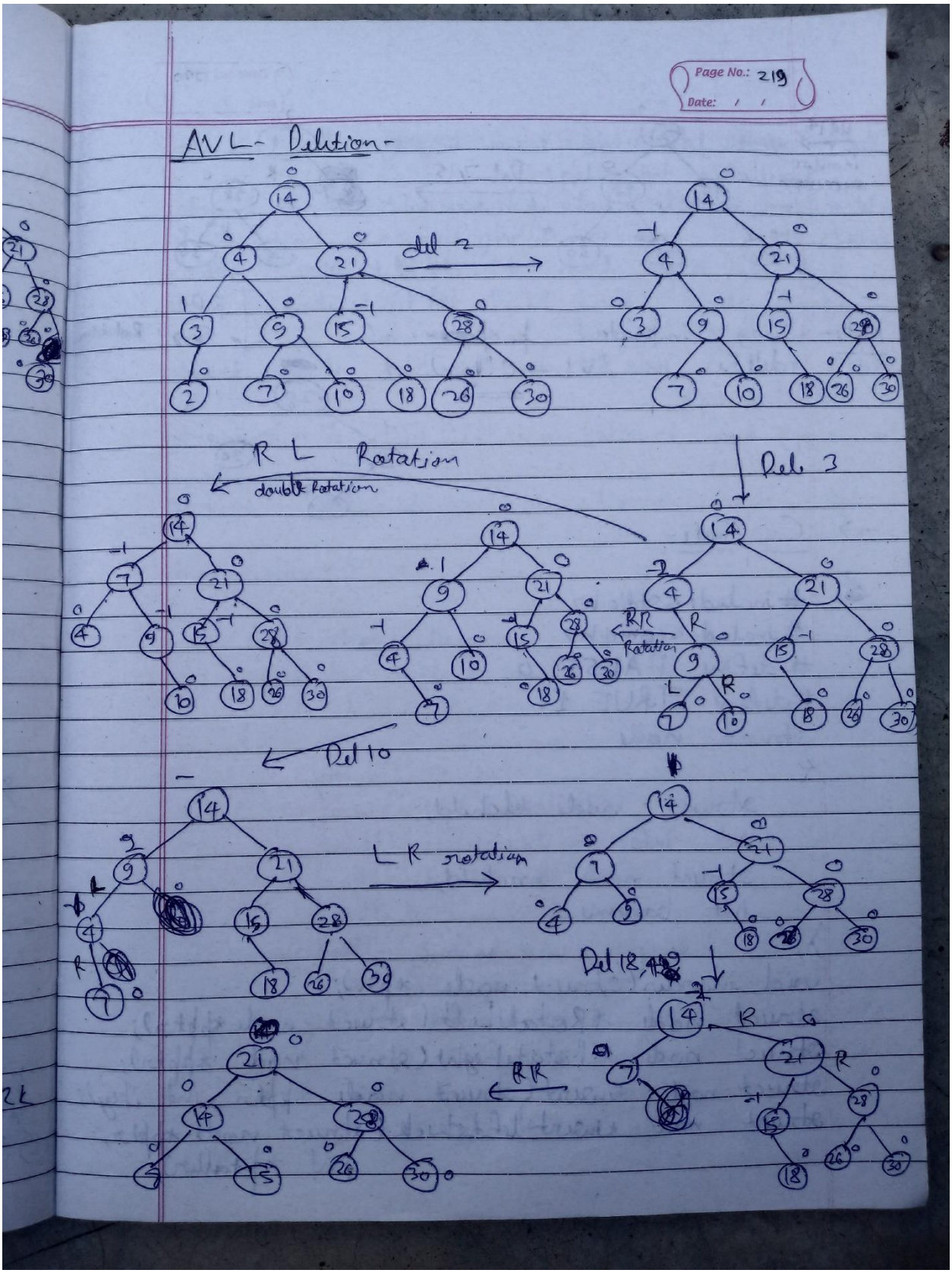
LR- 2 Rotations -
node at the end of L & R
- 1st Rotation - Left & pull
- 2nd rotation - Right & pull.

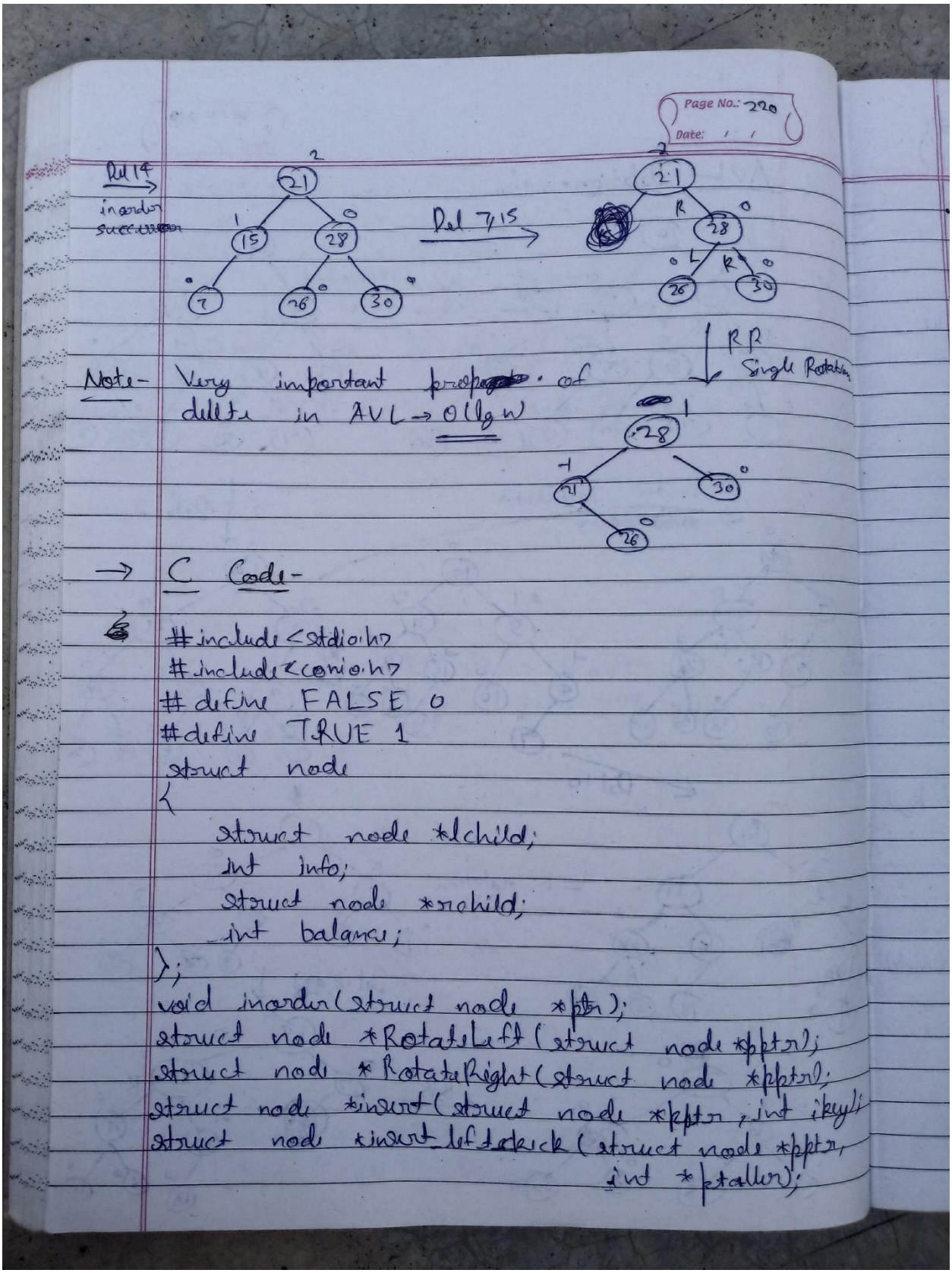
RL- node at the end of L & R
1st Rotation - Right & pull
2nd Rotation - Left & pull


Page No.: 216
Date: / /
Eg- 21, 26, 30, 9, 4, 14, 28, 18, 15, 10, 2, 3, 7
Insert 21, 26, 30-
RR
Insert 9, 4-
LL
Insert 14-
Rotation1 Left
Rotation2 Right
Insert 28-
Rotation1 Right
Rotation 2 Left


Page No.: 217
Date:
Insert - 18, 15 -
Rotation 1
Right
Rotation 2
Left
Insert 10.
Rotation 1
Right
Rotation -2
Left
Insert 2
LL

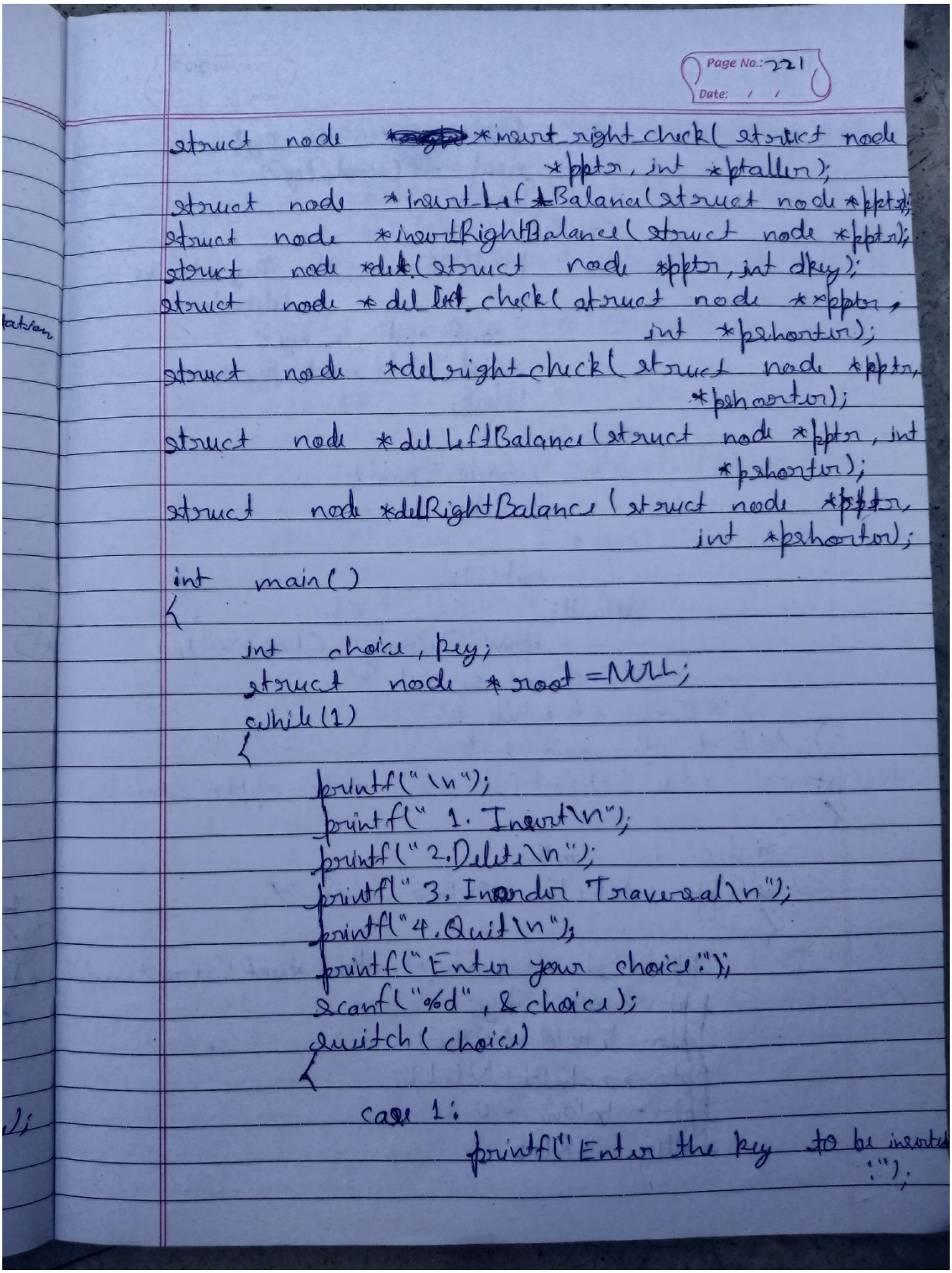

Page No.: 218
Date: / /
Insert -3
Rotation 1
Left
Rotation 2
Right
Insert 7-
Rotation 1
Left
Rotation-2
Right
Note: We use LL or RR over LR or RL Rotation during deletion if there is option to choose.


Page No.: 219
Date: / /
AVL- Deletion-
del 2
RL Rotation
double Rotation
Del. 3
RR Rotation
Del 10
LR rotation
Del 18,
RR

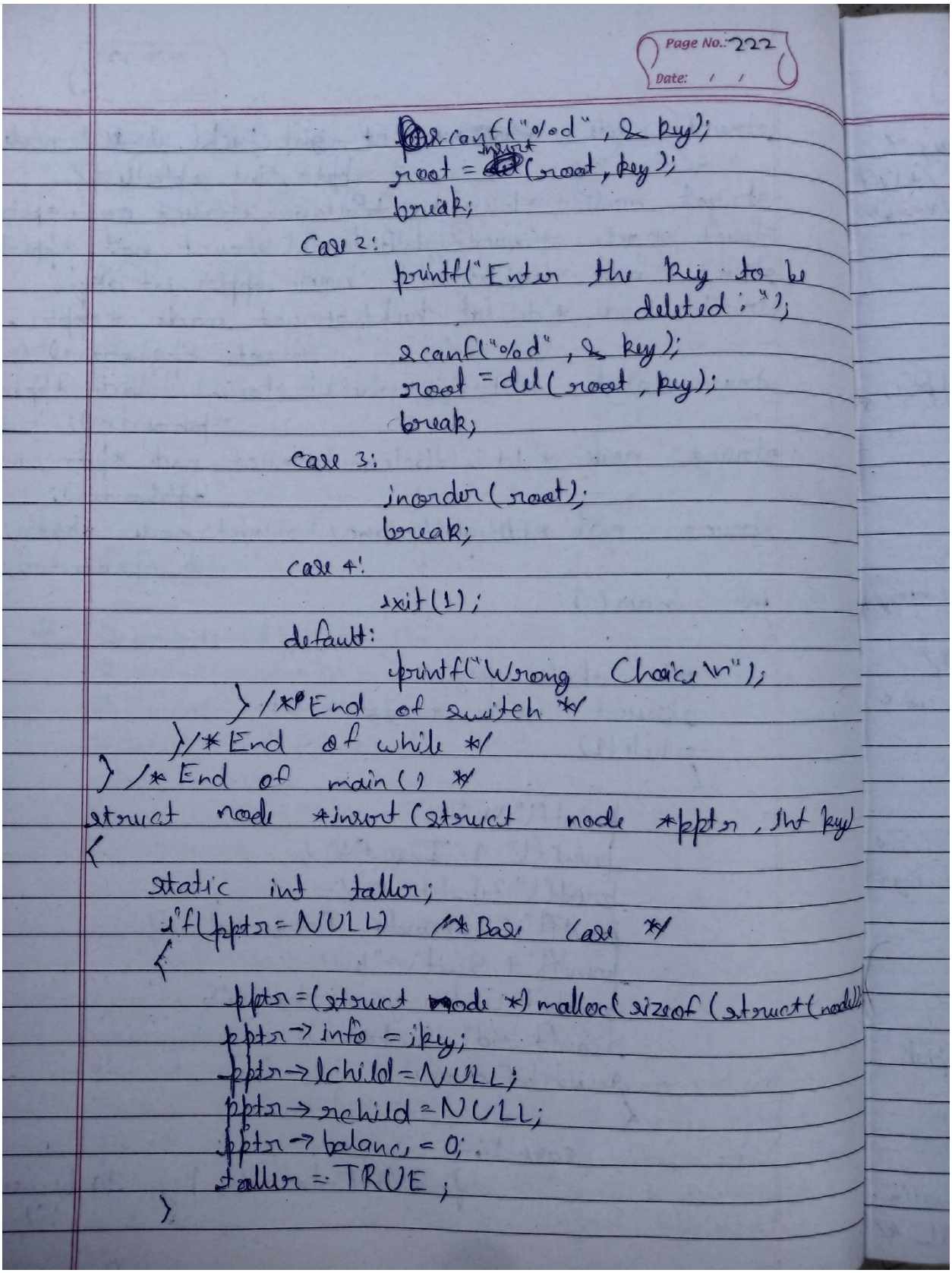


→ C Code-

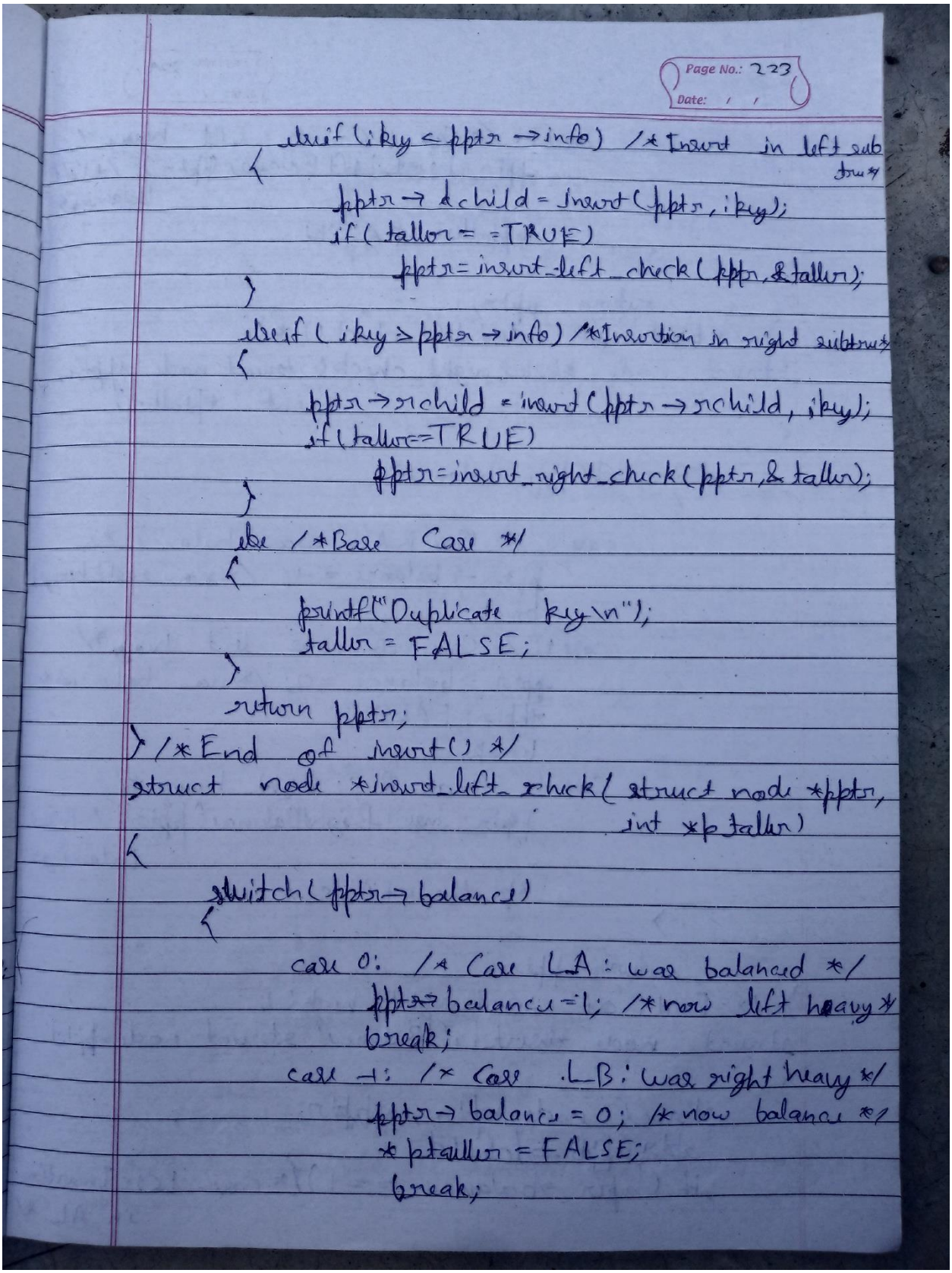
```
#include<stdio.h>
#include<conio.h>
#define FALSE 0
#define TRUE 1
struct node
{
    struct node *lchild;
    int info;
    struct node *rchild;
    int balance;
};
void inorder(struct node *ptr);
struct node *RotateLeft(struct node *pptr);
struct node *RotateRight(struct node *pptr);
struct node *insert(struct node *pptr, int ikey);
struct node *insert_leftcheck(struct node *pptr,
                              int *ptaller);
```



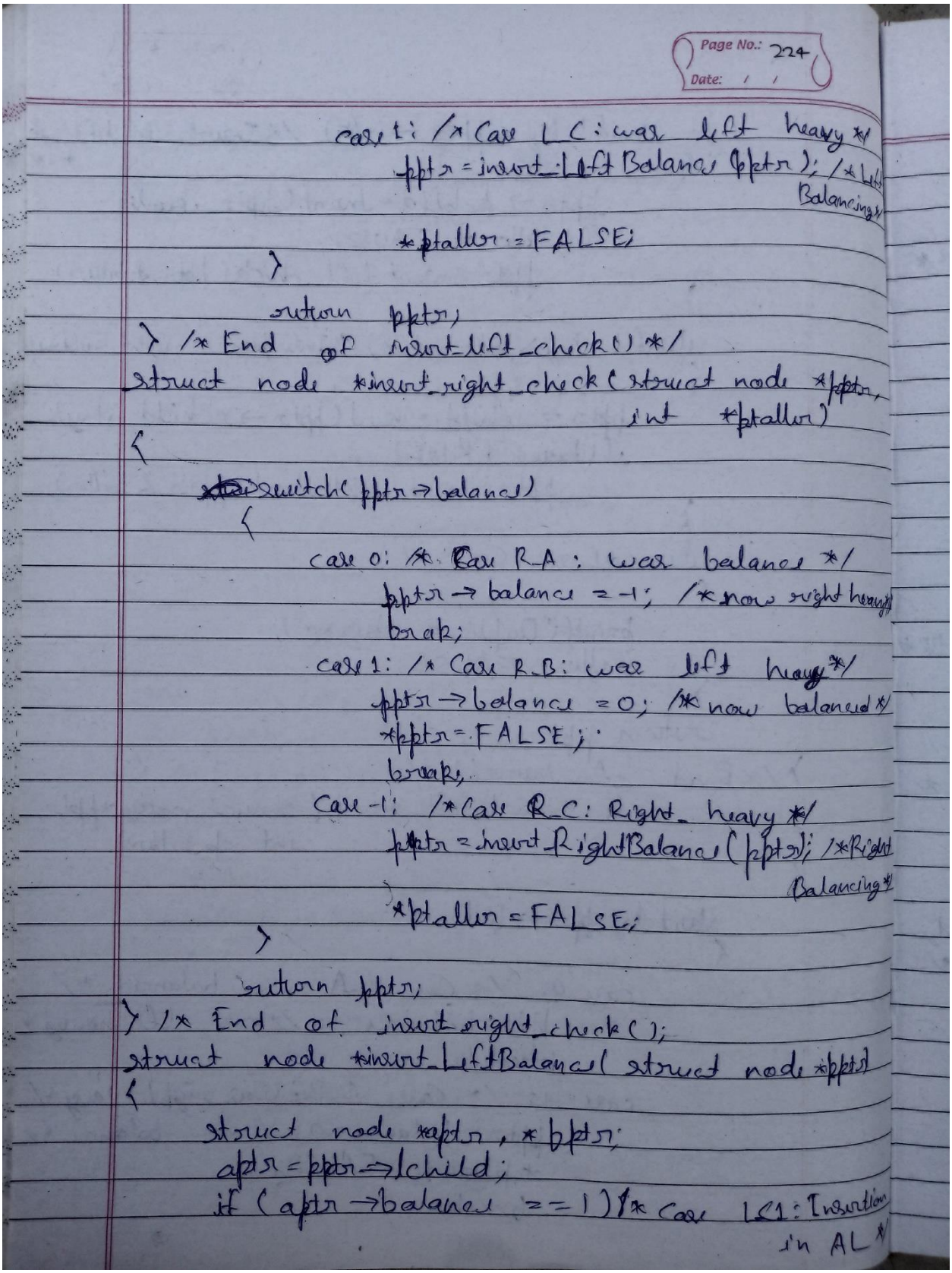
```
struct node *insert_right_check(struct node
                    *pptr, int *ptaller);
struct node *insert_LeftBalance(struct node *pptr);
struct node *insertRightBalance(struct node *pptr);
struct node *del(struct node *pptr, int dkey);
struct node *del_left_check(struct node **pptr,
                    int *pshorter);
struct node *del_right_check(struct node *pptr,
                    *pshorter);
struct node *delLeftBalance(struct node *pptr, int
                    *pshorter);
struct node *delRightBalance(struct node *pptr,
                    int *pshorter);

int main()
{
    int choice, key;
    struct node *root=NULL;
    while(1)
    {
        printf("\n");
        printf(" 1. Insert\n");
        printf("2.Delete\n");
        printf(" 3. Inorder Traversal\n");
        printf("4.Quit\n");
        printf("Enter your choice:");
        scanf("%d", &choice);
        switch(choice)
        {
            case 1:
                printf("Enter the key to be inserted
                    :");
```



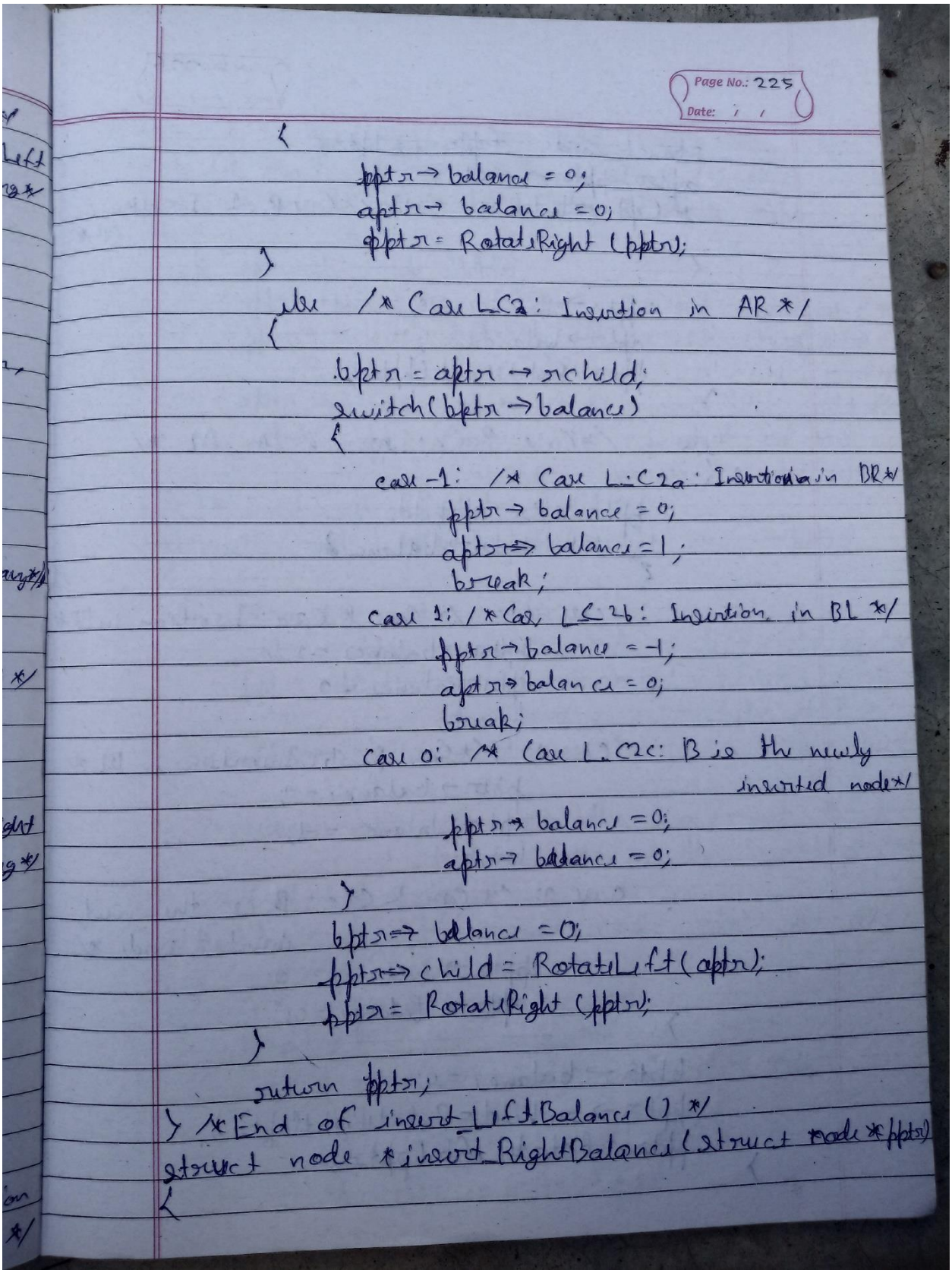
```
                scanf("%d", &key);
                root = insert(root, key);
                break;
        case 2:
                printf("Enter the key to be
                                deleted: ");
                scanf("%d", &key);
                root = del(root, key);
                break;
        case 3:
                inorder(root);
                break;
        case 4:
                exit(1);
        default:
                printf("Wrong Choice\n");
        } /* End of switch */
    } /* End of while */
} /* End of main() */
struct node *insert(struct node *pptr, int key)
{
    static int taller;
    if(pptr==NULL)  /* Base case */
    {
        pptr=(struct node *)malloc(sizeof(struct node));
        pptr->info=ikey;
        pptr->lchild=NULL;
        pptr->rchild=NULL;
        pptr->balance=0;
        taller=TRUE;
    }
```



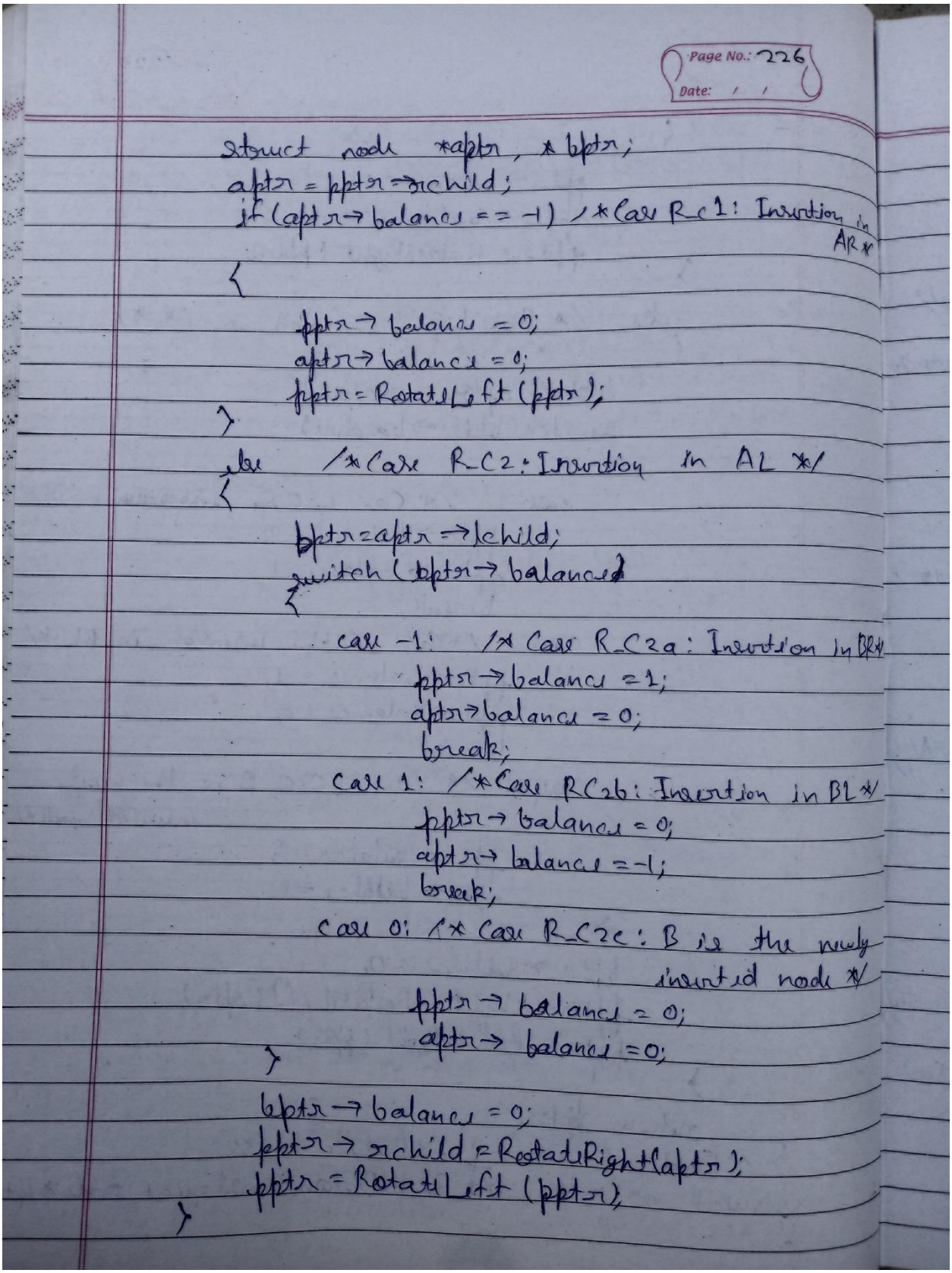
```
    else if (ikey < pptr->info)  /* Insert in left subtree */
    {
        pptr->lchild = insert(pptr->lchild, ikey);
        if(taller == TRUE)
            pptr = insert_left_check(pptr, &taller);
    }
    else if (ikey > pptr->info) /* Insertion in right subtree */
    {
        pptr->rchild = insert(pptr->rchild, ikey);
        if(taller==TRUE)
            pptr = insert_right_check(pptr, &taller);
    }
    else /* Base Case */
    {
        printf("Duplicate key\n");
        taller = FALSE;
    }
    return pptr;
} /* End of insert() */
struct node *insert_left_check(struct node *pptr,
                               int *ptaller)
{
    switch(pptr->balance)
    {
        case 0: /* Case LA: was balanced */
            pptr->balance = 1; /* now left heavy */
            break;
        case 1: /* Case LB: was right heavy */
            pptr->balance = 0; /* now balanced */
            *ptaller = FALSE;
            break;
```



```
Page No.: 224
Date: / /

        case 1: /* Case L_C: was left heavy */
                fptr = insert_LeftBalance (fptr); /* Left
                                                  Balancing */
                *ptaller = FALSE;
        }
        return fptr;
} /* End of insert_left_check() */
struct node *insert_right_check (struct node *fptr,
                                 int *ptaller)
{
    switch(fptr->balance)
    {
        case 0: /* Case R_A : was balance */
                fptr -> balance = -1; /* now right heavy
                break;
        case 1: /* Case R_B: was left heavy */
                fptr -> balance = 0; /* now balanced */
                *fptr = FALSE;
                break;
        case -1: /* Case R_C: Right heavy */
                fptr = insertRightBalance (fptr); /* Right
                                                  Balancing */
                *ptaller = FALSE;
        }
        return fptr;
} /* End of insert_right_check(); */
struct node *insert_LeftBalance (struct node *fptr)
{
    struct node *aptr, *bptr;
    aptr = fptr->lchild;
    if (aptr->balance == 1) /* Case LC1: Insertion
                               in AL */
```

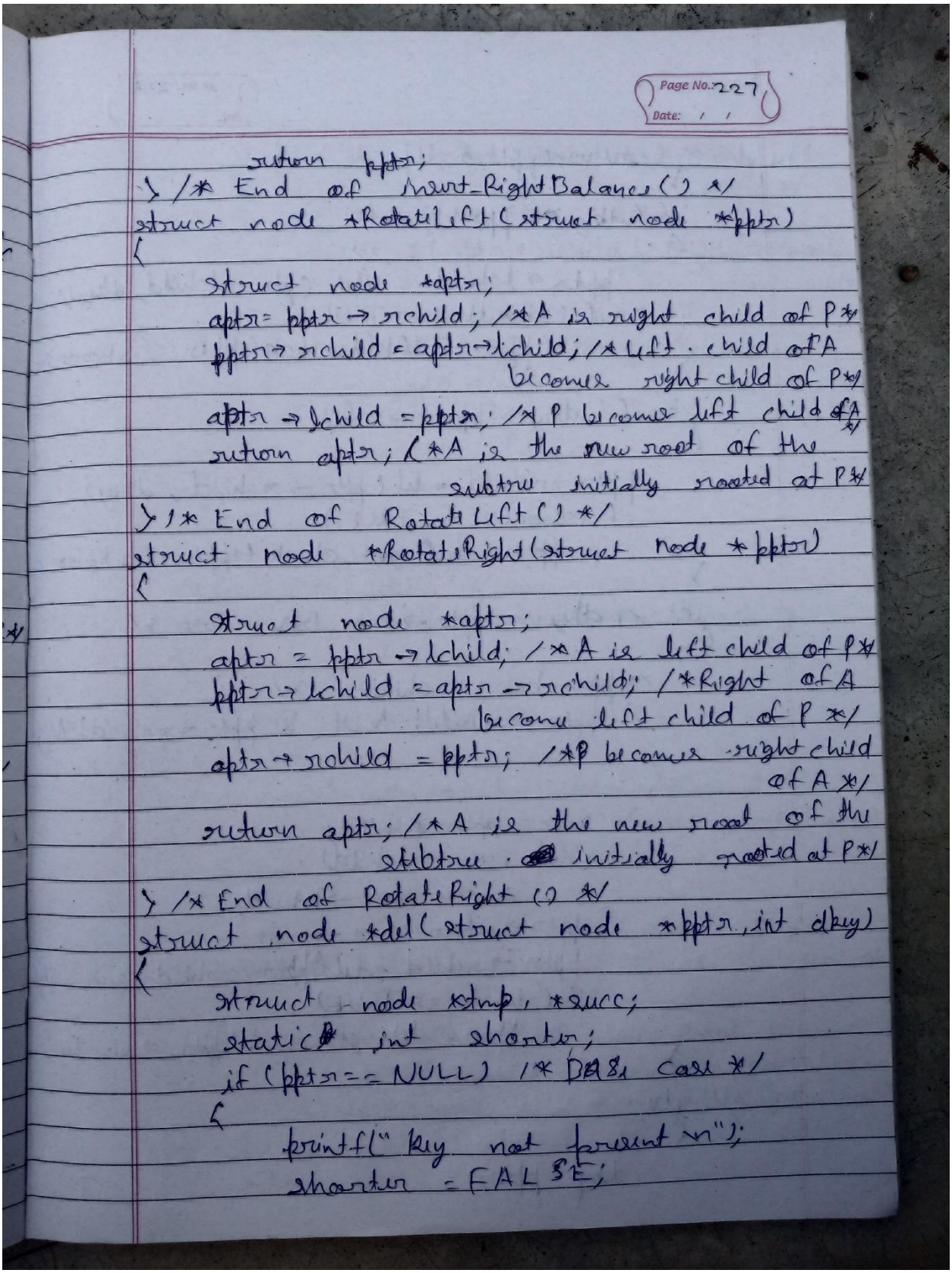


```
        {
            fptr -> balance = 0;
            aptr -> balance = 0;
            fptr = RotateRight (fptr);
        }
        else /* Case L-C2 : Insertion in AR */
        {
            bptr = aptr -> rchild;
            switch(bptr -> balance)
            {
                case -1: /* Case L-C2a : Insertion in BR */
                    fptr -> balance = 0;
                    aptr -> balance = 1;
                    break;
                case 1: /* Case L-C2b : Insertion in BL */
                    fptr -> balance = -1;
                    aptr -> balance = 0;
                    break;
                case 0: /* Case L-C2c : B is the newly
                                        inserted node */
                    fptr -> balance = 0;
                    aptr -> balance = 0;
            }
            bptr -> balance = 0;
            fptr -> child = RotateLeft (aptr);
            fptr = RotateRight (fptr);
        }
        return fptr;
    } /* End of insert_LeftBalance () */
    struct node *insert_RightBalance (struct node *fptr)
    {
```

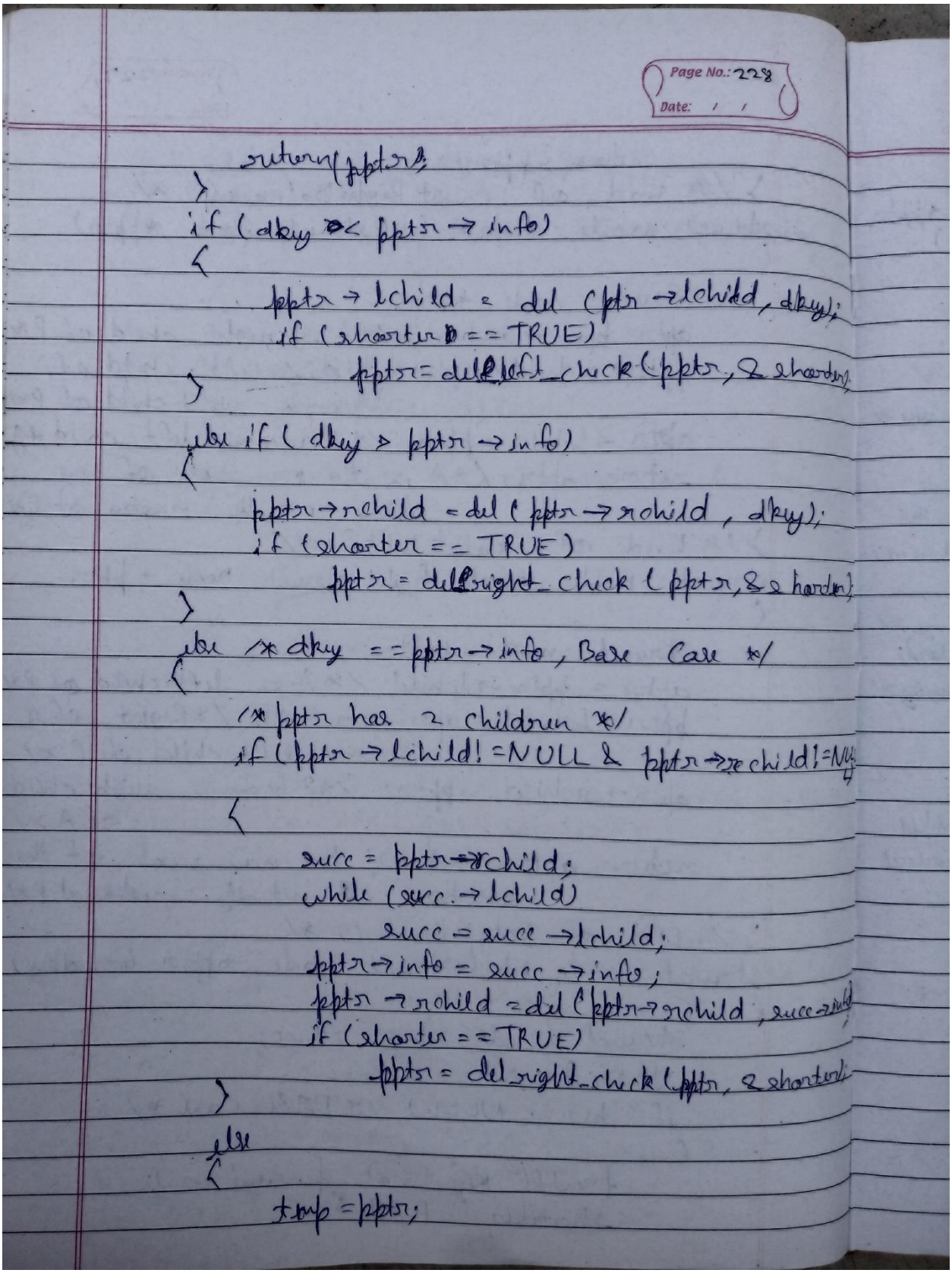


Page No.: 226
Date: / /

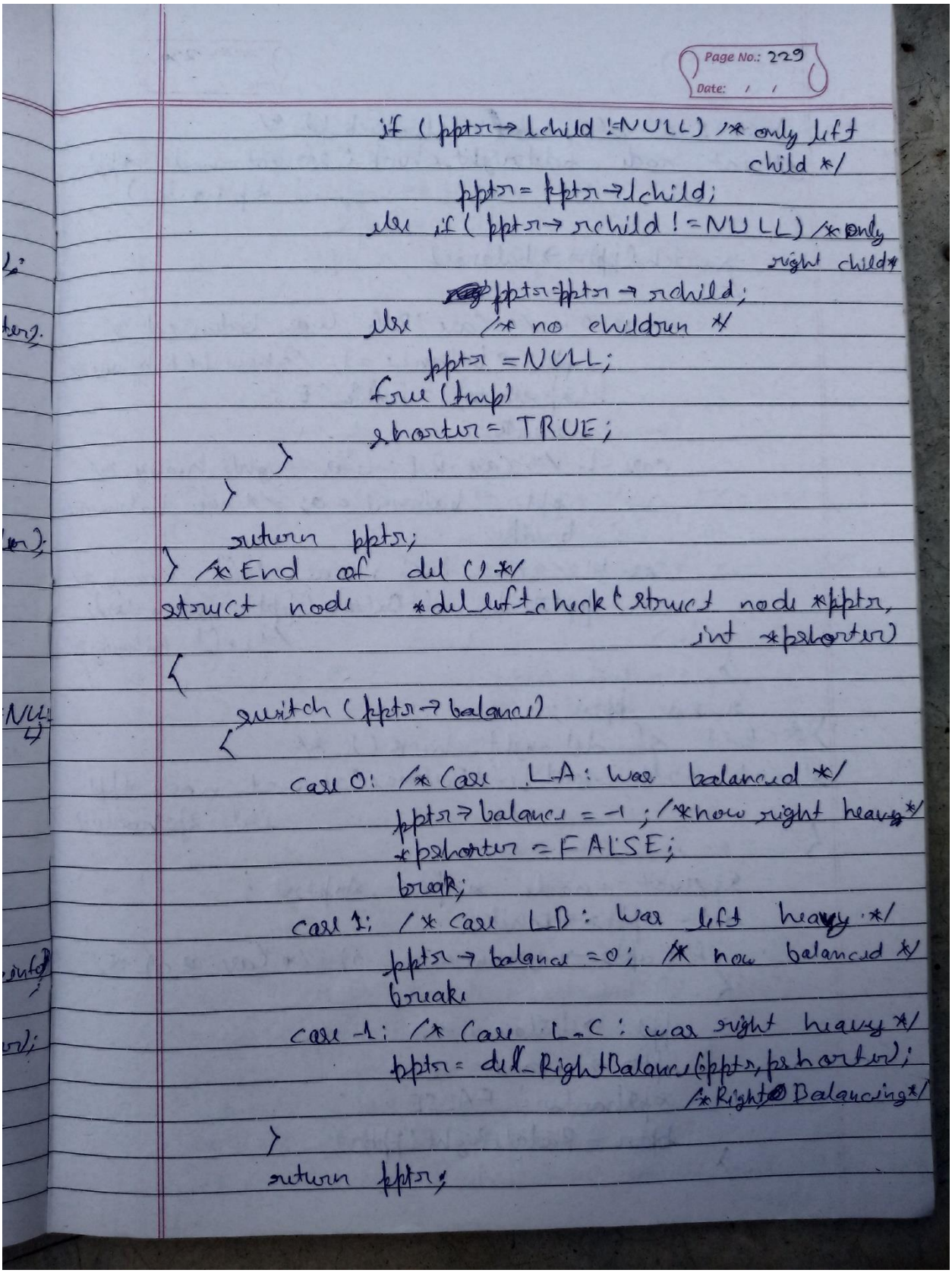
```
struct node *aptr, *bptr;
aptr = pptr->rchild;
if(aptr->balance == -1) /* Case R_C1: Insertion in AR */
{
    pptr->balance = 0;
    aptr->balance = 0;
    pptr = RotateLeft(pptr);
}
else /* Case R_C2: Insertion in AL */
{
    bptr = aptr->lchild;
    switch(bptr->balance)
    {
        case -1: /* Case R_C2a: Insertion in BR */
            pptr->balance = 1;
            aptr->balance = 0;
            break;
        case 1: /* Case R_C2b: Insertion in BL */
            pptr->balance = 0;
            aptr->balance = -1;
            break;
        case 0: /* Case R_C2c: B is the newly
                   inserted node */
            pptr->balance = 0;
            aptr->balance = 0;
    }
    bptr->balance = 0;
    pptr->rchild = RotateRight(aptr);
    pptr = RotateLeft(pptr);
}
```



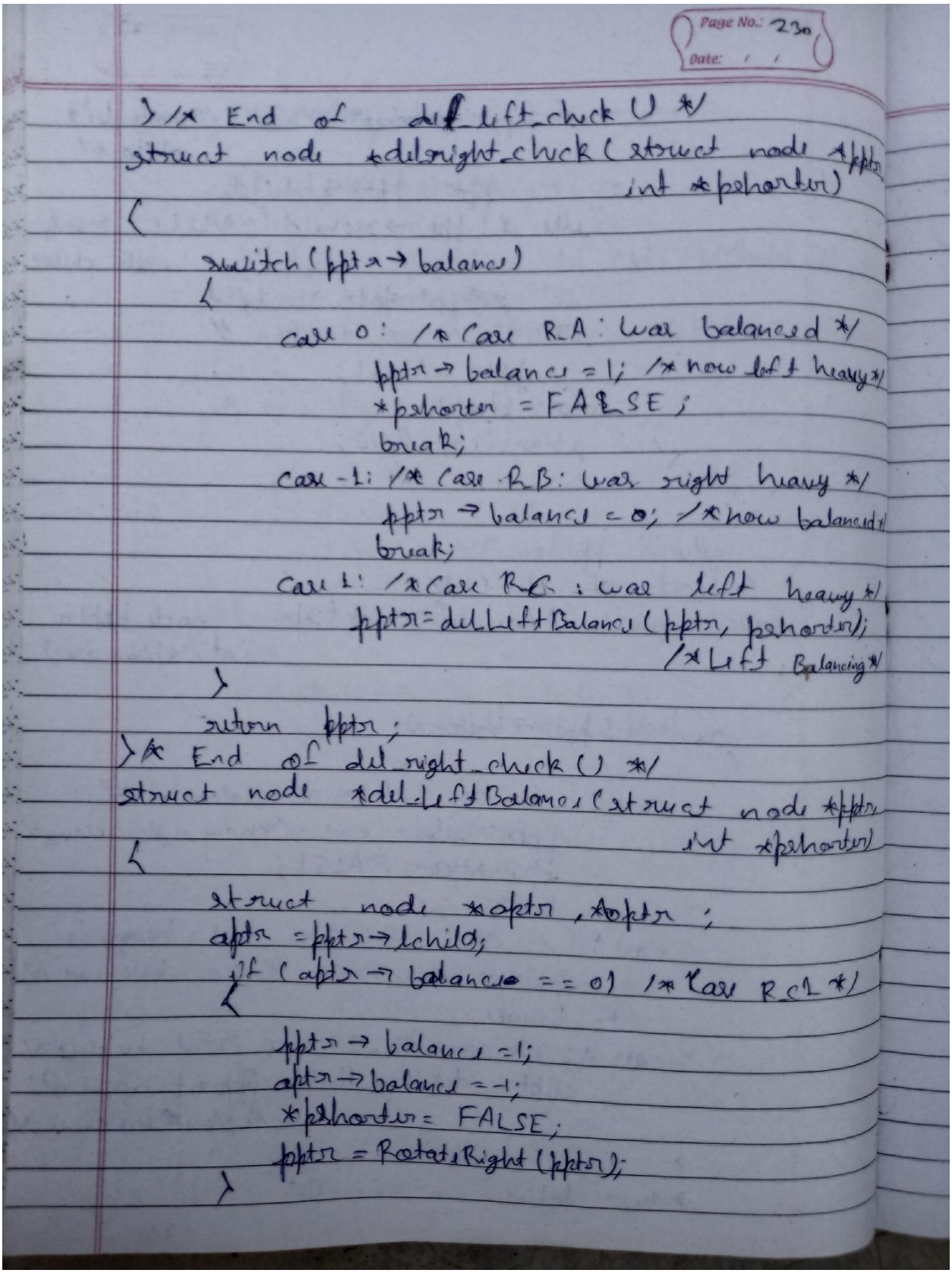
```
        return pptr;
} /* End of Insert_RightBalance() */
struct node *RotateLeft(struct node *pptr)
{
    struct node *aptr;
    aptr = pptr -> rchild; /*A is right child of P*/
    pptr -> rchild = aptr -> lchild; /*Left child of A
                                becomes right child of P*/
    aptr -> lchild = pptr; /*P becomes left child of A*/
    return aptr; /*A is the new root of the
                    subtree initially rooted at P*/
} /* End of RotateLeft() */
struct node *RotateRight(struct node *pptr)
{
    struct node *aptr;
    aptr = pptr -> lchild; /*A is left child of P*/
    pptr -> lchild = aptr -> rchild; /*Right of A
                                becomes left child of P */
    aptr -> rchild = pptr; /*P becomes right child
                                of A */
    return aptr; /*A is the new root of the
                    subtree initially rooted at P*/
} /* End of RotateRight() */
struct node *del(struct node *pptr, int dkey)
{
    struct node *tmp, *succ;
    static int shorter;
    if (pptr == NULL) /* Base case */
    {
        printf("Key not present\n");
        shorter = FALSE;
```



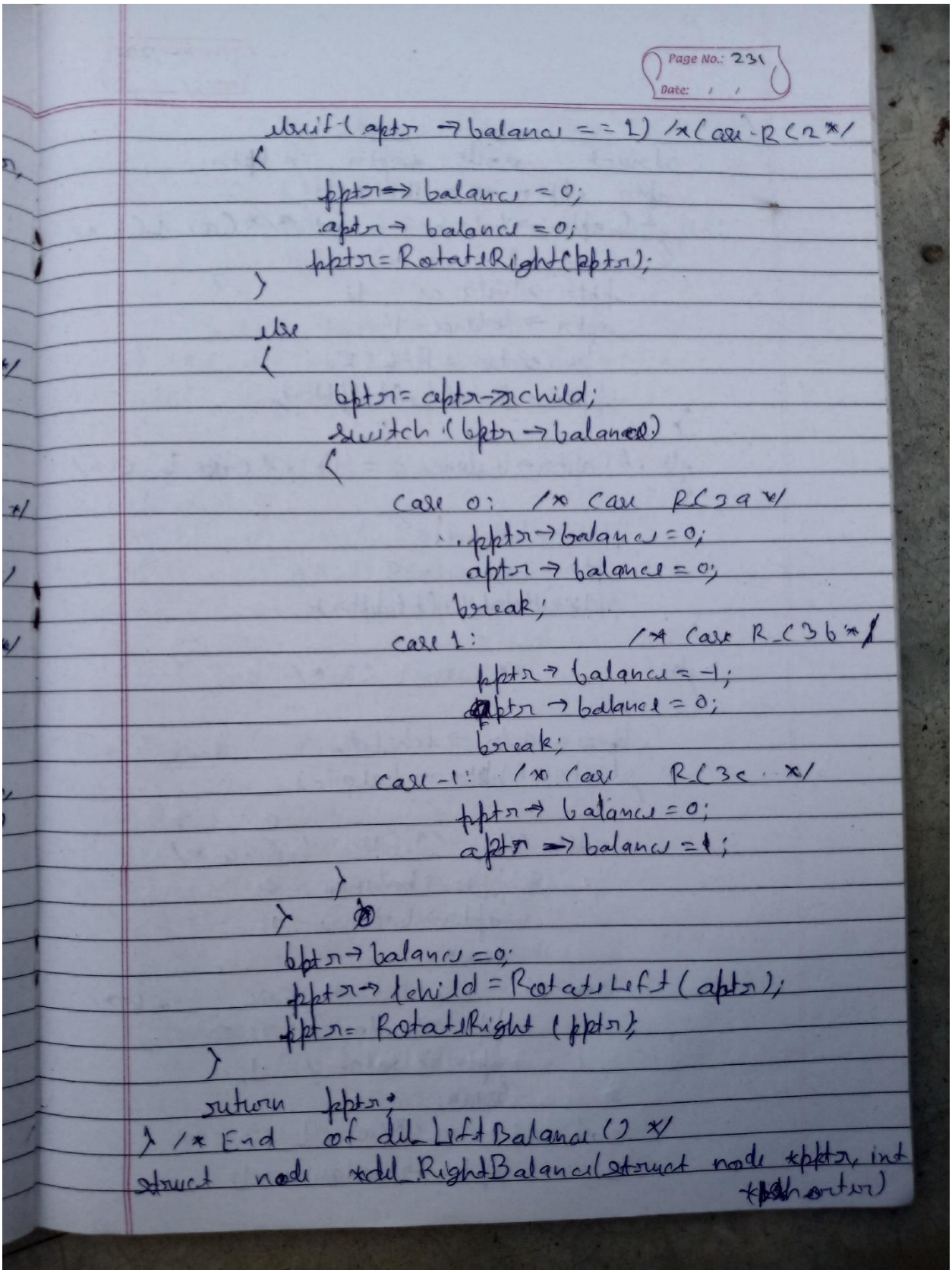
```
        return(pptr);
    }
    if (dkey < pptr->info)
    {
        pptr->lchild = del(pptr->lchild, dkey);
        if (shorter == TRUE)
            pptr = del_left_check(pptr, &shorter);
    }
    else if (dkey > pptr->info)
    {
        pptr->rchild = del(pptr->rchild, dkey);
        if (shorter == TRUE)
            pptr = del_right_check(pptr, &shorter);
    }
    else /* dkey == pptr->info, Base Case */
    {
        /* pptr has 2 children */
        if (pptr->lchild != NULL && pptr->rchild != NULL)
        {
            succ = pptr->rchild;
            while (succ->lchild)
                succ = succ->lchild;
            pptr->info = succ->info;
            pptr->rchild = del(pptr->rchild, succ->info);
            if (shorter == TRUE)
                pptr = del_right_check(pptr, &shorter);
        }
        else
        {
            temp = pptr;
```



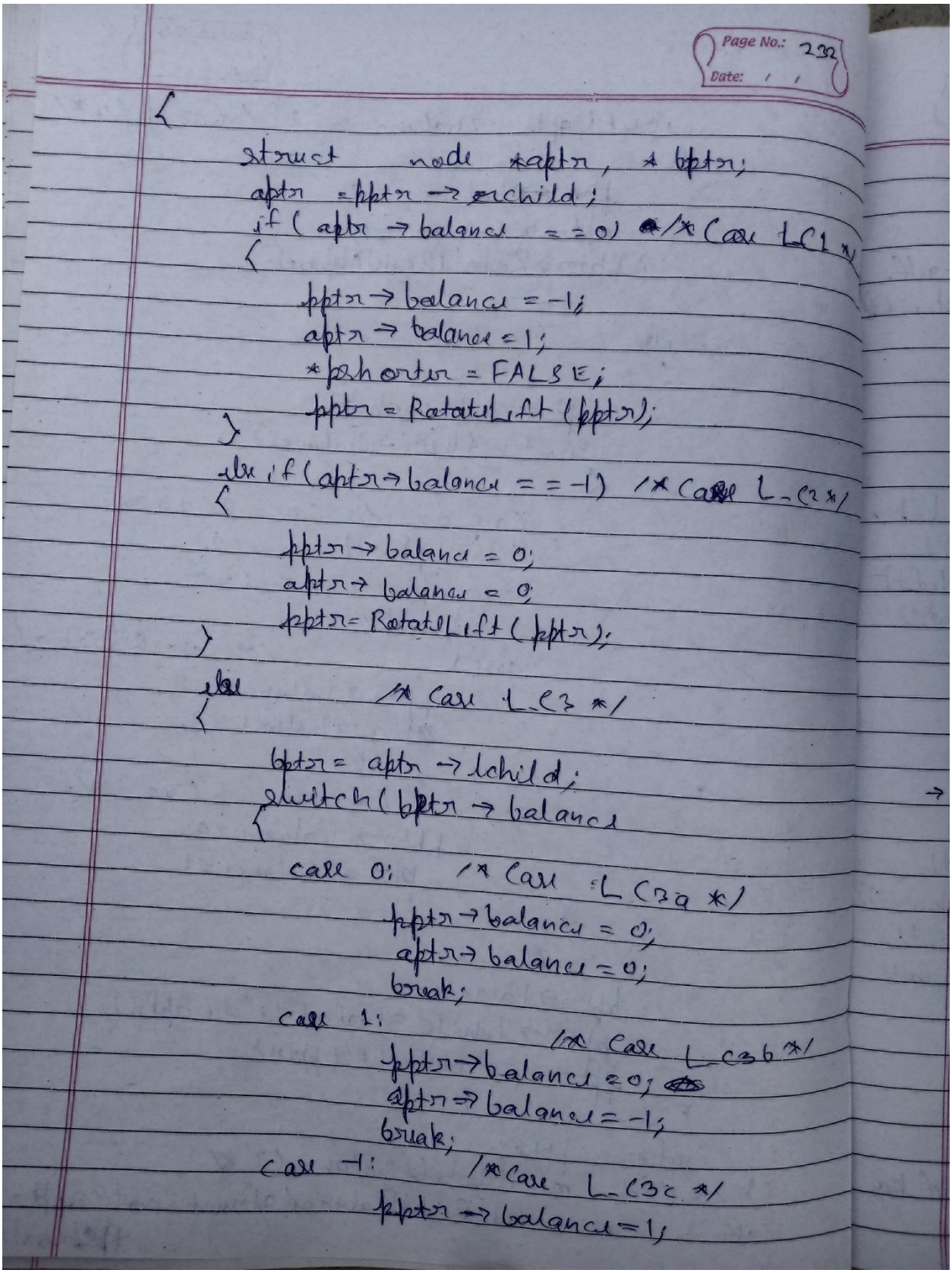
```
Page No.: 229
Date: / /

if (pptr->lchild != NULL) /* only left
                              child */
    pptr = pptr->lchild;
else if (pptr->rchild != NULL) /* only
                              right child */
    pptr = pptr->rchild;
else    /* no children */
    pptr = NULL;
free (tmp)
*shorter = TRUE;
}
}
return pptr;
} /* End of del() */
struct node *del_left_check(struct node *pptr,
                            int *pshorter)
{
switch (pptr->balance)
{
case 0: /* Case L_A: Was balanced */
    pptr->balance = -1; /* now right heavy */
    *pshorter = FALSE;
    break;
case 1: /* Case L_B: Was left heavy */
    pptr->balance = 0; /* now balanced */
    break;
case -1: /* Case L_C: was right heavy */
    pptr = del_RightBalance(pptr, pshorter);
                        /* Right Balancing */
}
return pptr;
```



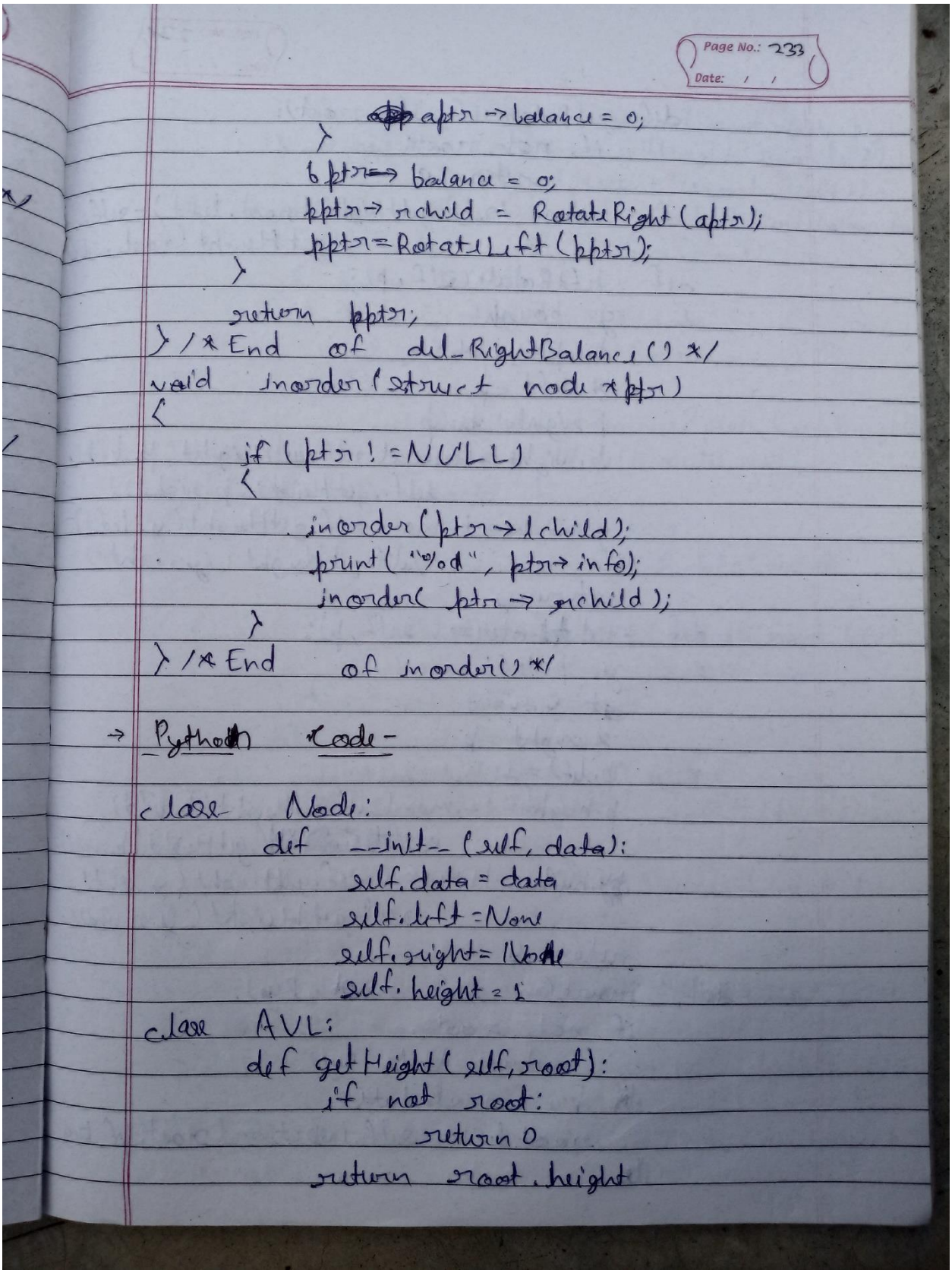
```
Page No.: 230
Date:  /  /

}/* End of del_left_chuck () */
struct node *delright_chuck (struct node *pptr,
                             int *pshorter)
{
    switch (pptr->balance)
    {
        case 0: /* case R_A: was balanced */
                pptr->balance = 1; /* now left heavy */
                *pshorter = FALSE;
                break;
        case -1: /* case R_B: was right heavy */
                pptr->balance = 0; /* now balanced */
                break;
        case 1: /* case R_C: was left heavy */
                pptr = delLeftBalance (pptr, pshorter);
                                /* Left Balancing */
    }
    return pptr;
}/* End of del_right_chuck () */
struct node *delLeftBalance (struct node *pptr,
                             int *pshorter)
{
    struct node *aptr, *bptr;
    aptr = pptr->lchild;
    if (aptr->balance == 0) /* case R_C1 */
    {
        pptr->balance = 1;
        aptr->balance = -1;
        *pshorter = FALSE;
        pptr = RotateRight (pptr);
    }
```



```
Page No.: 231
Date:

if (aptr -> balance == 2) /* case RC2 */
{
    fptr -> balance = 0;
    aptr -> balance = 0;
    fptr = RotateRight(fptr);
}
else
{
    bptr = aptr -> rchild;
    switch (bptr -> balance)
    {
        case 0: /* case RC2a */
            fptr -> balance = 0;
            aptr -> balance = 0;
            break;
        case 1:                 /* case RC3b */
            fptr -> balance = -1;
            aptr -> balance = 0;
            break;
        case -1: /* case RC3c */
            fptr -> balance = 0;
            aptr -> balance = 1;
    }
    bptr -> balance = 0;
    fptr -> rchild = RotateLeft(aptr);
    fptr = RotateRight(fptr);
}
return fptr;
} /* End of dd_LeftBalance() */
struct node *dd_RightBalance(struct node *fptr, int *shorter)
```



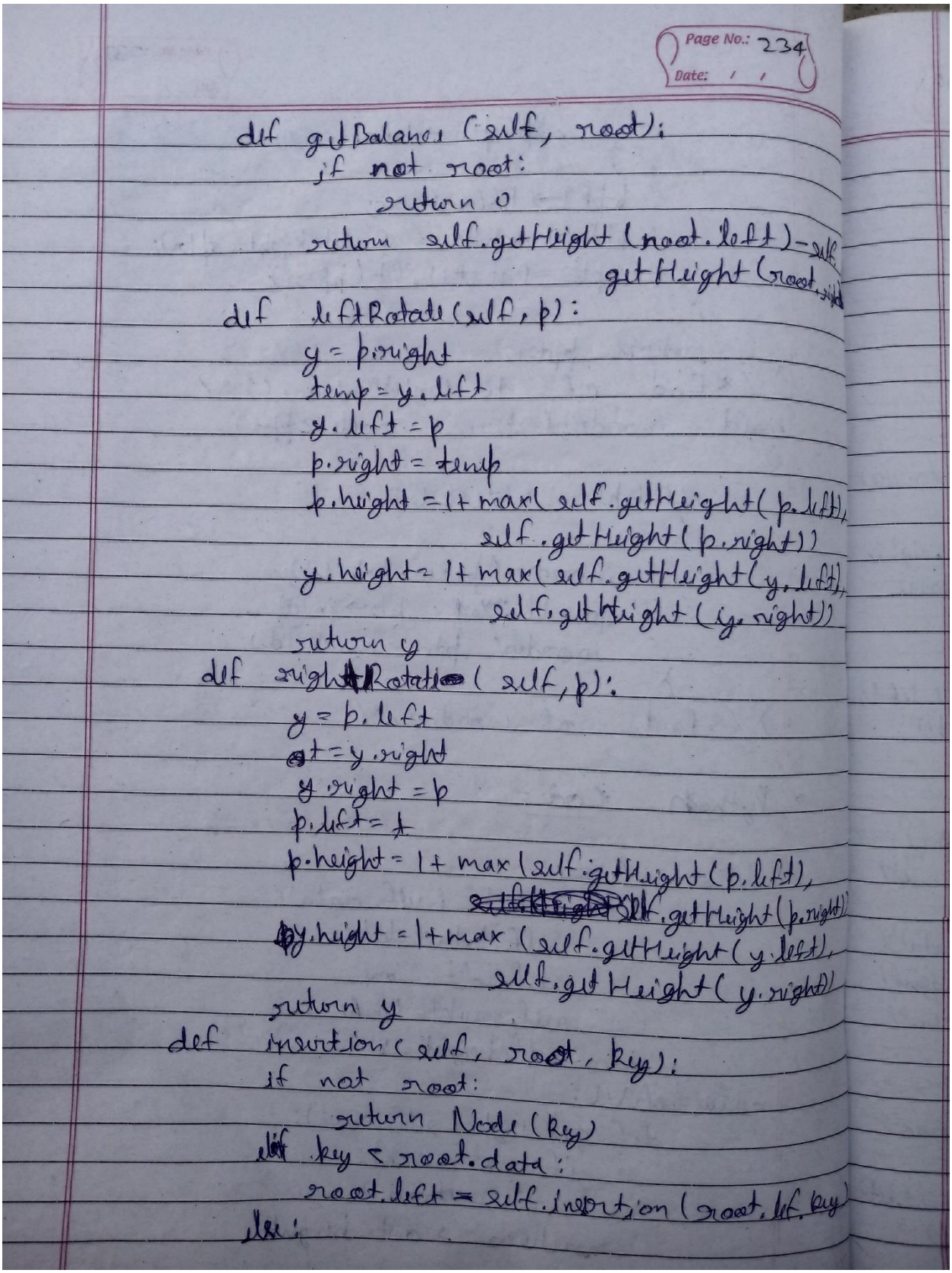
```
Page No.: 232
Date: / /

{
    struct node *aptr, *bptr;
    aptr = pptr -> rchild;
    if ( aptr -> balance == 0) /* Case LC1 */
    {
        pptr -> balance = -1;
        aptr -> balance = 1;
        *shorter = FALSE;
        pptr = RotateLeft (pptr);
    }
    else if (aptr -> balance == -1) /* Case L_C2 */
    {
        pptr -> balance = 0;
        aptr -> balance = 0;
        pptr = RotateLeft (pptr);
    }
    else                    /* Case L_C3 */
    {
        bptr = aptr -> lchild;
        switch ( bptr -> balance
        {
            case 0:     /* Case L_C3a */
                pptr -> balance = 0;
                aptr -> balance = 0;
                break;
            case 1:                 /* Case L_C3b */
                pptr -> balance = 0;
                aptr -> balance = -1;
                break;
            case -1:    /* Case L_C3c */
                pptr -> balance = 1;
```



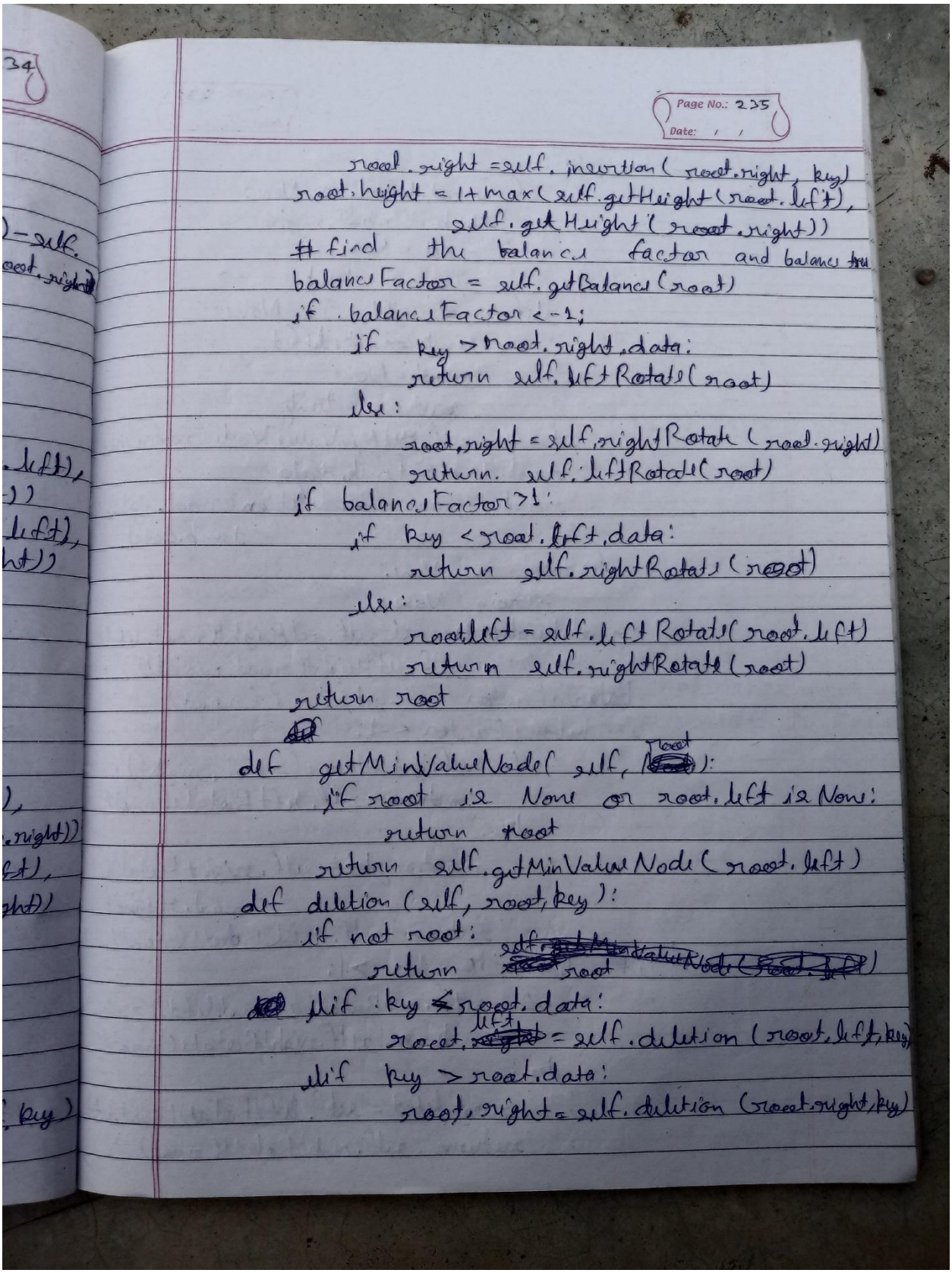
```
        aptr -> balance = 0;
    }
    bptr -> balance = 0;
    pptr -> rchild = RotateRight(aptr);
    pptr = RotateLeft(pptr);
}
return pptr;
} /* End of dbl_RightBalance() */
void inorder(struct node *ptr)
{
    if (ptr != NULL)
    {
        inorder(ptr -> lchild);
        printf("%d", ptr -> info);
        inorder(ptr -> rchild);
    }
} /* End of inorder() */
```

→ Python Code -

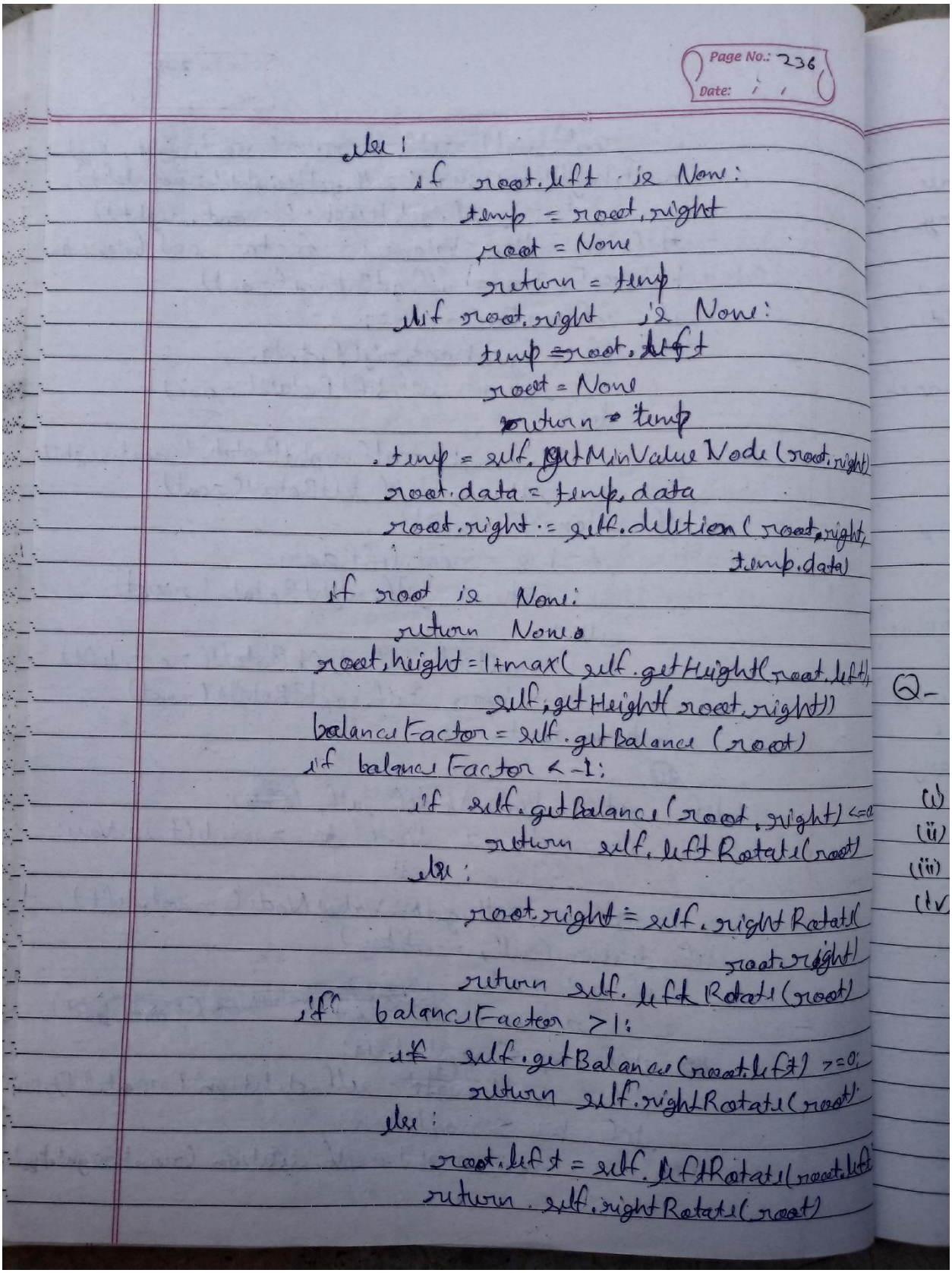
```
class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = Node
        self.height = 1
class AVL:
    def getHeight(self, root):
        if not root:
            return 0
        return root.height
```



```
Page No.: 234
Date: / /

def getBalance(self, root):
    if not root:
        return 0
    return self.getHeight(root.left) - self.
                                getHeight(root.
def leftRotate(self, p):
    y = p.right
    temp = y.left
    y.left = p
    p.right = temp
    p.height = 1 + max(self.getHeight(p.left),
                       self.getHeight(p.right))
    y.height = 1 + max(self.getHeight(y.left),
                       self.getHeight(y.right))
    return y
def rightRotate(self, p):
    y = p.left
    t = y.right
    y.right = p
    p.left = t
    p.height = 1 + max(self.getHeight(p.left),
                       self.getHeight(p.right))
    y.height = 1 + max(self.getHeight(y.left),
                       self.getHeight(y.right))
    return y
def insertion(self, root, key):
    if not root:
        return Node(key)
    elif key < root.data:
        root.left = self.insertion(root.left, key)
    else:
```



```
Page No.: 225
Date: / /

        root.right = self.insertion(root.right, key)
    root.height = 1 + max(self.getHeight(root.left),
                          self.getHeight(root.right))
    # find the balanced factor and balance the
    balanceFactor = self.getBalance(root)
    if balanceFactor < -1:
        if key > root.right.data:
            return self.leftRotate(root)
        else:
            root.right = self.rightRotate(root.right)
            return self.leftRotate(root)
    if balanceFactor > 1:
        if key < root.left.data:
            return self.rightRotate(root)
        else:
            root.left = self.leftRotate(root.left)
            return self.rightRotate(root)
    return root

def getMinValueNode(self, root):
    if root is None or root.left is None:
        return root
    return self.getMinValueNode(root.left)
def deletion(self, root, key):
    if not root:
        return root
    if key < root.data:
        root.left = self.deletion(root.left, key)
    if key > root.data:
        root.right = self.deletion(root.right, key)
```



```
else:
    if root.left is None:
        temp = root.right
        root = None
        return temp
    elif root.right is None:
        temp = root.left
        root = None
        return temp
    temp = self.getMinValueNode(root.right)
    root.data = temp.data
    root.right = self.deletion(root.right,
                               temp.data)
if root is None:
    return None
root.height = 1+max(self.getHeight(root.left),
                    self.getHeight(root.right))
balanceFactor = self.getBalance(root)
if balanceFactor < -1:
    if self.getBalance(root.right) <= 0:
        return self.leftRotate(root)
    else:
        root.right = self.rightRotate(
                                root.right)
        return self.leftRotate(root)
if balanceFactor > 1:
    if self.getBalance(root.left) >= 0:
        return self.rightRotate(root)
    else:
        root.left = self.leftRotate(root.left)
        return self.rightRotate(root)
```

Q-

(i)
(ii)
(iii)
(iv)

Page No.: 237
Date: / /

```
        return root
    def inorder(self, root):
        if not root:
            return
        self.inorder(root.left)
        print(root.data)
        self.inorder(root.right)
a = AVL
root = None
for ele in [21, 26, 30, 9, 4, 14, 28, 18, 15, 10, 2, 3, 7]:
    root = a.insertion(root, ele)
a.inorder(root)
a.deletion(root, 21)
print("After Deletion")
a.inorder(root)
```

Q- What is the maximum height of any AVL-tree with 7 nodes? Assume that the height of a tree with a single node is 0-

(i) 2
(ii) 3
(iii) 4
(iv) 5

$n(h) = n(h-1) + n(h-2) + 1$

min number of nodes in an AVL tree of height h

$n(0) = 1,\ n(1) = 2$

$n(0) = 1;\ n(1) = 2$

$n(2) = 1 + 2 + 1 = 4$

$n(3) = 1 + 4 + 2 = 7$ $\quad h = 3,\ n = 7$

Page No.: 28
Date: / /

Q- Which of the following is TRUE?

(i) The cost of searching an AVL tree is $\Theta(\lg n)$ but that of a binary tree is $O(n)$ (✓)

(ii) The cost of searching an AVL tree is $\Theta(\lg n)$ but that of a complete binary tree is $\Theta(n \lg n)$

(iii) The cost of searching a binary search tree is $O(\lg n)$ but that of an AVL tree $\Theta(n)$

(iv) The cost of searching a AVL tree is $\Theta(n \lg n)$ but that of a binary search tree is $O(n)$

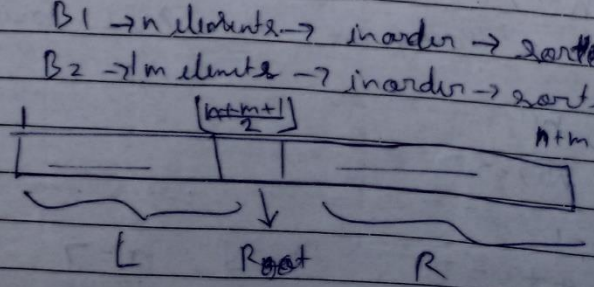
Q- Given two balanced binary search trees B1 having n elements and B2 having m elements, what is the time complexity of the best-known algorithm to merge them to form another balanced binary search tree containing m+n elements?

(i) $O(m+n)$ (✓)

(ii) $O(m \lg n)$

(iii) $O(n \lg n)$

(iv) $O(m^2 + n^2)$



Hash Table (Dictionary)

Page No.: 239
Date: 13/11/20

- It is collection of items. Avg: O(1). Time. Space complexity of

Operations -

Insert(S, x)
Delete(S, x)
Search(S, K)
} Very Very fast

→ Direct - access - Table - Simplest way.

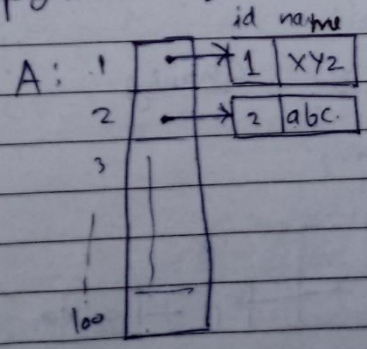
Eg- Store names & roll nos of students in a class max 100

Unique identifiers (key). {1, 2, ——— 100}

id	name
1	XYZ
2	ZAB

Set of U = {1, 2, 3. - - - 100}
|U| = size of U = 100.

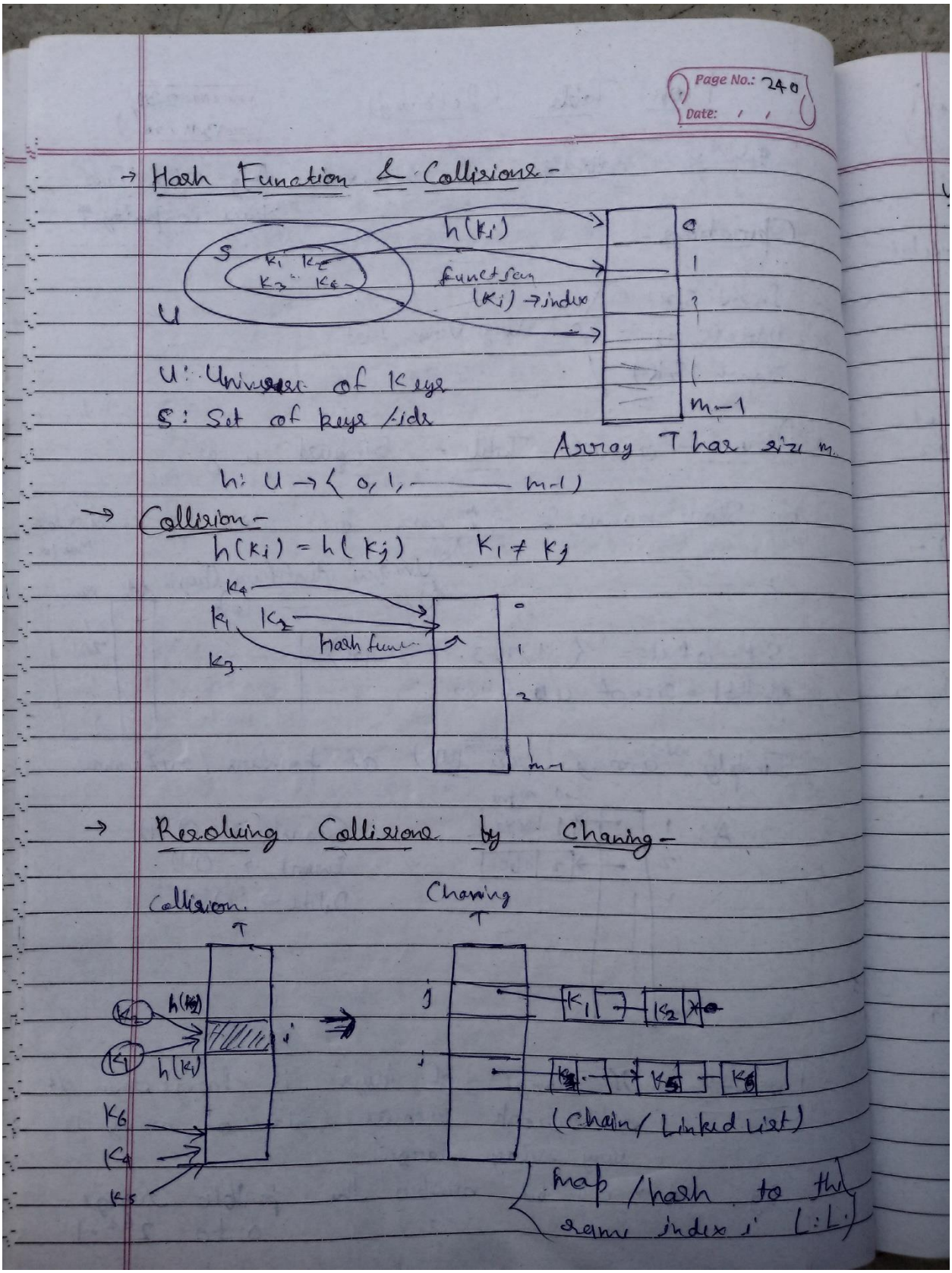
Simply using array (size |U|) of pointers / references



Search → O(1)
Insert → O(1)
Delete → O(1)

- Limitation - If range of keys is large then it not work because size of array is very-very large.

Eg- id → 64 bit number the possible range - 0 to $2^{64}-1$


Page No.: 240
Date: / /
→ Hash Function & Collisions -
h(ki)
S
k1 k2
k3 k6
U
function
(ki) → index
0
1
2
m-1
U: Universe of keys
S: Set of keys /ids
Array T has size m.
h: U → {0, 1, ——— m-1}
→ Collision -
h(ki) = h(kj) ki ≠ kj
k4
k1 k2
k3
Hash function
0
1
2
m-1
→ Resolving Collisions by Chaining -
Collision
Chaining
T
h(k2)
h(k1)
k6
k4
k5
(chain / Linked list)
map / hash to the same index i (i.e.)

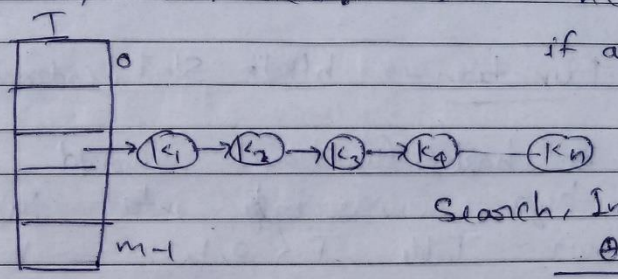
Page No.: 241
Date: / /

Worst-case -

n keys - $k_1, k_2, \ldots, k_n$.

$|S| = n$, $h(k_1) = h(k_2) = h(k_3) = \ldots = h(k_n) = j$

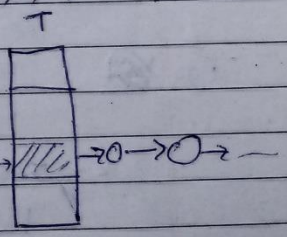
if all the keys collide



Search, Insert, Delete
$\Theta(n)$

Simple Uniform Hashing

{ for each $k \in S$ is equally to be hashed to any slot in T $\{0, 1, 2, \ldots, m-1\}$



n = number of keys in S

m = number of slots / indices in T

$$\text{Load factor} = \frac{\text{average number of keys}}{\text{slot}}$$

↓ average length of each chain

$$\boxed{\alpha = n/m}$$ Load Factor.

Time complexity to search for a key x.

hash function - $\Theta(1)$

average comparisions $\Theta(\alpha) = \Theta(1+\alpha)$

Page No.: 242
Date: / /

if $\alpha = \theta(1)$ then $\theta(1)$

Worst Case - all keys hash to the slot/bucket $\theta(n)$

→ Hash Function - $h(K)$ = Slot / index / bucket

A Good hash-function should distribute the keys uniformly into the slots in our Table $T\{0, 1, 2, \ldots m-1\}$ 2-

1- Division Method (Modulo Hash Function) -

$$h(K) = K \bmod m$$

→ very popular (K%m)

where, $K \bmod m$ → reminder by dividing K by m

Eg- T → size m

Let $m = 101$

$K = 56$; $m = 101$

$h(K) = 56 \bmod 101 = 56$

$K = 206$; $m = 101$

$h(K) = 206 \bmod 101 = 4$

$K = 4$; $m = 101$

$h(K) = 4 \bmod 101 = 4$

→ hash collision (chaining)

Page No. 243
Date: / /

Let $m = 2^r = 2^6$

$k = 101100111\underbrace{101010}$ ← last 6 digits

$k \bmod m = 101010$

very fast to compute. $h(k)$ is dependent only on a few bits of k

Recommendation- Pick m to be a prime number

2- Multiplicative Method- (most widly used)

$0 < A < 1$: $m = 2^p$

$$h(k) = \lfloor m * (k \cdot A \bmod 1) \rfloor$$

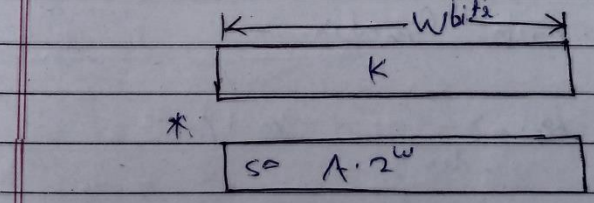
↑ integer.

$k \cdot A \bmod 1 \Rightarrow$ fractional part of $k \cdot A$

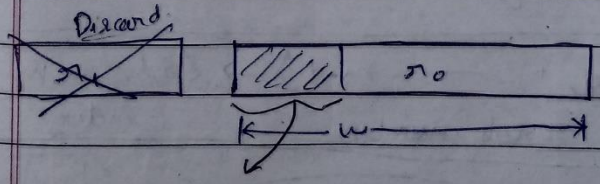
↑ real Number

Eg- $2.6132 \bmod 1 = 0.6132$

binary representation of k



$* \quad s = A \cdot 2^w$



$m = 2^p$

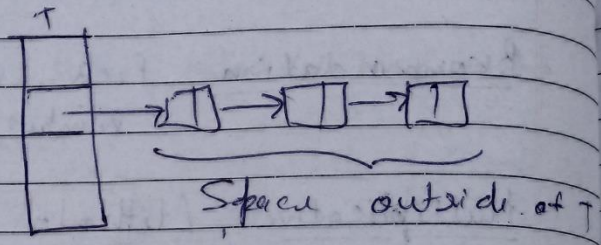
p bits = $h(k) = \lfloor m * (kA \bmod 1) \rfloor$

Good Value for A is - $A = \frac{\sqrt{5}-1}{2} = 0.618$

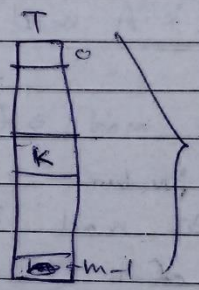
Date: / /

→ Open Addressing (Close Hashing) -

In case of chaining we use space outside of our table.



What if I want to use only the space in T



Control over the space

In Typical hash function -

$h : U \rightarrow \{0, 1, 2, \ldots, m-1\}$

key index

In Open Addressing -

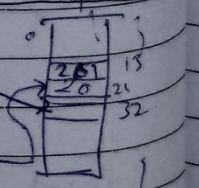
$h : U \times \{0, 1, \ldots, m-1\} \rightarrow \{0, 1, \ldots, m-1\}$

key probe number index

h: (K=26, 0) = 32

h: (K=20, 0) = 15 already filled

h: (K=20, 1) = 21



Page No.: 245
Date: / /

① $h(K,0)$ if collision
② increase prob-number. $h(K,1)$ ——— $h(K,m-1)$

(i) **Linear Probing** —

$$h(k,i) = (h'(K) + i) \bmod m$$

Key, Probe number.

Let, $K = 32$; $h'(K) = 12$

$h(K=32, i=0) = (12+0) \bmod 101 = 12$ (collision)

$h(K=32, i=1) = (12+1) \bmod 101 = 13$ (collision)

$h(K=32, i=2) = (12+2) \bmod 101 = 14$ ✓

	0
14	12
15	13
32	14
	m=101

Note: Prob-number 0, 1, 2, ——— m-1

(ii) **Double Hashing** — (Very popular)

$h_1(x)$ & $h_2(x)$: Key → index
h_1 & h_2 : $U \to \{0, 1, 2, \ldots, m-1\}$
} Standard hash functions.

$$h(K,i) = \{h_1(K) + (i \times h_2(K))\} \bmod m$$

Key, Prob-number

Page No.: 246
Date: / /

(iii) Quadratic Probing -

$h'(K) : U \rightarrow \{0, 1, 2, \ldots, m-1\}$ → Standard hash fun.
(Key, index)

$$h(K,i) = \{h'(K) + (C_1 * i) + (C_2 * i^2)\} \bmod m \quad C_2 \neq 0$$

$i = 0$ to $m-1$

Note- if $C_2 = 0$ ⇒ Linear Probing.

→ Application- Sparse Matrix Representation-

n = num. of rows
m = num. of items
O(n×m) Space
• most of cells will be 0
• Very few cells will be non-zero.

	j_1	j_2	j_3		j_{m-1}
i_1					
i_2	1	0	1	0	1
		1		0	
i_{n-1}					

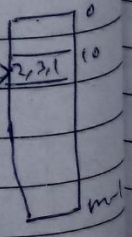
← Purchase info

n×m

Instead of storing all the cell's information Lets only store the info of non-zero cells.

	row	Col	val
2	2	3	1
	4	8	1
	3	21	1

key (row, col)
str → (2, 3) → integer
h("2,3") = 10



Space Θ(m)
if we use efficient hashing the Θ(m) reduced to Θ(2)

Page No.: 247
Date: / /

Q- Given the following input (4322, 1334, 1471, 9679, 1989, 6171, 6173, 4199) and the hash function - X mod 10, which of the following statements are true?

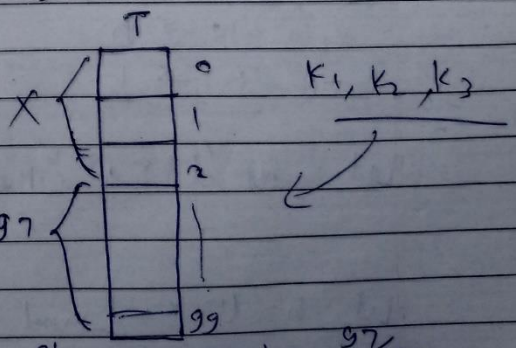
(i) 9679, 1989, 4199 hash to the same value
(ii) 1471, 6171 hash to the same value
(iii) All elements hash to the same value
(iv) Each element hashes to a different value

(A) 1 only
(B) 2 only
(C) 1 and 2 only ✓
(D) 3 or 4

Q- Consider a hash table with 100 slots. Collisions are resolved using chaining. Assuming simple uniform hashing, what is the probability that the first 3 slots are unfilled after the first 3 insertions?

(i) $(97 \times 97 \times 97)/100^3$ ✓
(ii) $(98 \times 98 \times 97)/100^3$
(iii) $(97 \times 96 \times 95)/100^3$
(iv) $(97 \times 96 \times 95)/(3! \times 100 \wedge 3)$

(Simple Uniform hashing) ↓ every key is equally likely to be hashed to any of the slots.



Prob (K1 is hashed to 1st 3 slots) = 97/100

we use chaining then - $\frac{97}{100} \times \frac{97}{100} \times \frac{97}{100}$ ✓

Page No.: 248
Date: / /

Q- Which one of the following hash functions on integers will distribute keys most uniformly over 10 buckets numbered 0 to 9 for i ranging from 0 to 2020?

(i) $h(i) = i^2 \bmod 10$

(ii) $h(i) = i^3 \bmod 10$

(iii) $h(i) = (11 * i^2) \bmod 10$

(iv) $h(i) = (12 * i^2) \bmod 10$

Let $h(i) = i^2 \bmod 10$, {0, 1, 4, 5, 6, 9} only 6 slots required

for

$i = 0 \to 0 \xrightarrow{h} 0$

$i = 1 \to 1 \xrightarrow{h} 1$

$2 \to 4 \xrightarrow{h} 4$

$3 \to 9 \to 9$

$4 \to 16 \to 6$

$5 \to 25 \to 5$

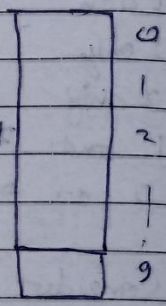
$6 \to 36 \to 6$

$7 \to 49 \to 9$

$8 \to 64 \to 4$

$9 \to 81 \to 1$

all numbers collided except 0, and 5



$10^2 = 100 \Rightarrow 0$

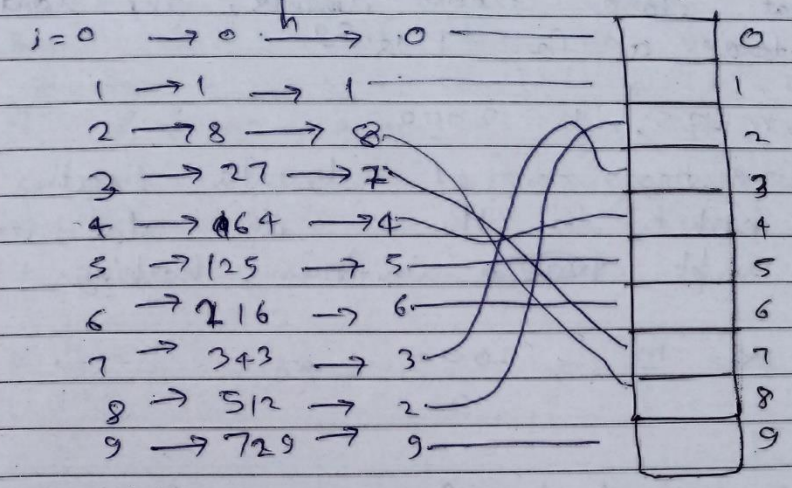
$11^2 = 121 \Rightarrow 1$

Let $h(i) = (11 \times i^2) \bmod 10$, {0, 1, 4, 5, 6, 9} only 6 slots

Let $h = (12 \times i^2) \bmod 10$, {0, 1, 4, 5, 6, 9} only 6 slots

Page No.: 249
Date: / /

Let $h(i) = i^3 \bmod 10$



(9 slots required.)

Q- Which of the following statement(s) is TRUE?

(A) A hash function takes a message of arbitrary length and generates a fixed length code.

(B) A hash function takes a message of fixed length and generates a code of variable length.

(C) A hash function may give the same hash value for distinct messages.

(i) A only
(ii) B and C only
(iii) A and C only (✓)
(iv) B only

$h(K) = \{0/1 \text{ ---- } m-1\}$

↓ String (variable length) → fixed length.

Page No.: 250
Date: / /

Q- Given a hash table T with 25 slots that stores 2000 elements, the load factor α for T is 80

$m = 25, \quad n = 2000$

α = average no. of elements that are hashed to the same slot assuming simple uniform hashing.

$$\alpha = \frac{n}{m} = \frac{2000}{25} = 80$$

Q- Consider a hash function that distributes keys uniformly. The hash table size is 20. After hashing of how many keys will the probability that any new key hashed collides with an existing one exceed 0.5.

(i) 5
(ii) 6
(iii) 7
(iv) 10 (✓)

(5) Prob 'k' will collide with existing key.

$$\frac{5}{20} = \frac{1}{4} = 0.25$$

(6) $\rightarrow \frac{6}{20} = \frac{3}{10} = 0.3$

(7) $\rightarrow \frac{7}{20} < 0.5$

(10) $\rightarrow \frac{10}{20} = 0.5$

Page No.: 251
Date: / /

Q- The Characters of the string K R P C S N Y T J M are inserted into a hash table of size 10 using hash function. - $h(X) = ((ord(X) - ord(A) + 1)) \bmod 10$ If linear probing is used to resolve collision, then the following insertion cause collision -

(A) Y
(B) C
(C) M (✓)
(D) P

Ord(X) = alphabatic order of X
like - A = 1, B = 2, C = 3, — Z = 26

$h(X) = ((ord(X) - ord(A) + 1)) \bmod 10$
$= ((ord(X) - 1 + 1)) \bmod 10$
$= ord(x) \bmod 10$

Char	ord	ord(x) mod 10
K	11	1
R	18	8
P	16	6
C	3	3
S	19	9
N	14	4
Y	25	5
T	20	0
J	10	0 ← collision
M	13	3 ← collision.

Page No.: 253
Date: / /

Q- Consider the following statements -

S1- An insertion in an AVL with n nodes requires $\Theta(n)$ rotations.

S2- Finding the minimum value in an AVL tree containing n elements takes $O(\lg n)$ time

S3- Both insert and find in an AVL tree takes $O(\lg n)$ time.

S4- Finding the K-th largest item in a standard AVL tree implementation containing n elements takes $O(n)$ time.

S5- A set of numbers are inserted into an empty BST in sorted order and inserted into an empty AVL tree in random order. Listing all elements in sorted order from the BST is $O(n)$, while listing them in order from the AVL tree is $O(\lg n)$.

Which of the above statement(s) are INCORRECT?

(A) S1 only
(B) S1, S2, and S3
(C) S4 and S5
(D) S1 and S5 (✓)

Q- The maximum height an AVL tree can have with 87 nodes?

(A) 9
(B) 7 (✓)
(C) 6
(D) 8

Page No.: 253
Date: / /

min. number nodes in AVL tree of h

$n(h) = 1 + n(h-1) + n(h-2)$

$n(0) = 1;\ n(1) = 2$

$n(2) = 3+1 = 4$

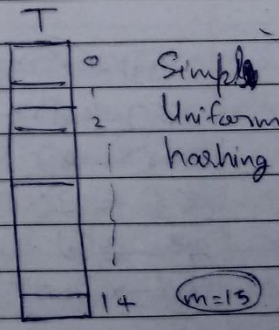
$n(3) = 7,\ n(4) = 12$

$n(5) = 20,\ n(6) = 33$

$n(7) = 54,\ n(8) = 88$

min of 88 node → height 8

Q- Consider a hash function that distributed key uniformly. The hash table size is 15. The number of entries in the hashtable after which the probability that any new key hashed collides with an existing one exceeds 0.6 is 9



$1 \to 2^{nd}$ key = P(2nd key collides with non empty slot)

$= 1/15$

$2 \to 3^{rd}$ key $\to 2/15$

$n \to (n+1)^{th}$ key $\Rightarrow \alpha_m = n/15$

$\frac{n}{15} = 0.6$

$\boxed{n = 9}$

Page No.: 254
Date: / /

Q- The number of different insertion sequences of numbers {14, 31, 54, 68, 71, 92} on an initially empty hash table H of size 66 and a hash function h(K) = K mod 6 with linear probing scheme for collision resolution such that the hash table obtained after the insertion looks as shown in table below is equal to 120

0	54
1	31
2	14
3	68
4	92
5	71
6	

h(K) = K mod 6 , linear probing.

14 → 2
31 → 1
54 → 0
68 → 2 → collide
71 → 5
92 → 2

54	0
31	1
14	2
68	3
	4
71	5
	6
	65

14, 68, 92 have to be placed in the sequence in this exact order.

31, 54, 71 → Can be placed anywhere in sequence

71 31 (14) 54 (68) (92)

6 × 5 × 4 × 1 = 120

(A)
(B)
(C)
(D)

Q-

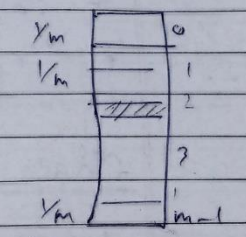
Q-

(i)
(ii)
(iii)
(iv)

Page No.: 255
Date: / /

Q- Suppose we use a hash function h to hash n distict keys into an array T of length m. Assuming simple uniform hashing, the expected number of collisions (Assume $n <= m$) ?

(A) $n^2/2m$
(B) $(n^2-n)/2m$ (✓)
(C) $(n^2+n)/2m$
(D) $(n-1)^2/2m$



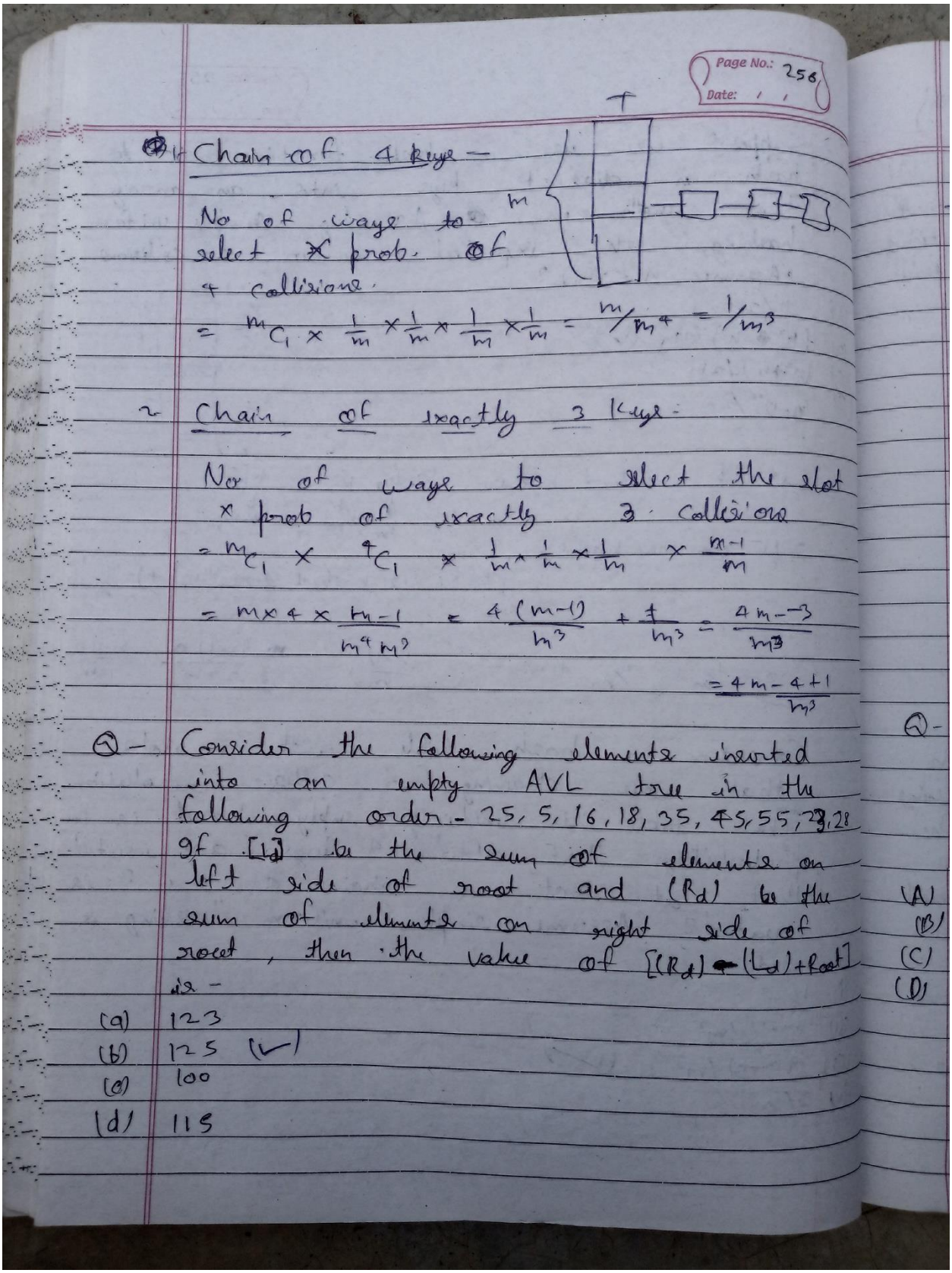
Let n distinct keys are- $k_1, k_2, k_3 \dots k_n$
$\downarrow 0 \quad \downarrow 1/m \quad \downarrow 2/m \quad \downarrow (n-1)/m$

Simple uniform Hashing- $[P(k_i \text{ is hashed to } i^{th} \text{ slot}) = \frac{1}{m}]$

$0 + 1/m + 2/m + \dots \frac{(n-1)}{m} = \frac{(n-1)\,n}{2*m} = \frac{(n-1)\,n}{2m}$

Q- Consider a hash table with m slots that uses chaining for collision resolution. The table is initially empty. What is the probability that after 4 keys are inserted, that at least a chain of size 3 is created? (Assume simple uniform hashing is used) -

(i) $1/m^2$
(ii) $(m-1)/m^3$
(iii) $(4m-3)/m^3$ (✓)
(iv) $3/m$



Page No.: 256
Date: / /

1. Chain of 4 keys –

No of ways to select × prob. of 4 collisions.

$= {}^{m}C_1 \times \frac{1}{m} \times \frac{1}{m} \times \frac{1}{m} \times \frac{1}{m} = \frac{m}{m^4} = \frac{1}{m^3}$

2. Chain of exactly 3 keys –

No of ways to select the slot × prob of exactly 3 collisions

$= {}^{m}C_1 \times {}^{4}C_1 \times \frac{1}{m} \times \frac{1}{m} \times \frac{1}{m} \times \frac{m-1}{m}$

$= m \times 4 \times \frac{m-1}{m^4 m^3} = \frac{4(m-1)}{m^3} + \frac{1}{m^3} = \frac{4m-3}{m^3}$

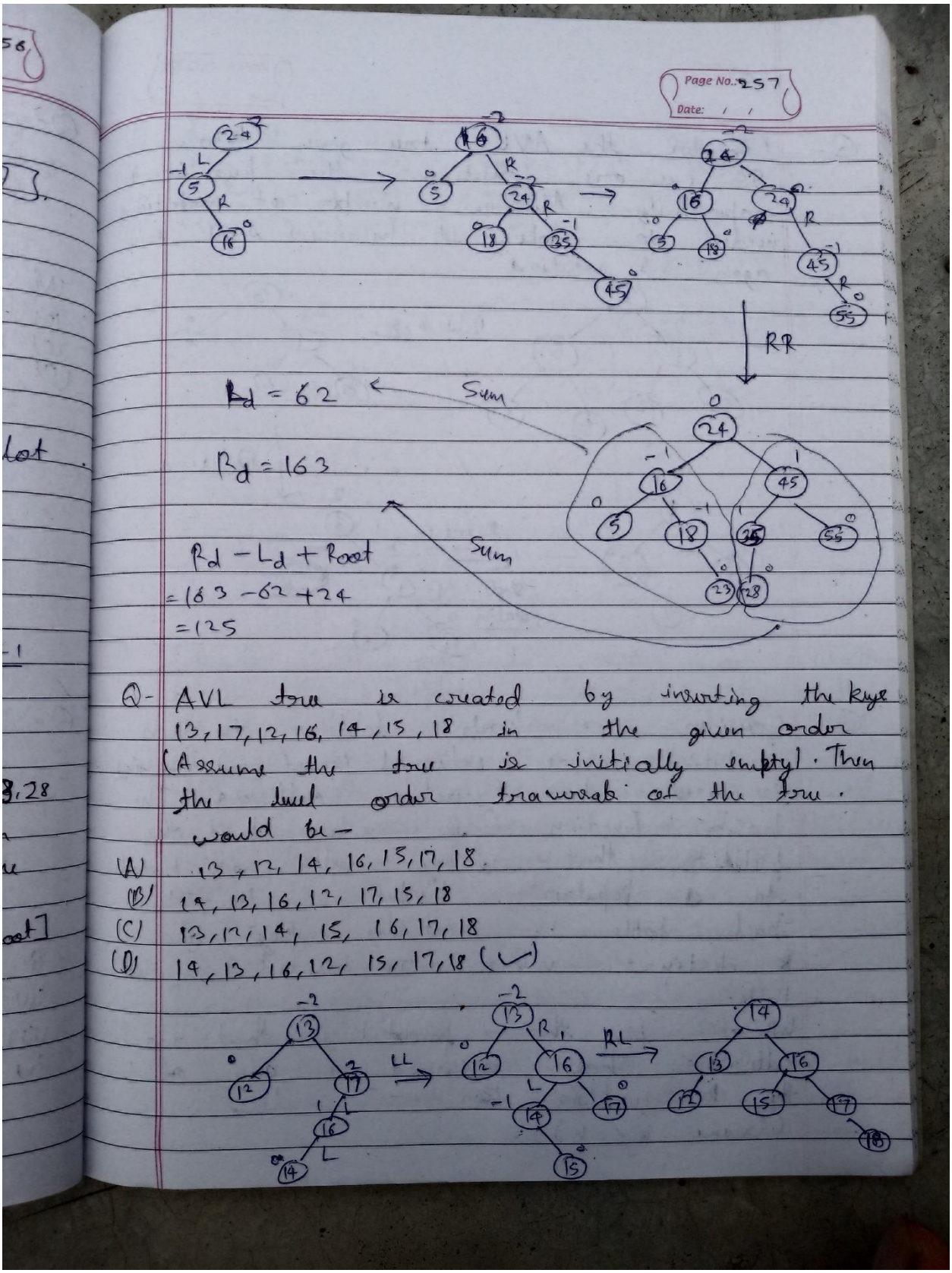
$= \frac{4m-4+1}{m^3}$

Q- Consider the following elements inserted into an empty AVL tree in the following order - 25, 5, 16, 18, 35, 45, 55, 23, 28

If [Ld] be the sum of elements on left side of root and (Rd) be the sum of elements on right side of root, then the value of [(Rd) − (Ld) + Root] is -

(a) 123
(b) 125 (✓)
(c) 100
(d) 115

Q-
(A)
(B)
(C)
(D)



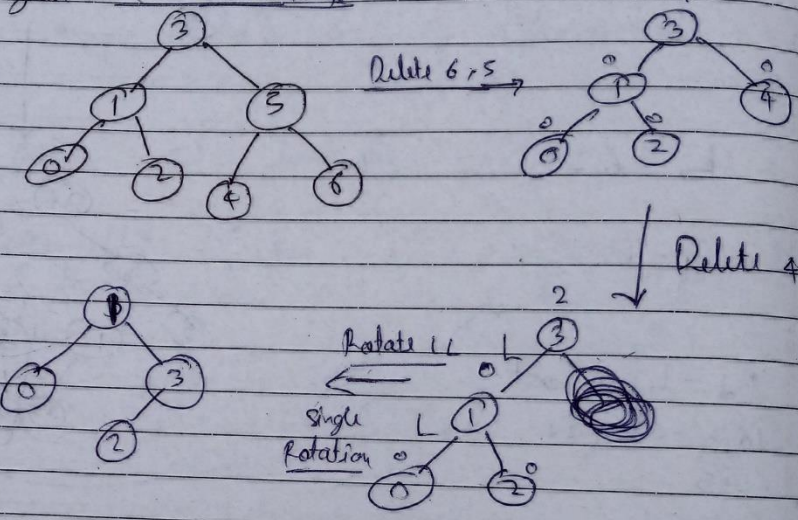
Q- AVL tree is created by inserting the keys 13, 17, 12, 16, 14, 15, 18 in the given order (Assume the tree is initially empty). Then the level order traversal of the tree would be -

(A) 13, 12, 14, 16, 15, 17, 18.
(B) 14, 13, 16, 12, 17, 15, 18
(C) 13, 12, 14, 15, 16, 17, 18
(D) 14, 13, 16, 12, 15, 17, 18 (✓)

Page No.: 258
Date: / /

Q- Consider the AVL tree given below, If we are deleting the keys 6, 5, 4 respectively. Minimum number of rotations needed to make it balanced AVL tree again? (1 rotations)

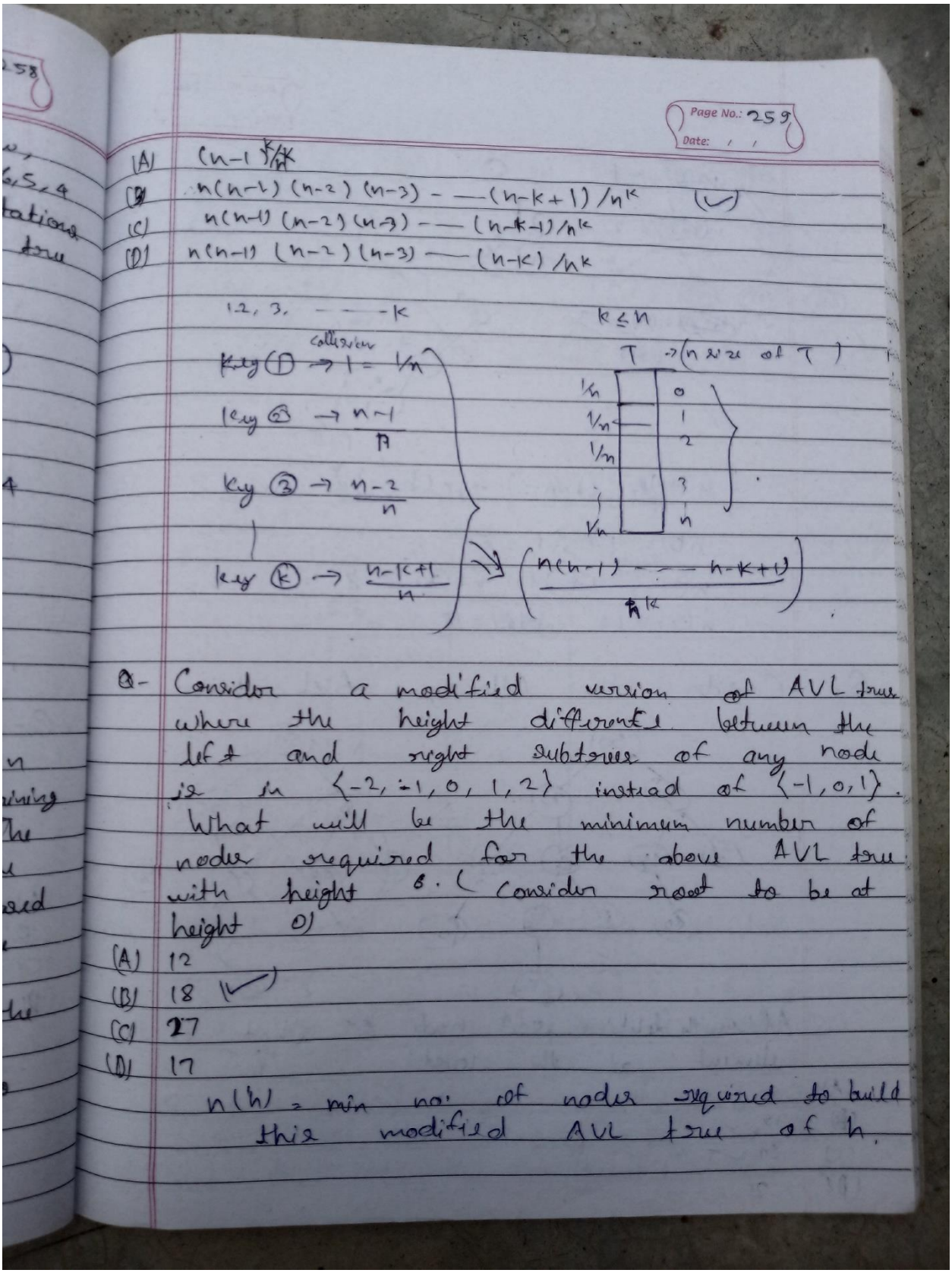
(A)
(B)
(C)
(D)



Q-

Q- Consider a hash table with n buckets, where external (overflow) chaining is used to resolve collisions. The hash function is such that the probability that a key value is hased to a popular bucket is $\frac{1}{n}$. The hash table is initially empty and k distinct values are inserted in the table.
What is the probability that no collisions has occured in any of the k questions?
(Assume $k \leq n$)

(A)
(B)
(C)
(D)



Page No.: 259
Date: / /

(A) $(n-1)^k/n^k$
(B) $n(n-1)(n-2)(n-3)\cdots(n-k+1)/n^k$ ✓
(C) $n(n-1)(n-2)(n-3)\cdots(n-k-1)/n^k$
(D) $n(n-1)(n-2)(n-3)\cdots(n-k)/n^k$

1, 2, 3, ---- k $k \le n$

collision
key (1) → 1 = 1/n
key (2) → (n-1)/n
key (3) → (n-2)/n
key (k) → (n-k+1)/n

T → (n size of T)
1/n 0
1/n 1
1/n 2
3
1/n n

$\Rightarrow \left(\frac{n(n-1)\cdots(n-k+1)}{n^k}\right)$

Q- Consider a modified version of AVL tree where the height difference between the left and right subtrees of any node is in $\{-2, -1, 0, 1, 2\}$ instead of $\{-1, 0, 1\}$. What will be the minimum number of nodes required for the above AVL tree with height 6. (Consider root to be at height 0)

(A) 12
(B) 18 ✓
(C) 27
(D) 17

n(h) = min no. of nodes required to build this modified AVL tree of h.

Page No.: 260
Date: / /

n(0) = 1

n(1) = 2

n(2) = 3

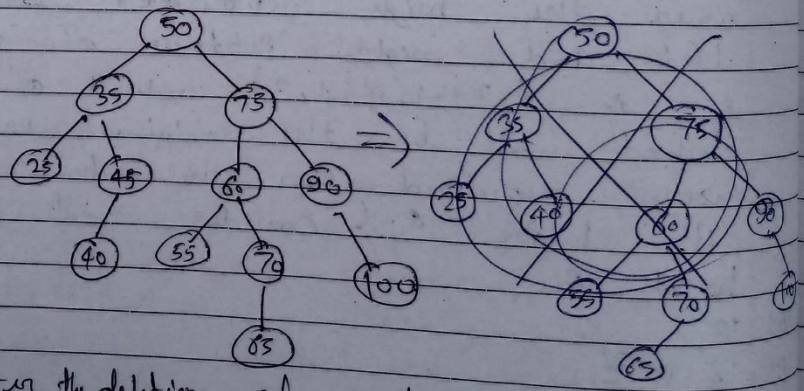
{ n(h) = 1 + n(h-1) + n(h-2) }

n(3) = 1 + 3 + 1 = 5

n(4) = 1 + n(3) + n(2) = 8

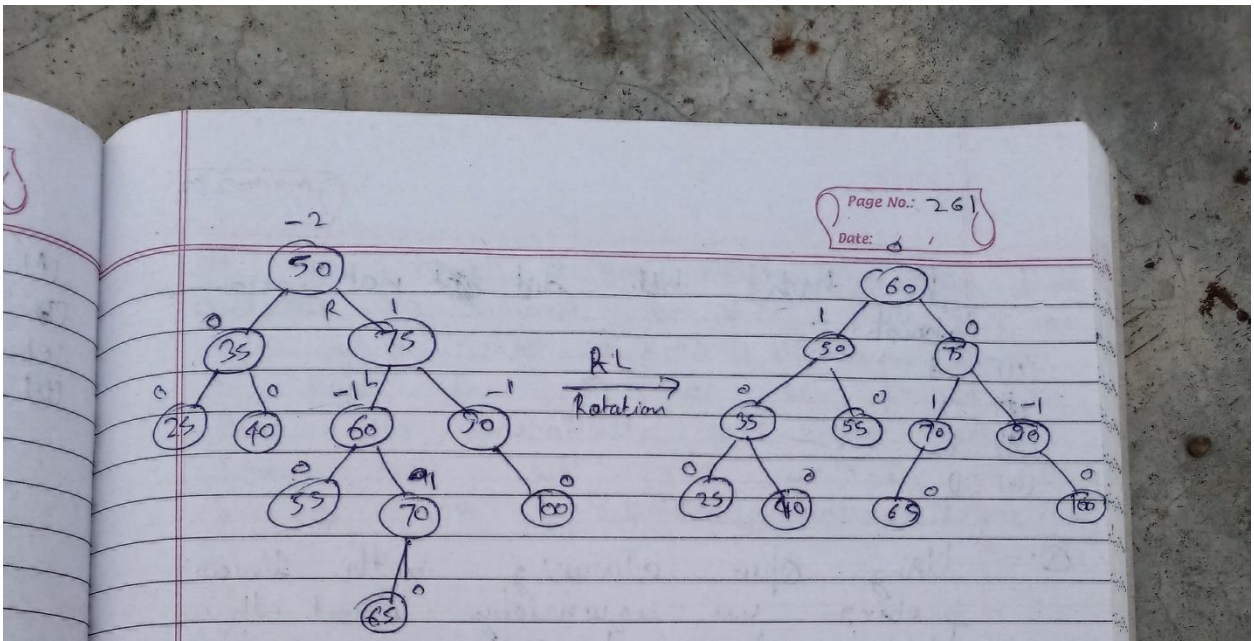
n(5) = 12, n(6) = 18

Q- Consider the following AVL tree-



After the deletion of node 45 what is the element at the root?

(A) 55
(B) 35
(C) 60 ✓
(D) 75



Q- Suppose that the following keys are inserted in same order into an initially empty linear-probing hash table-

Key	hash
A	1
D	5
L	6
M	0
N	1
S	6
X	4

Following is the resultant hash table-

0	1	2	3	4	5	6
S	M	N	A	X	D	L

If the initial size of the hash table was 7. Which of the following keys is definitely not the last key inserted? (Assume that

Page No.: 262
Date: / /

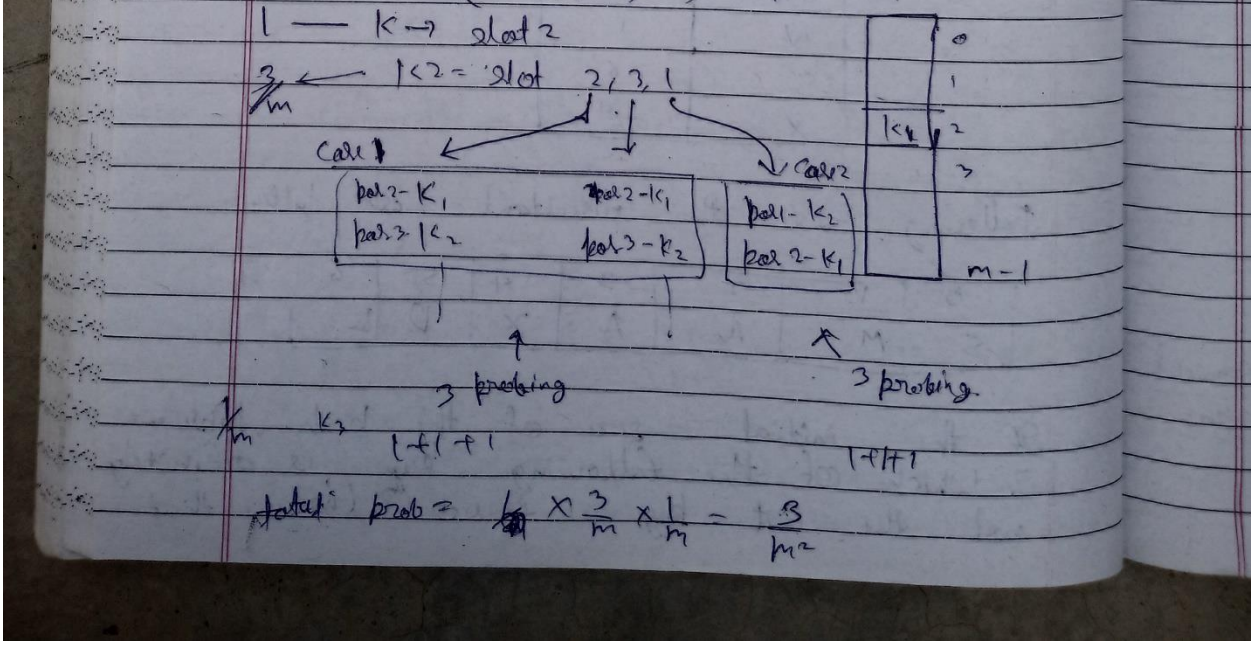
the hash table did not grow or shrink).

(i) A
(ii) X
(iii) L (✓)
(iv) D

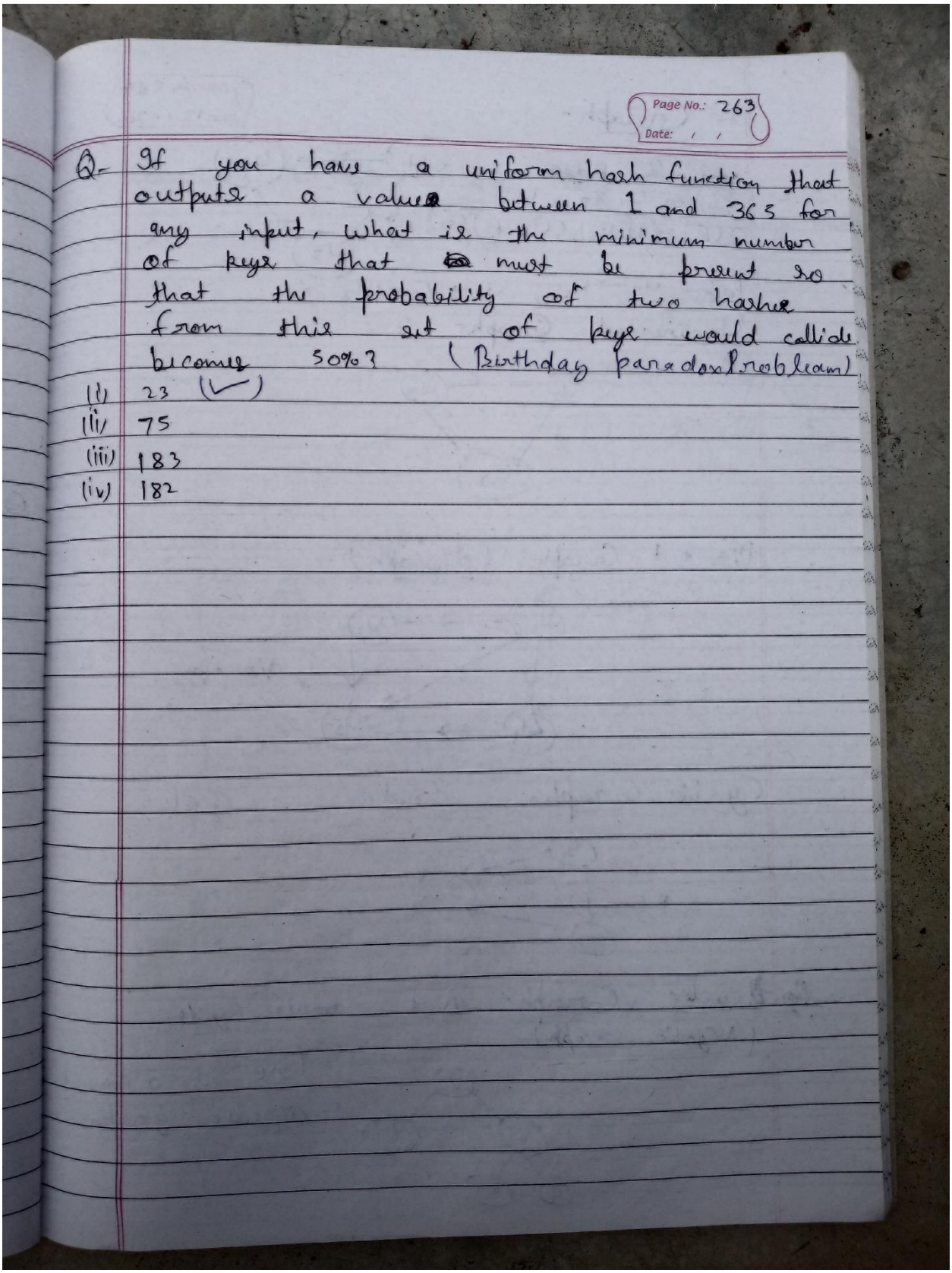
Q.- Using open addressing with linear probing, we sequentially insert three distinct keys k_1, k_2, k_3 into a hash table of size m. Assuming simple uniform hasing, what is probability that we will need three probes, when inserting the third key k_3?

(i) 3/m
(ii) 2/m^2
(iii) 3/m^2 (✓)
(iv) 2/m

$P(k_1$ hashed into slot $j) = \frac{1}{m}$



total prob = $\frac{3}{m} \times \frac{1}{m} = \frac{3}{m^2}$



Page No.: 263
Date: / /

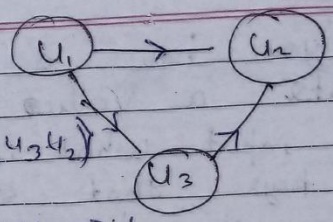
Q- If you have a uniform hash function that outputs a value between 1 and 365 for any input, what is the minimum number of keys that must be present so that the probability of two hashes from this set of keys would collide becomes 50%? (Birthday paradox Problem)

(i) 23 (✓)
(ii) 75
(iii) 183
(iv) 182

Page No.: 264
Date: 19/11/20

Graph -

Vertex & Edge -

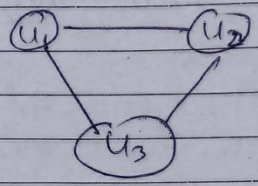


$G(V) = u_1, u_2, u_3$

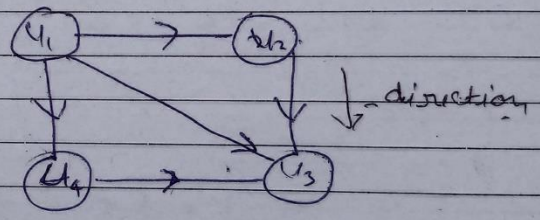
$G(E) = \{(u_1, u_2), (u_1, u_3), (u_3, u_2)\}$

Path - $u_1 \to u_2$, $u_1 \to u_3 \to u_2$

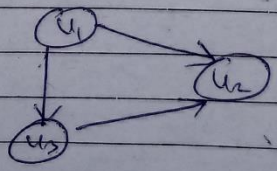
Undirected Graph -



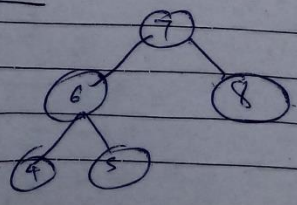
Directed Graph - (di-graph) -



Cyclic Graph - Contain a cycle.



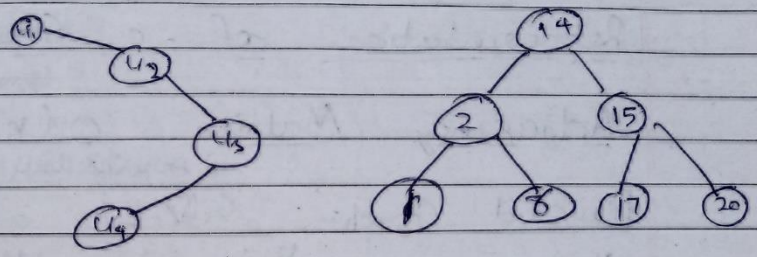
Non-Cyclic Graph - Not contain cycles.
(ACyclic Graph) -



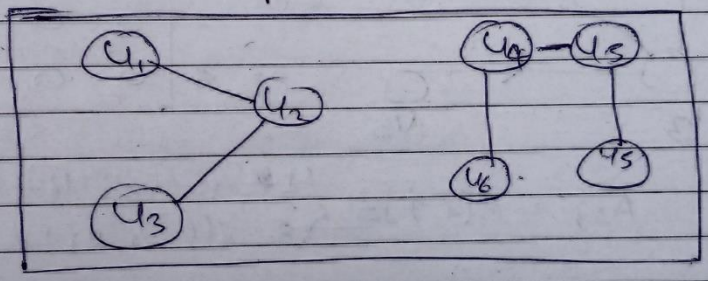
Tree is a connected acyclic graph.

Page No.: 265
Date: / /

Connected Graph- Graph where there is a ~~that~~ path from every node to every other node.



Disconnected Graph-



Properties of Graph-

G(V, E) if graph has m edges and n vertices then -

(V: Set of Vertex, E: Set of Edges)

Total No. of element in set E

$$|E| = \frac{(n-1)(n)}{2}$$

$$\boxed{|E| = O(|V|^2)}$$

$$\log(|E|) = O(\log|V|)$$

Page No.: 266
Date: / /

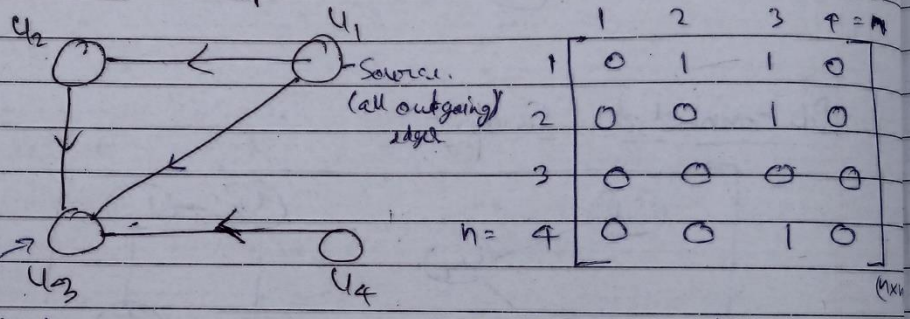
In Connected graph-

$$|E| > |V| - 1$$

Representation of a Graph- ~~Adjacency Matrix~~

Adjacency Matrix- $O(n^2)$ Space.
⇒ irrespective of the no. of edges

Directed Graph- $G(V, E)$, $|E| = m$ $|V| = n$

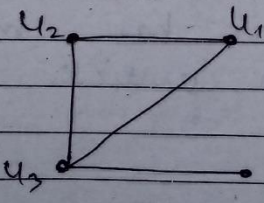


	1	2	3	4 = n
1	0	1	1	0
2	0	0	1	0
3	0	0	0	0
n = 4	0	0	1	0

(n x n)

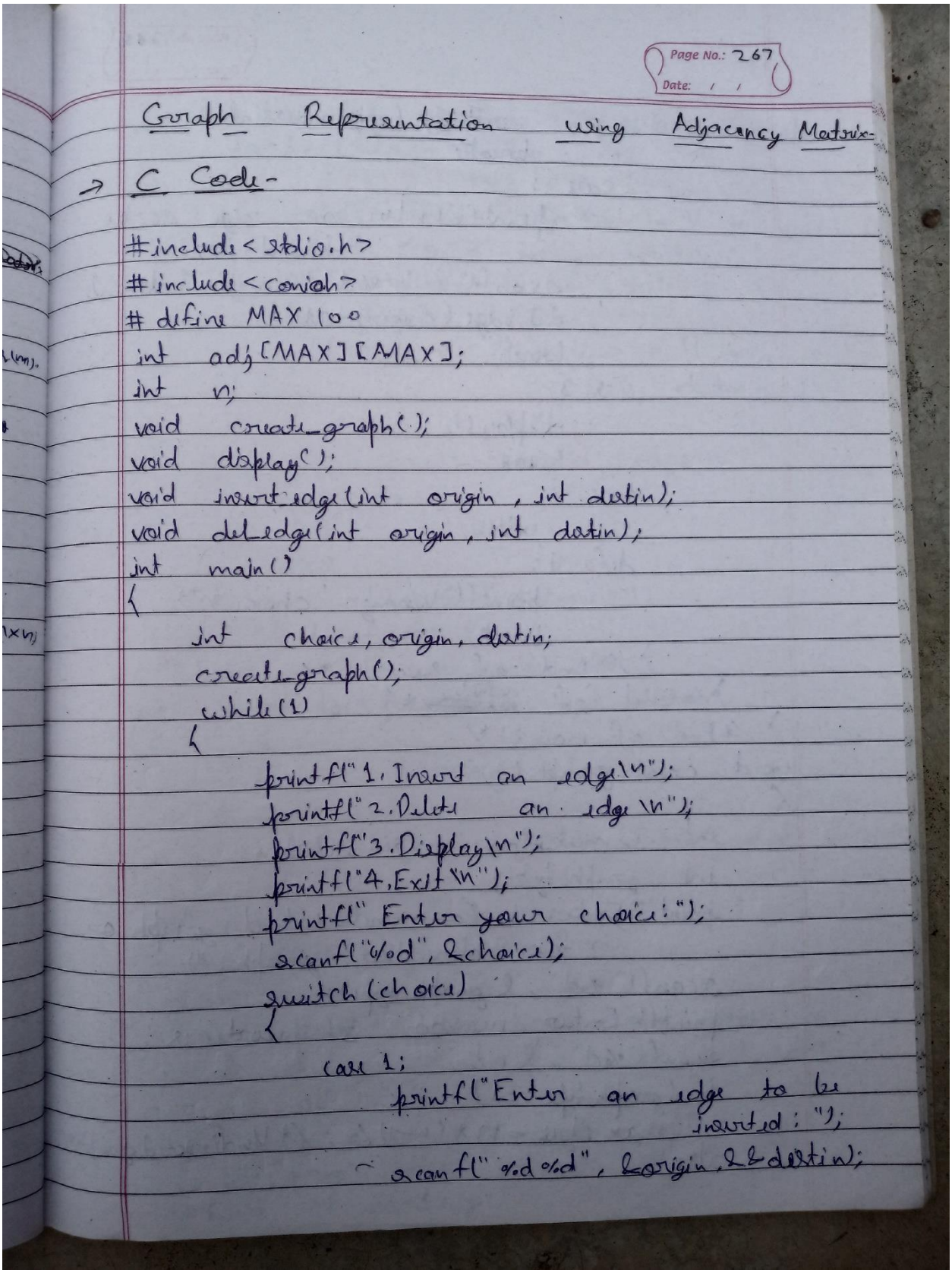
$$A_{i,j} = A[i,j] = \begin{cases} 1 & \text{iff } (u_i, u_j) \in E \\ 0 & \text{iff } (u_i, u_j) \notin E \end{cases}$$

Undirected Graph { if $(u_i, u_j) \in E$ then $(u_j, u_i) \in E$ }



	1	2	3	4
1	0	1	1	0
2	1	0	1	0
3	1	1	0	1
4	0	0	1	0

(Symmetric Matrix)

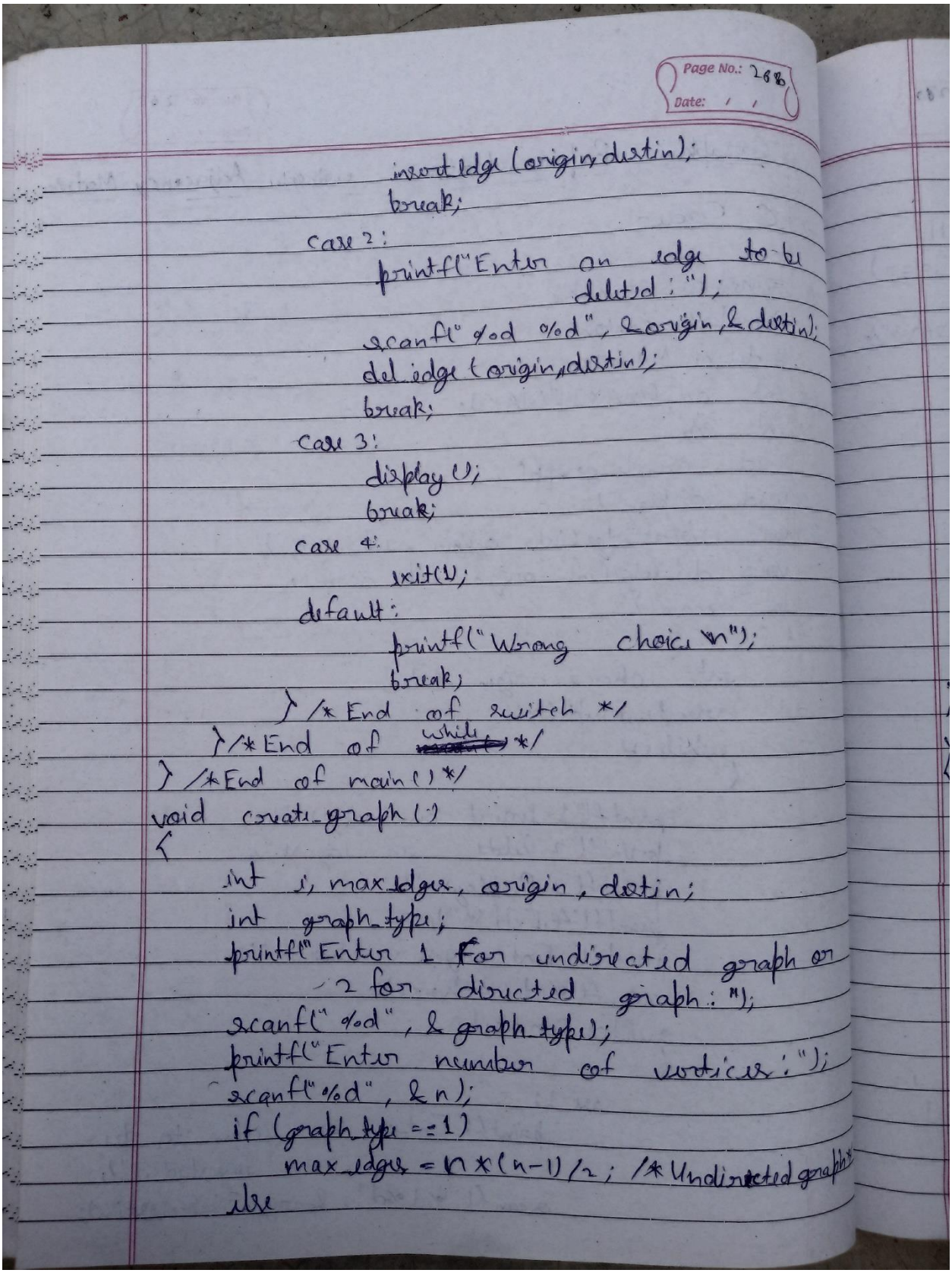


Page No.: 267
Date: / /

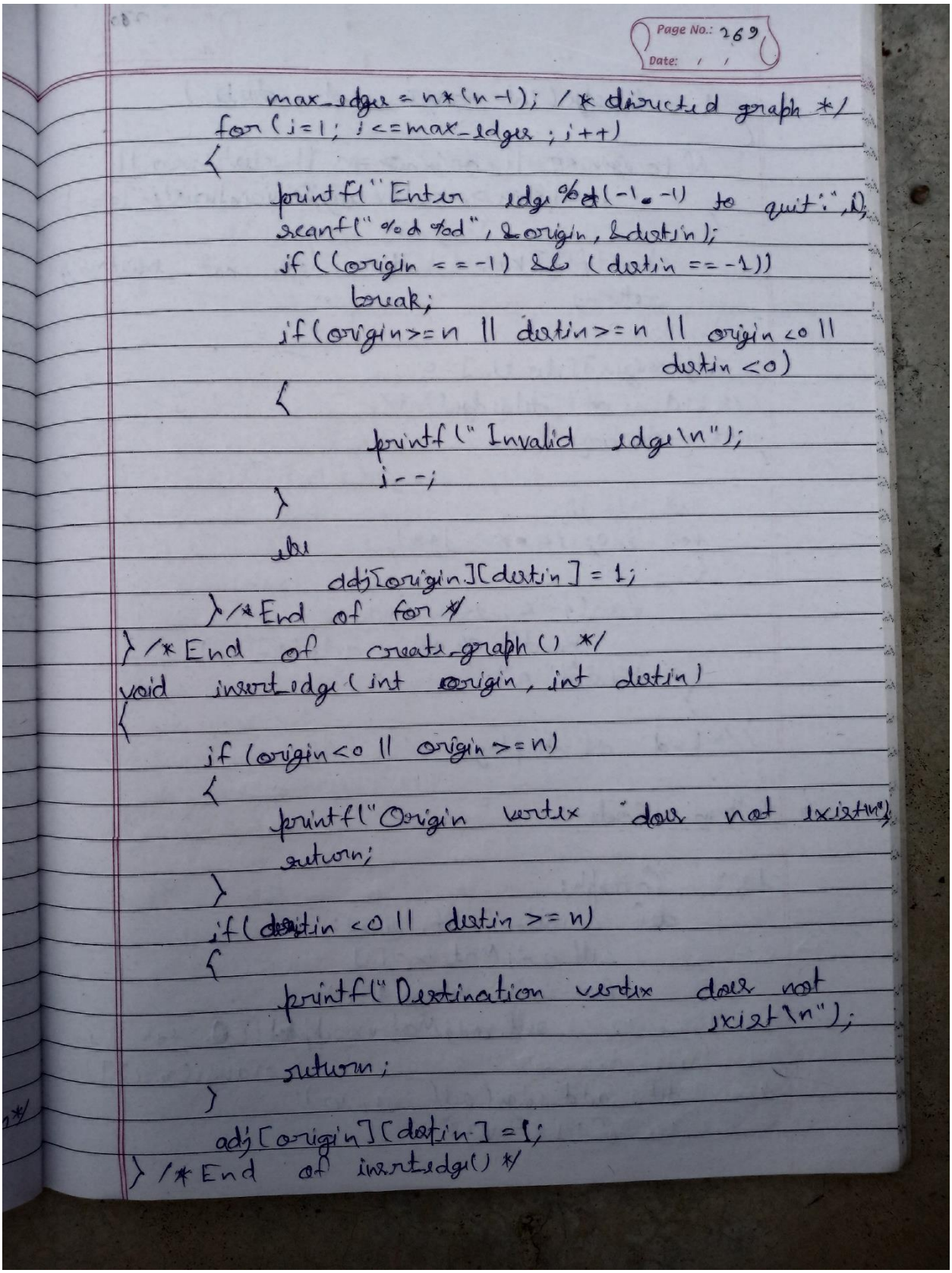
Graph Representation using Adjacency Matrix-

→ C Code-

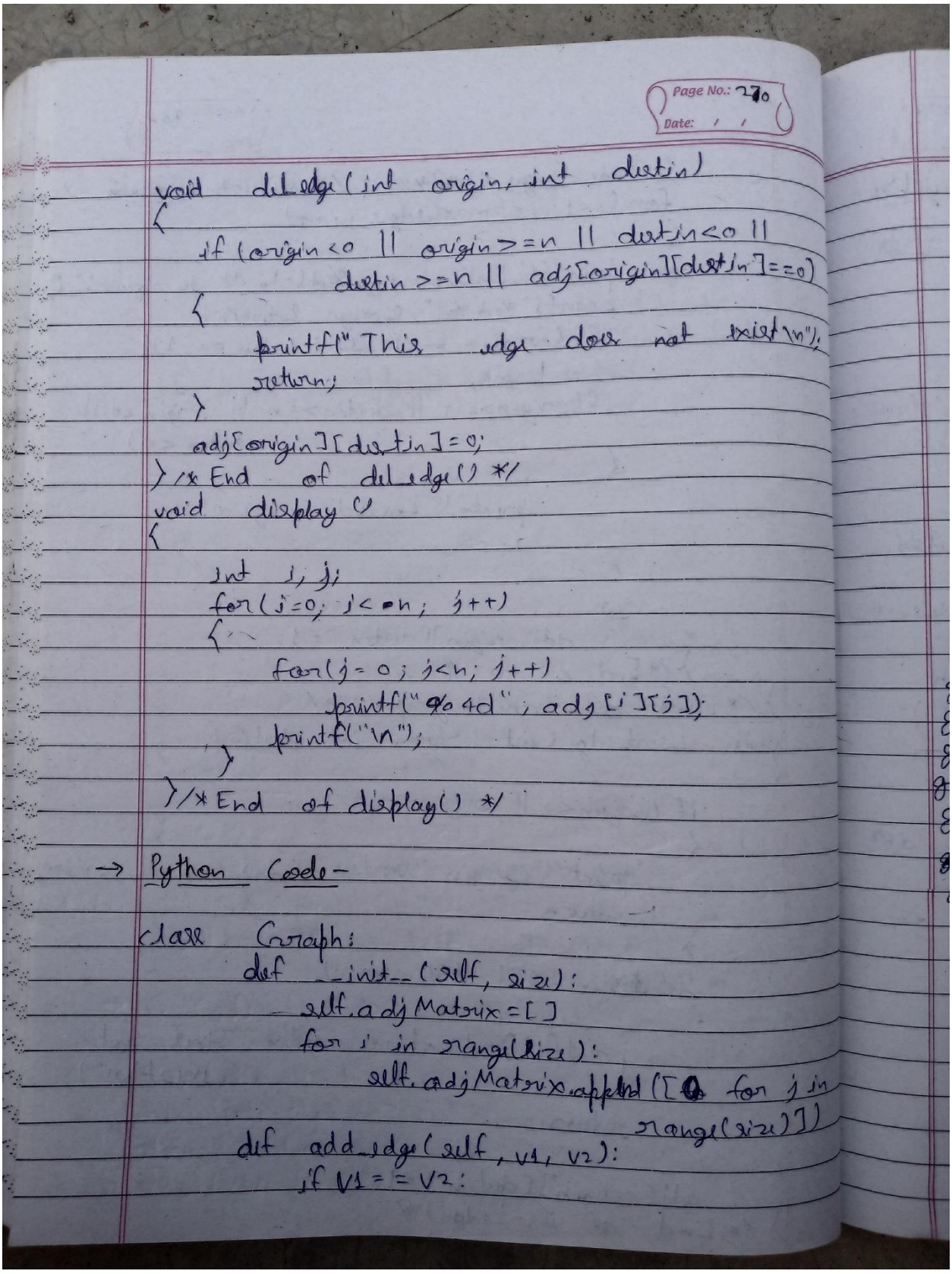
```
#include<stdio.h>
#include<conio.h>
#define MAX 100
int adj[MAX][MAX];
int n;
void create_graph();
void display();
void insert_edge(int origin, int destin);
void del_edge(int origin, int destin);
int main()
{
    int choice, origin, destin;
    create_graph();
    while(1)
    {
        printf("1. Insert an edge\n");
        printf("2. Delete an edge\n");
        printf("3. Display\n");
        printf("4. Exit\n");
        printf("Enter your choice:");
        scanf("%d", &choice);
        switch(choice)
        {
            case 1:
                printf("Enter an edge to be
                        inserted:");
                scanf("%d%d", &origin, &destin);
```



```
                insert_edge(origin,destin);
                break;
            case 2:
                printf("Enter an edge to be
                                deleted:");
                scanf("%d %d", &origin, &destin);
                del_edge(origin,destin);
                break;
            case 3:
                display();
                break;
            case 4:
                exit(1);
            default:
                printf("Wrong choice\n");
                break;
        } /* End of switch */
    } /* End of while */
} /* End of main() */
void create_graph()
{
    int i, max_edges, origin, destin;
    int graph_type;
    printf("Enter 1 For undirected graph or
            2 for directed graph:");
    scanf("%d", &graph_type);
    printf("Enter number of vertices:");
    scanf("%d", &n);
    if (graph_type == 1)
        max_edges = n*(n-1)/2; /* Undirected graph*/
    else
```



```
Page No.: 269
Date:  /  /

        max_edges = n*(n-1); /* directed graph */
    for (i=1; i<=max_edges ; i++)
    {
        printf("Enter edge %d(-1,-1) to quit:",i);
        scanf("%d %d", &origin, &destin);
        if ((origin ==-1) && (destin ==-1))
            break;
        if(origin>=n || destin>=n || origin<0 ||
                                    destin<0)
        {
            printf(" Invalid edge\n");
            i--;
        }
        else
            adj[origin][destin] = 1;
    }/*End of for */
}/*End of create_graph() */
void insert_edge(int origin, int destin)
{
    if (origin<0 || origin>=n)
    {
        printf("Origin vertex does not exist\n");
        return;
    }
    if(destin<0 || destin>=n)
    {
        printf("Destination vertex does not
                                    exist\n");
        return;
    }
    adj[origin][destin]=1;
}/*End of insert_edge() */
```

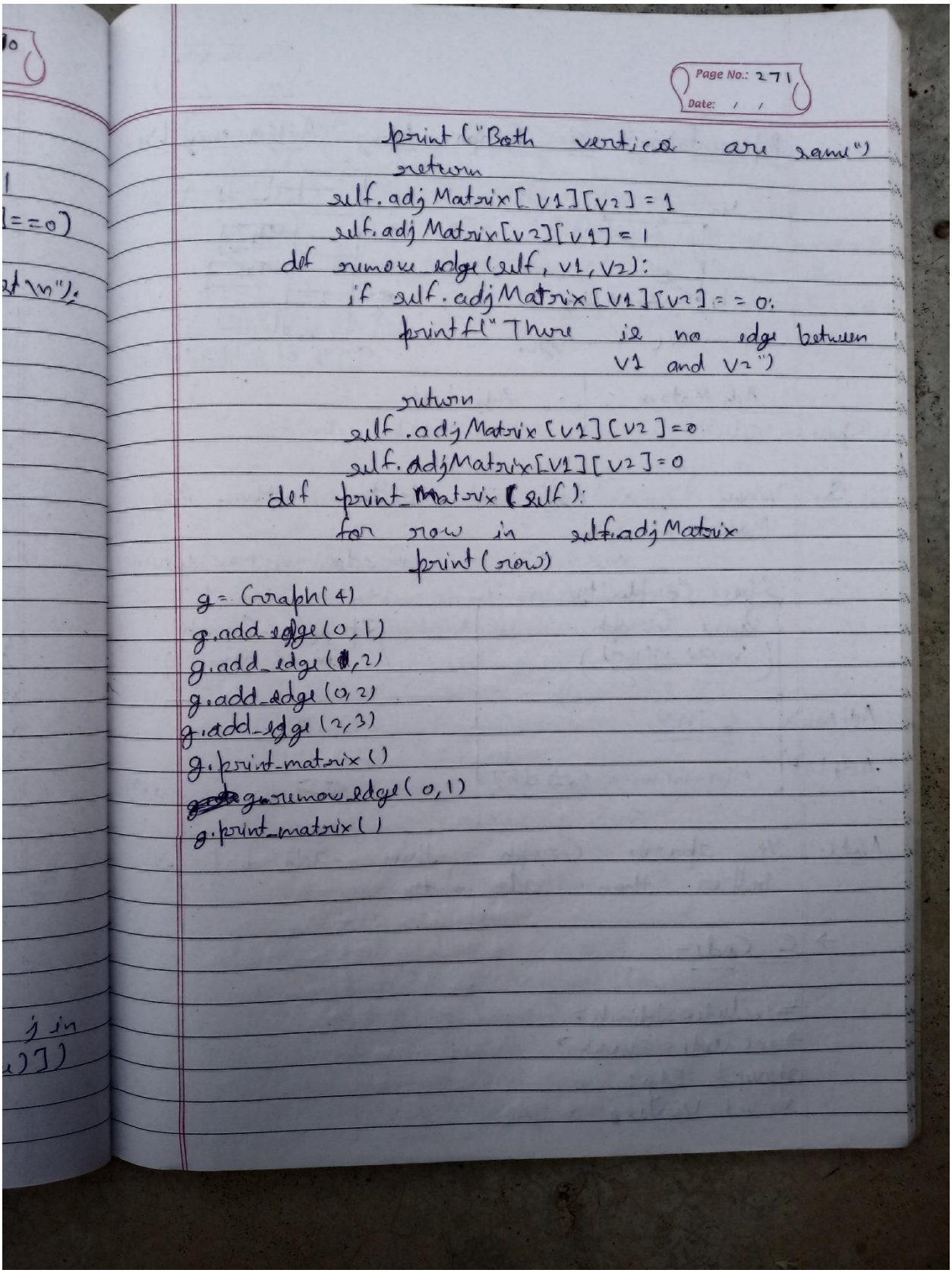


Page No.: 270
Date: / /

```
void delEdge(int origin, int destin)
{
    if(origin<0 || origin>=n || destin<0 ||
            destin>=n || adj[origin][destin]==0)
    {
        printf("This edge does not exist\n");
        return;
    }
    adj[origin][destin]=0;
}/* End of delEdge() */
void display()
{
    int i,j;
    for(i=0; i<n; i++)
    {
        for(j=0; j<n; j++)
            printf("%4d", adj[i][j]);
        printf("\n");
    }
}/* End of display() */
```

→ Python Code-

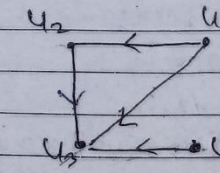
```
class Graph:
    def __init__(self, size):
        self.adjMatrix=[]
        for i in range(size):
            self.adjMatrix.append([0 for j in
                                   range(size)])
    def add_edge(self, v1, v2):
        if v1==v2:
```



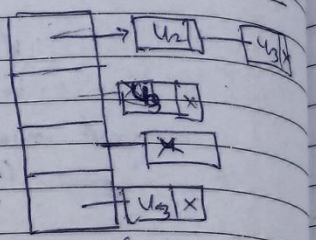
```
                print("Both vertices are same")
                return
            self.adjMatrix[v1][v2] = 1
            self.adjMatrix[v2][v1] = 1
        def remove_edge(self, v1, v2):
            if self.adjMatrix[v1][v2] == 0:
                print("There is no edge between
                                    v1 and v2")
                return
            self.adjMatrix[v1][v2] = 0
            self.adjMatrix[v1][v2] = 0
        def print_matrix(self):
            for row in self.adjMatrix
                print(row)
g = Graph(4)
g.add_edge(0,1)
g.add_edge(0,2)
g.add_edge(0,2)
g.add_edge(2,3)
g.print_matrix()
g.remove_edge(0,1)
g.print_matrix()
```

Page No.: 272
Date: / /

Representation of Graph - using Adjacency List



n - vertices, m - edges



array of pointer/reference

	Adj Matrix	Adj List
Space	$O(n^2)$	$O(n+m)$

Q- When is Adj List better than Adj Matrix-

$m << n^2$, min edge = 0, max edge = $\frac{n(n-1)}{2}$

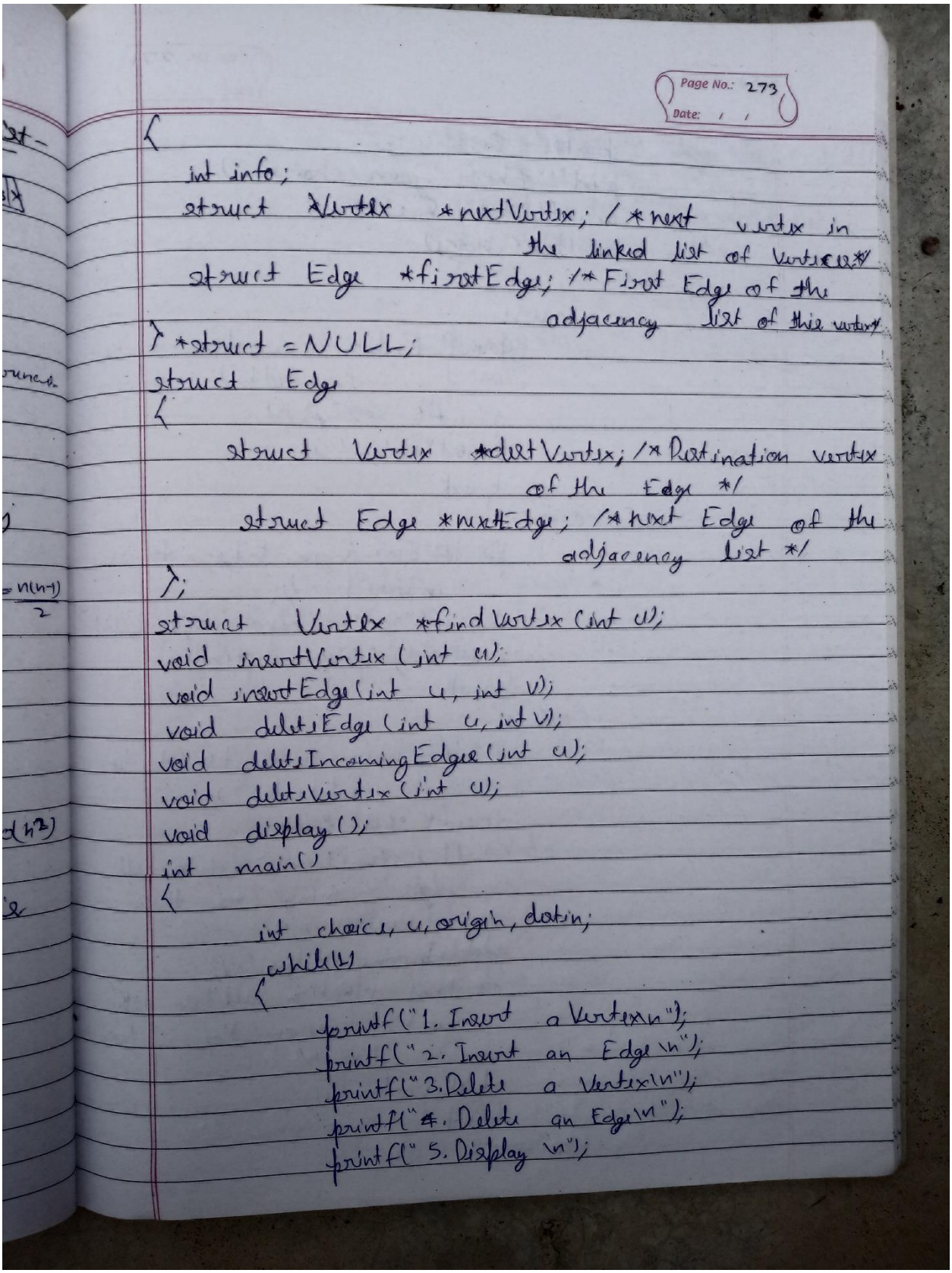
Space Complexity-

	Dense Graph ($m \approx \frac{n(n-1)}{2}$)	Sparse Graph ($m << \frac{n(n-1)}{2}$)
Adj Matrix	$O(n^2)$	$O(n^2)$
Adj List	$O(n+m)$ $m << n^2$ so $\Rightarrow O(n^2)$	less than $O(n^2)$

Note- If sparse Graph then adj-list is better than adj matrix.

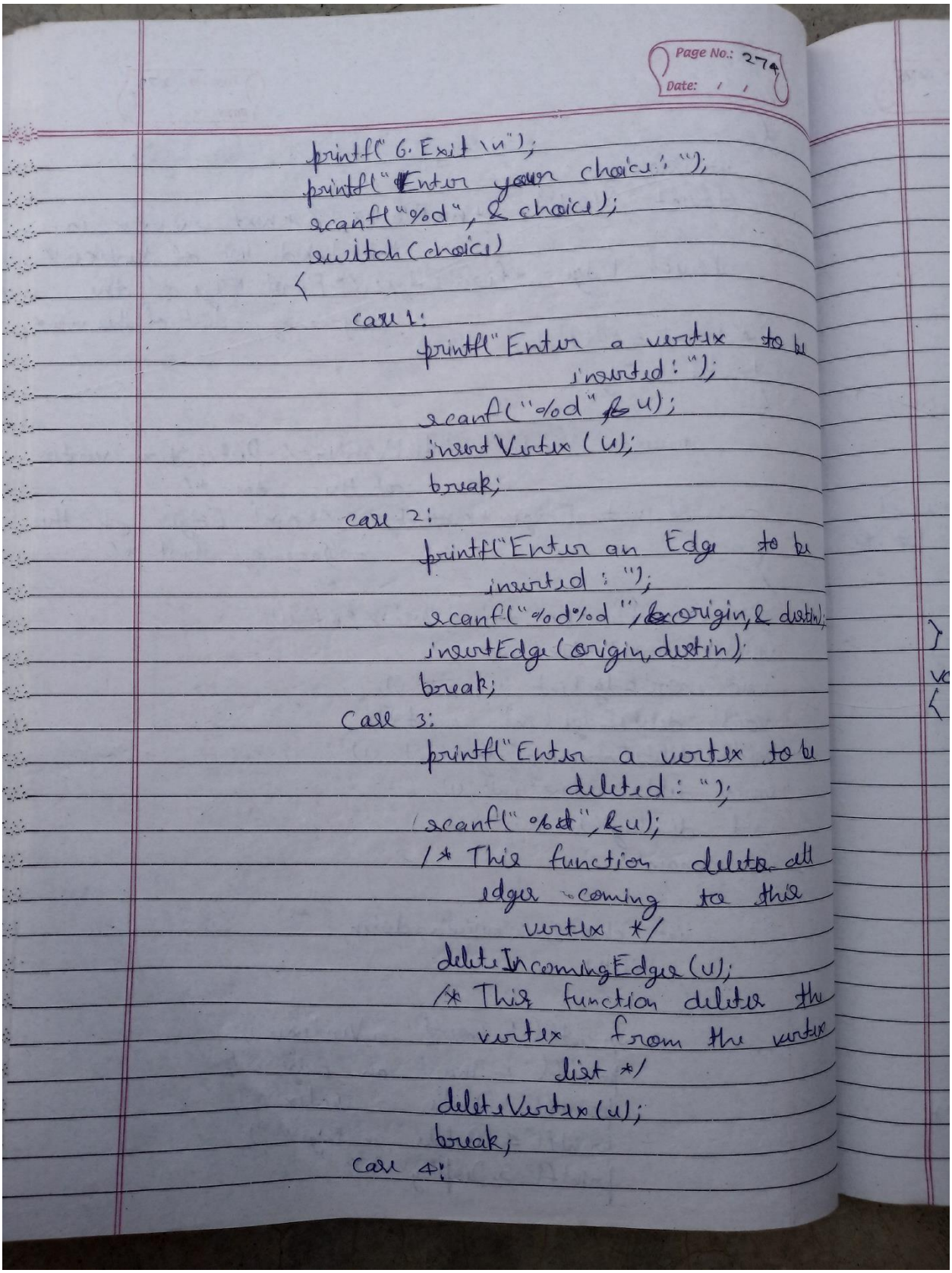
→ C Code-

```
#include<stdio.h>
#include<conio.h>
struct Edge;
struct Vertex;
```

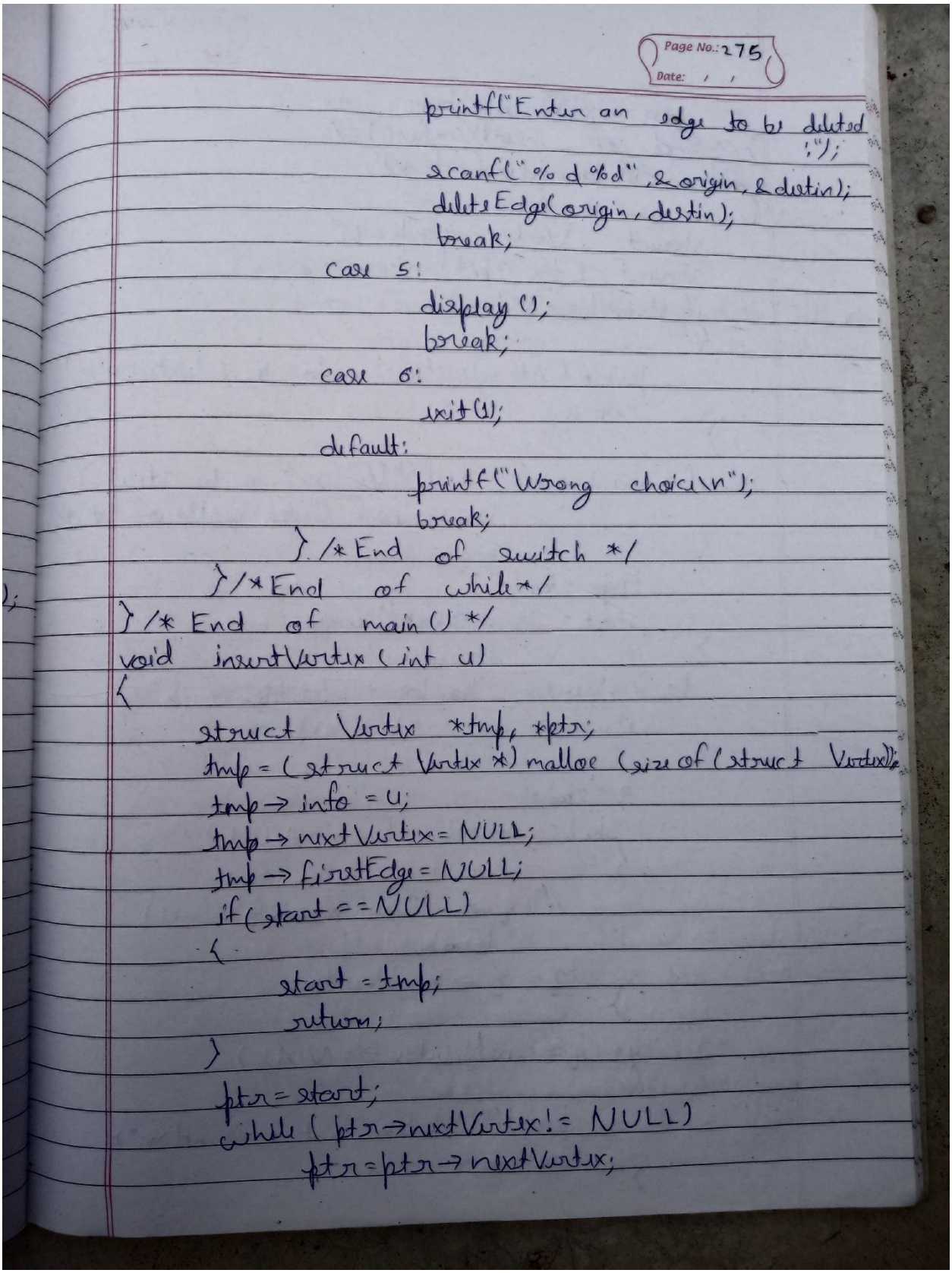


Page No.: 273
Date: / /

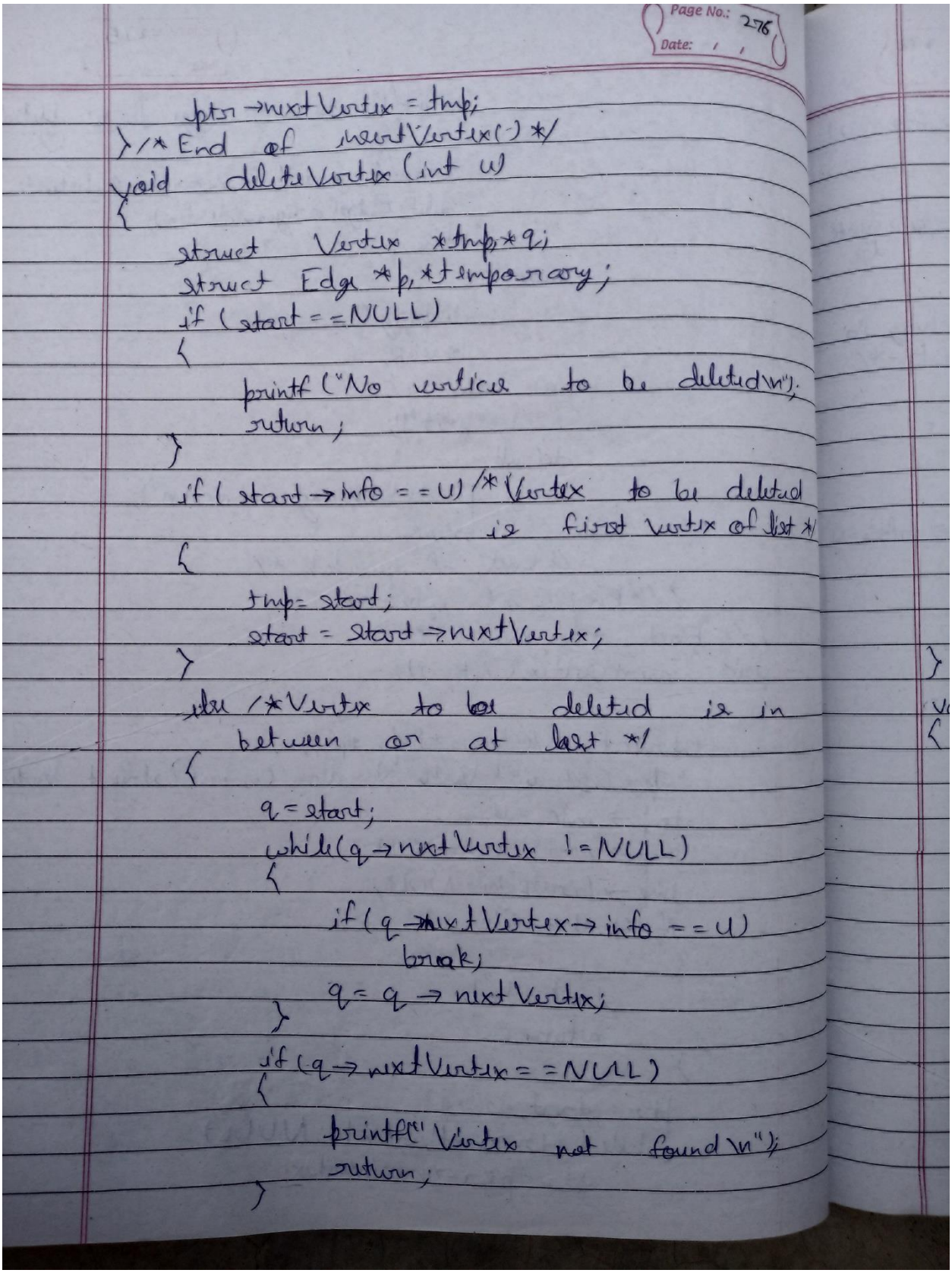
```
{
    int info;
    struct Vertex *nextVertex; /* next vertex in
                                  the linked list of Vertices */
    struct Edge *firstEdge; /* First Edge of the
                               adjacency list of this vertex */
} *struct = NULL;
struct Edge
{
    struct Vertex *destVertex; /* Destination vertex
                                  of the Edge */
    struct Edge *nextEdge; /* next Edge of the
                              adjacency list */
};
struct Vertex *findVertex(int u);
void insertVertex(int u);
void insertEdge(int u, int v);
void deleteEdge(int u, int v);
void deleteIncomingEdges(int u);
void deleteVertex(int u);
void display();
int main()
{
    int choice, u, origin, datin;
    while(1)
    {
        printf("1. Insert a Vertex\n");
        printf("2. Insert an Edge\n");
        printf("3. Delete a Vertex\n");
        printf("4. Delete an Edge\n");
        printf("5. Display\n");
```



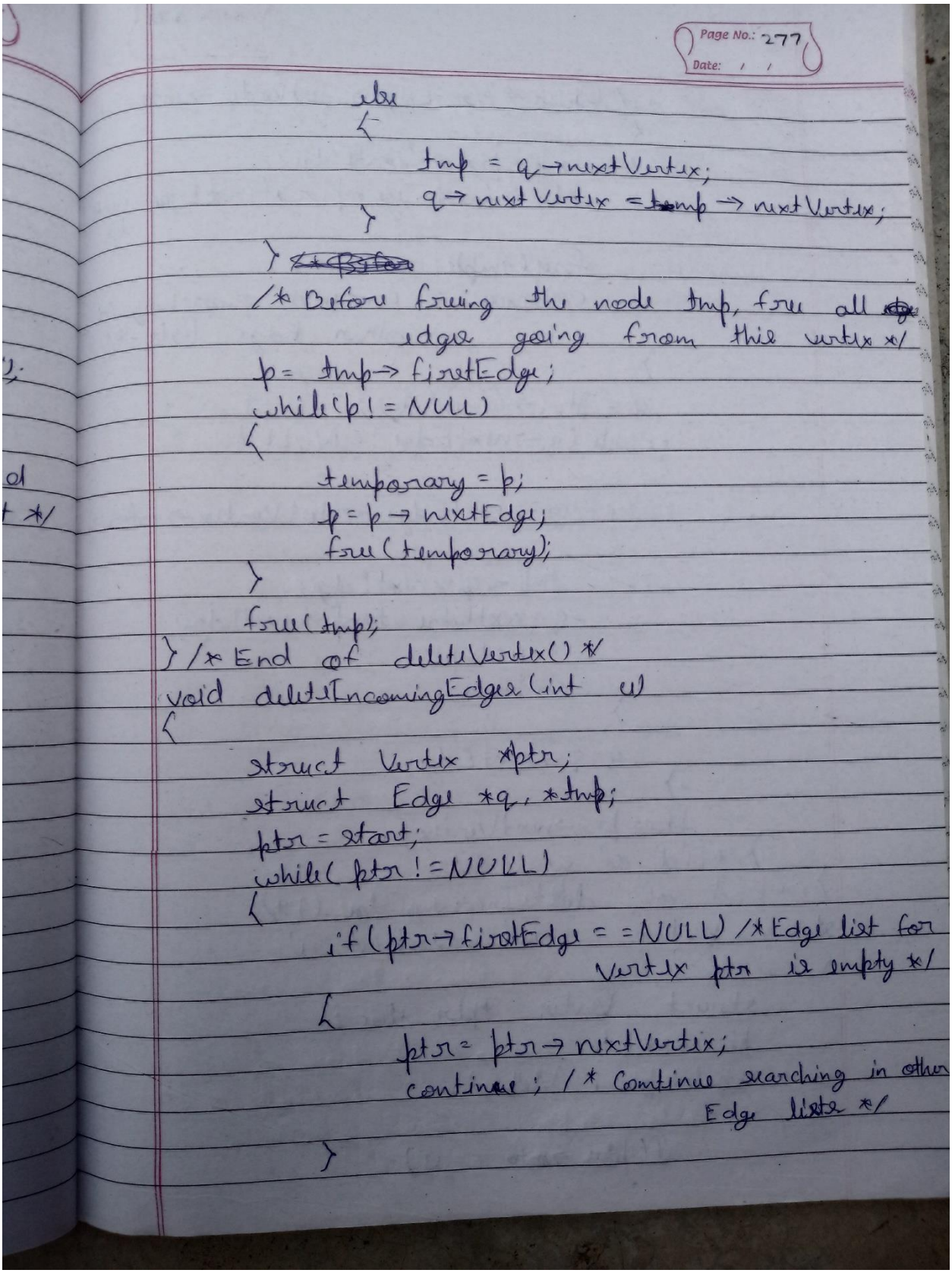
```
printf("6. Exit \n");
printf("Enter your choice: ");
scanf("%d", &choice);
switch(choice)
{
    case 1:
        printf("Enter a vertex to be
                inserted: ");
        scanf("%d", &u);
        insertVertex(u);
        break;
    case 2:
        printf("Enter an Edge to be
                inserted: ");
        scanf("%d%d", &origin, &destin);
        insertEdge(origin, destin);
        break;
    Case 3:
        printf("Enter a vertex to be
                deleted: ");
        scanf("%d", &u);
        /* This function deletes all
           edges coming to this
           vertex */
        deleteIncomingEdge(u);
        /* This function deletes the
           vertex from the vertex
           list */
        deleteVertex(u);
        break;
    Case 4:
```



```
                printf("Enter an edge to be deleted :");
                scanf("%d %d", &origin, &destin);
                deleteEdge(origin, destin);
                break;
        case 5:
                display();
                break;
        case 6:
                exit(1);
        default:
                printf("Wrong choice\n");
                break;
        } /* End of switch */
    } /* End of while */
} /* End of main() */
void insertVertex(int u)
{
    struct Vertex *tmp, *ptr;
    tmp = (struct Vertex *) malloc(sizeof(struct Vertex));
    tmp->info = u;
    tmp->nextVertex = NULL;
    tmp->firstEdge = NULL;
    if(start == NULL)
    {
        start = tmp;
        return;
    }
    ptr = start;
    while(ptr->nextVertex != NULL)
        ptr = ptr->nextVertex;
```

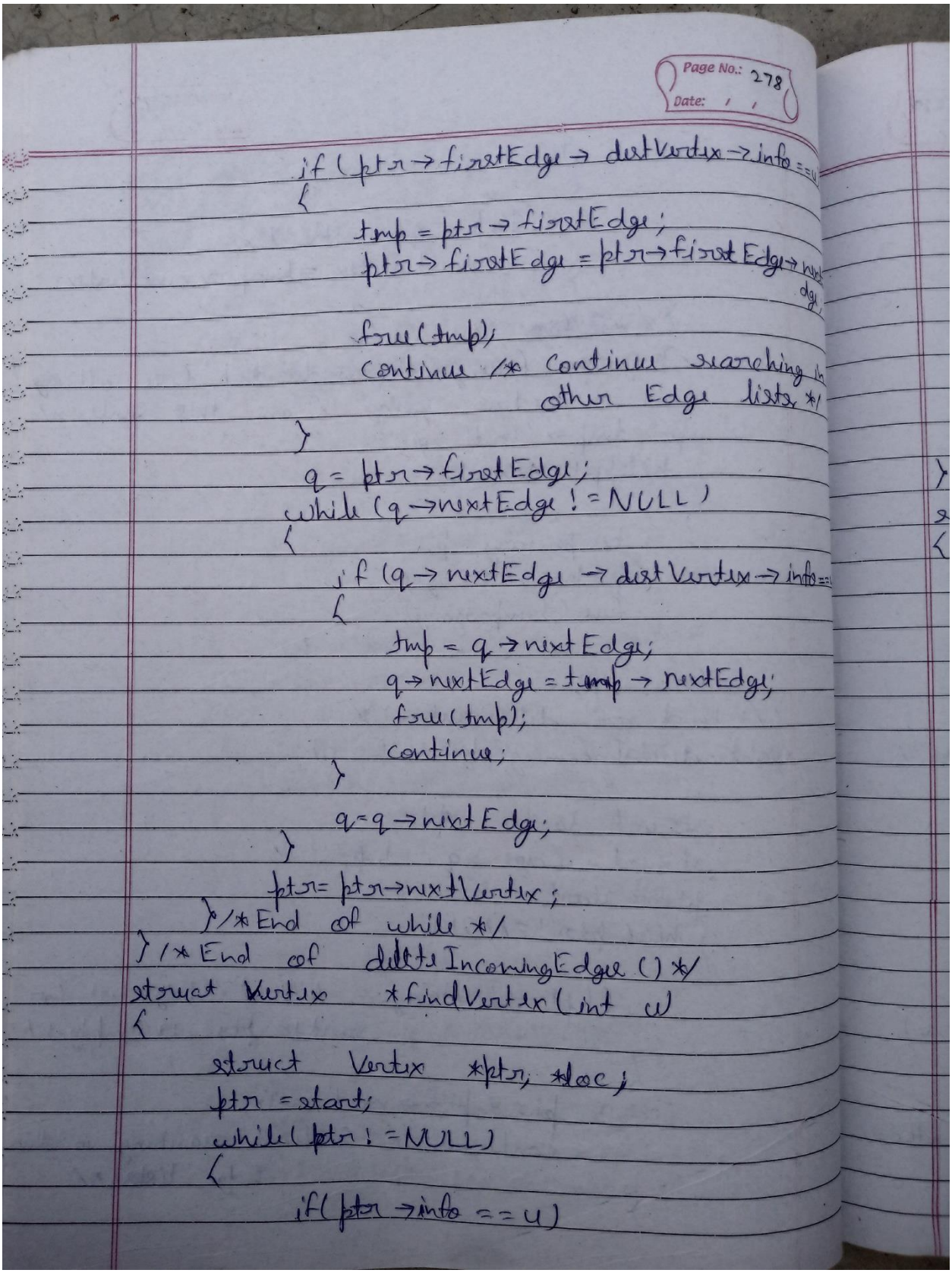


```
        ptr->nextVertex = tmp;
}/* End of insertVertex() */
void deleteVertex(int u)
{
    struct Vertex *tmp,*q;
    struct Edge *p,*temporary;
    if(start==NULL)
    {
        printf("No vertices to be deleted\n");
        return;
    }
    if(start->info==u) /* Vertex to be deleted
                          is first vertex of list */
    {
        tmp=start;
        start=start->nextVertex;
    }
    else /* Vertex to be deleted is in
            between or at last */
    {
        q=start;
        while(q->nextVertex!=NULL)
        {
            if(q->nextVertex->info==u)
                break;
            q=q->nextVertex;
        }
        if(q->nextVertex==NULL)
        {
            printf("Vertex not found\n");
            return;
        }
```

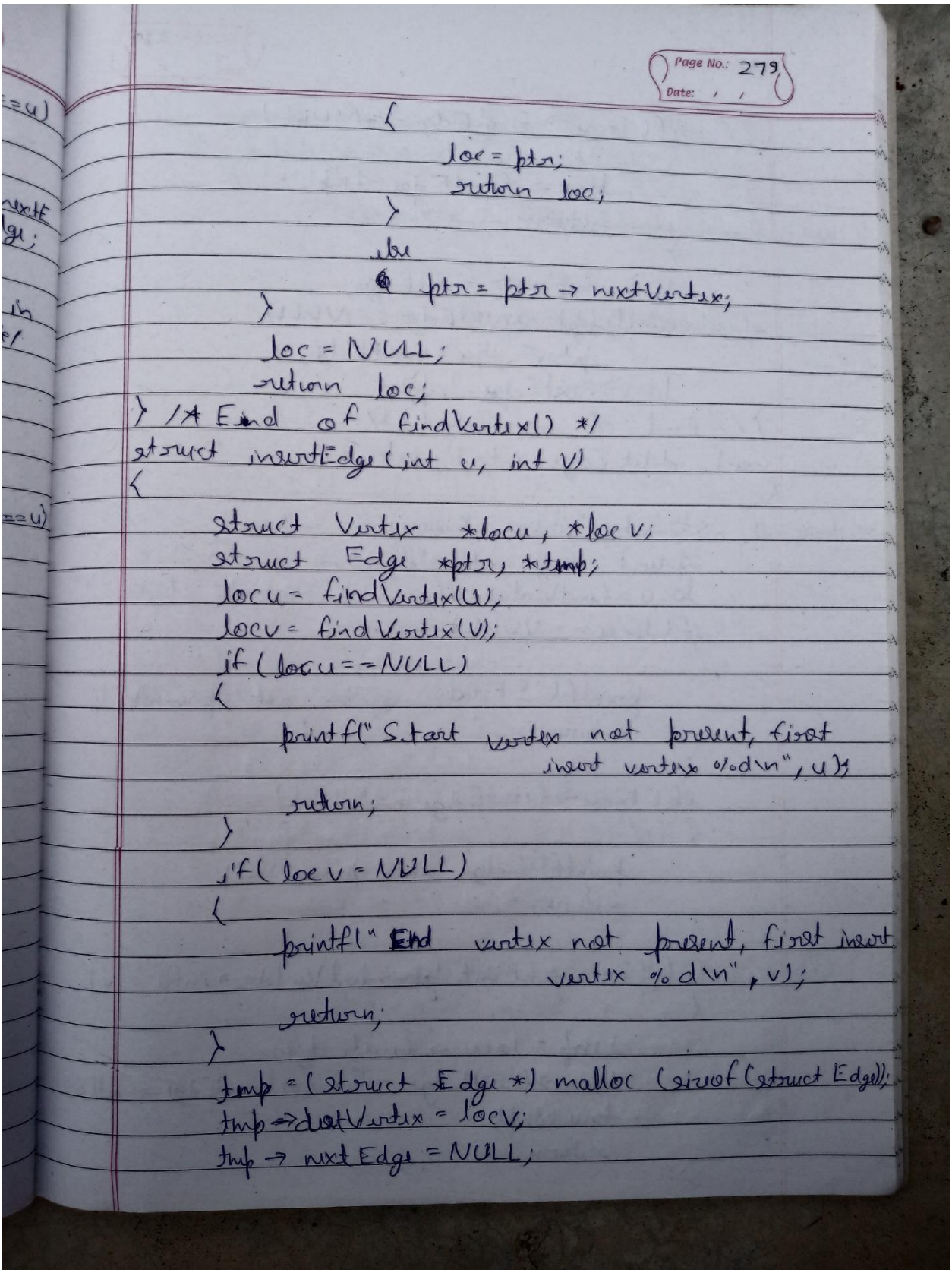


Page No.: 277
Date: / /

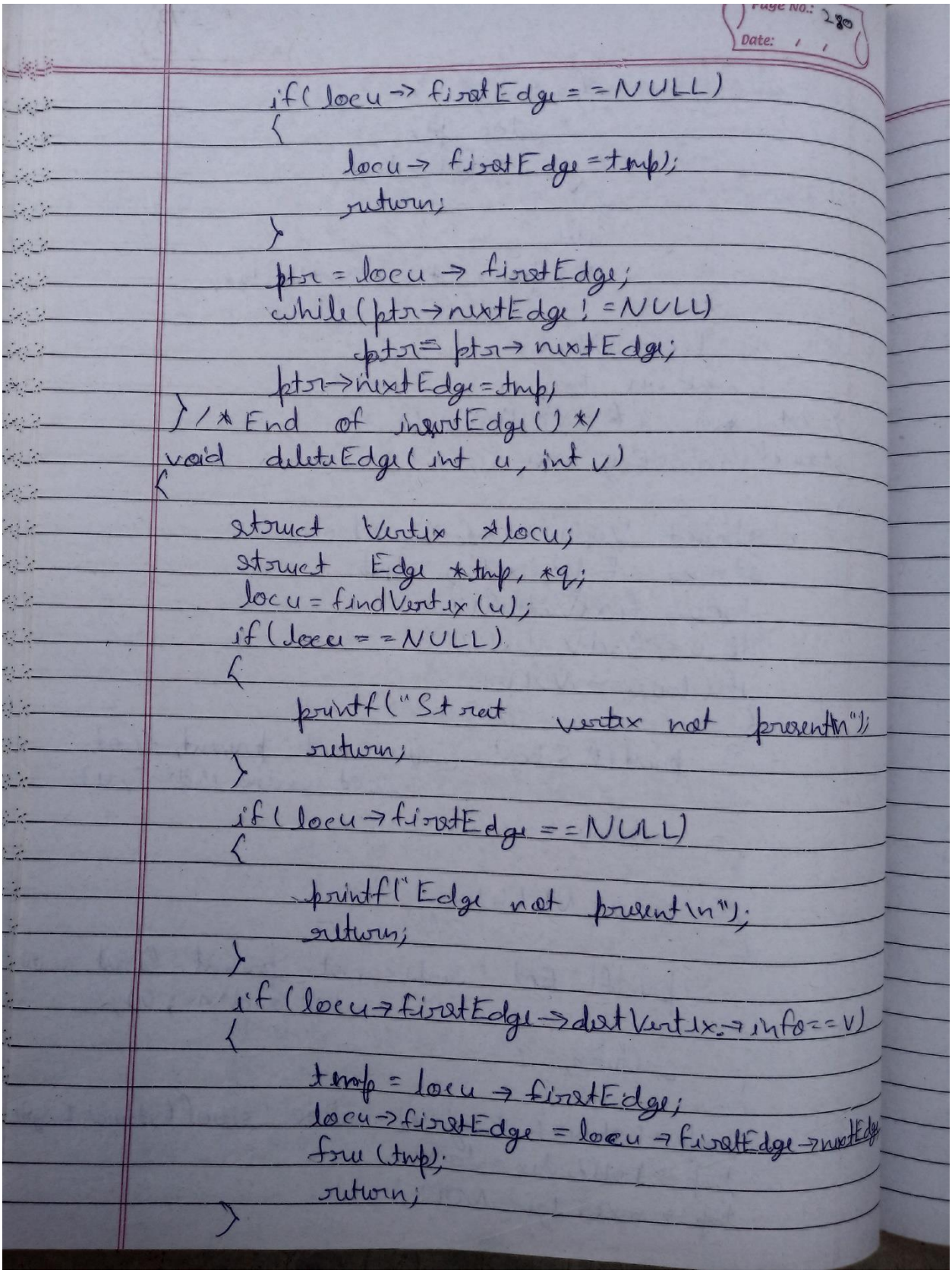
```
        else
        {
            tmp = q->nextVertex;
            q->nextVertex = tmp->nextVertex;
        }
    }
    /* Before freeing the node tmp, free all
       edges going from this vertex */
    p = tmp->firstEdge;
    while(p != NULL)
    {
        temporary = p;
        p = p->nextEdge;
        free(temporary);
    }
    free(tmp);
} /* End of deleteVertex() */
void deleteIncomingEdges(int u)
{
    struct Vertex *ptr;
    struct Edge *q, *tmp;
    ptr = start;
    while(ptr != NULL)
    {
        if(ptr->firstEdge == NULL) /* Edge list for
                                      Vertex ptr is empty */
        {
            ptr = ptr->nextVertex;
            continue; /* Continue searching in other
                         Edge lists */
        }
```



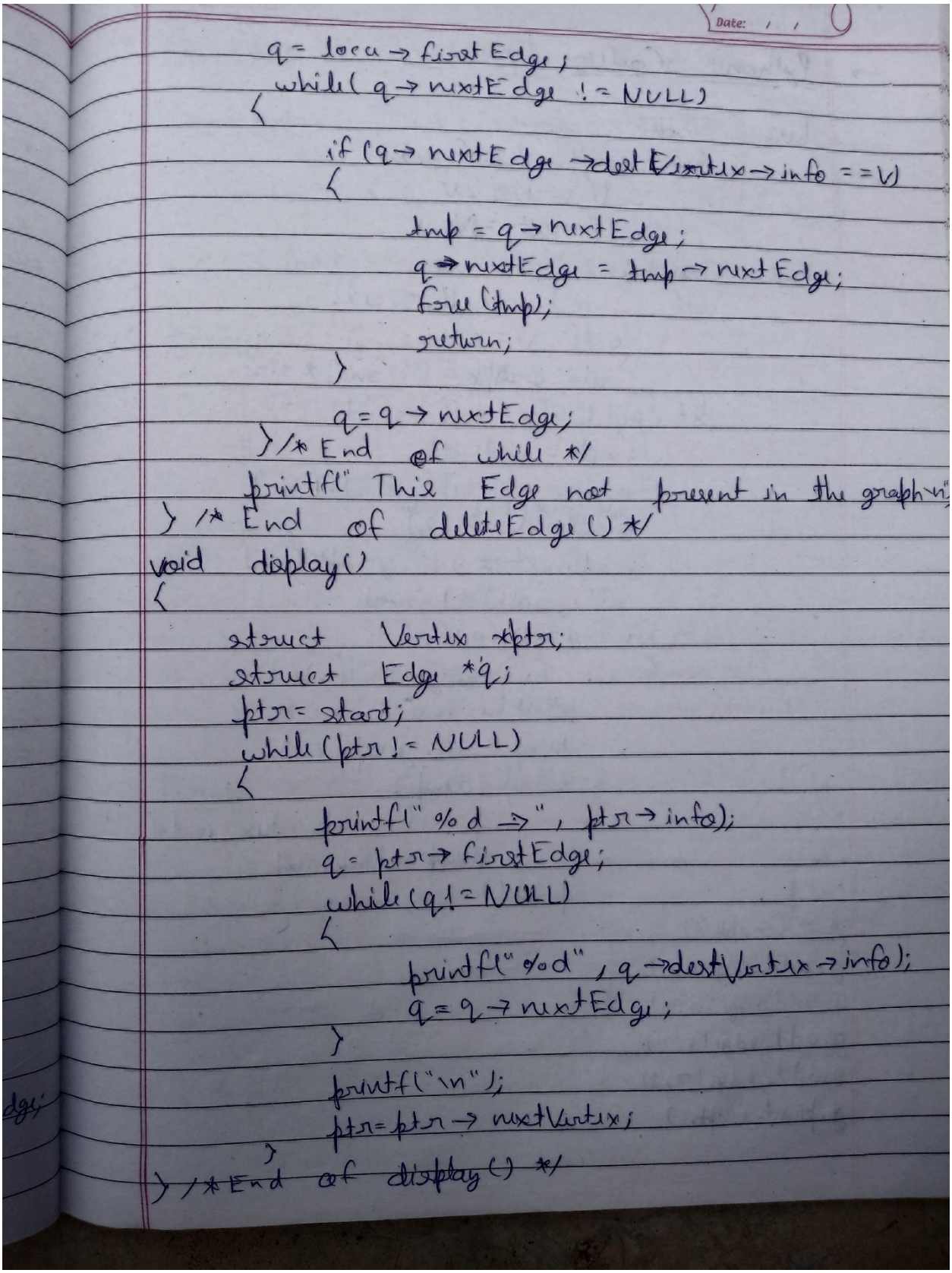
```
Page No.: 278
Date:  /  /

if (ptr->firstEdge->destVertex->info == u)
{
    tmp = ptr->firstEdge;
    ptr->firstEdge = ptr->firstEdge->nextEdge;
    free(tmp);
    continue /* continue searching in other Edge lists */
}
q = ptr->firstEdge;
while (q->nextEdge != NULL)
{
    if (q->nextEdge->destVertex->info == u)
    {
        tmp = q->nextEdge;
        q->nextEdge = tmp->nextEdge;
        free(tmp);
        continue;
    }
    q = q->nextEdge;
}
ptr = ptr->nextVertex;
}/* End of while */
}/* End of deleteIncomingEdge() */
struct Vertex *findVertex(int u)
{
    struct Vertex *ptr, *loc;
    ptr = start;
    while(ptr != NULL)
    {
        if(ptr->info == u)
```



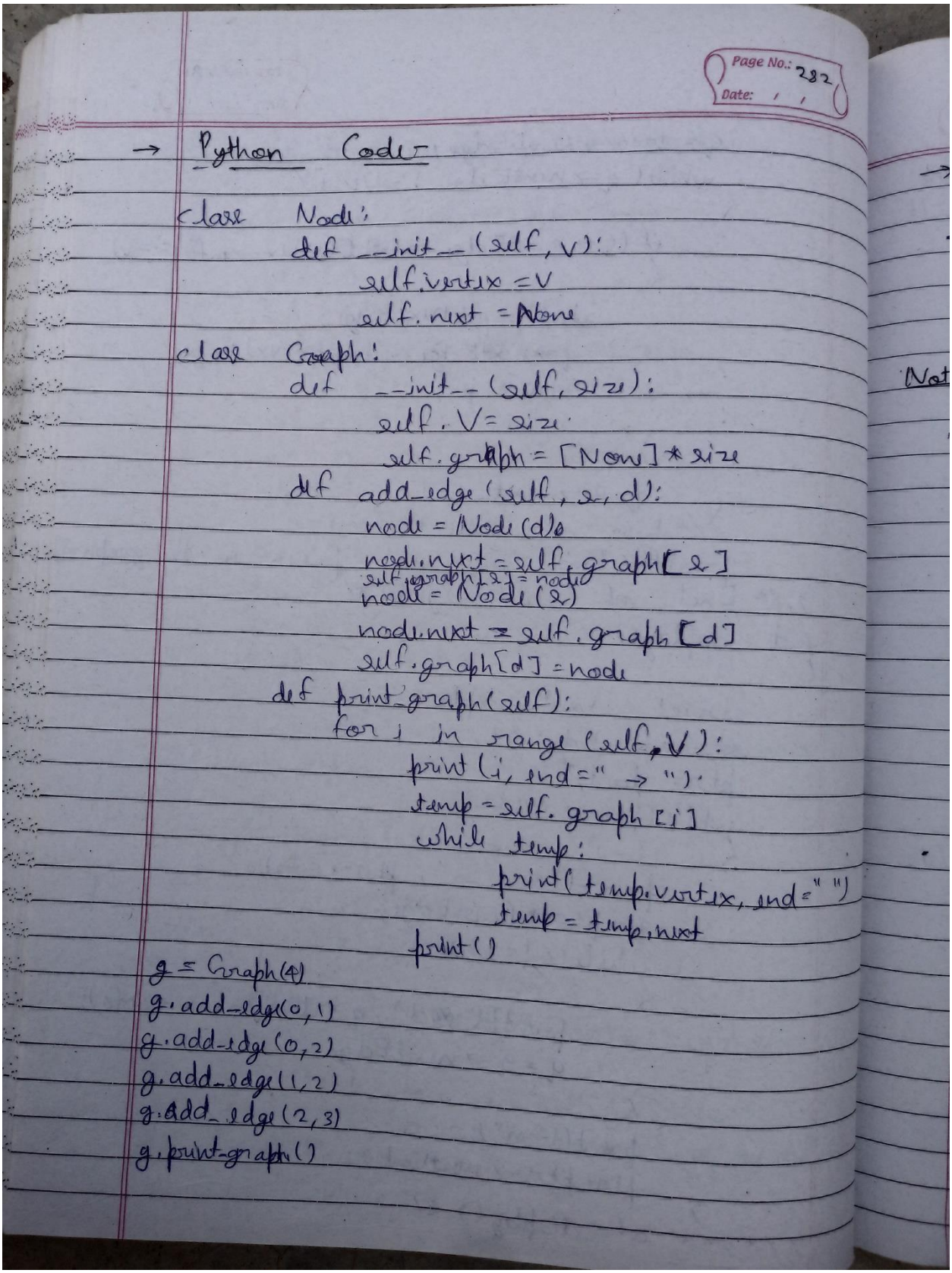
```
                {
                    loc = ptr;
                    return loc;
                }
                else
                    ptr = ptr -> nextVertex;
            }
            loc = NULL;
            return loc;
} /* End of findVertex() */
struct insertEdge (int u, int v)
{
    struct Vertex *locu, *locv;
    struct Edge *ptr, *tmp;
    locu = findVertex(u);
    locv = findVertex(v);
    if (locu == NULL)
    {
        printf("Start vertex not present, first
                insert vertex %d\n", u);
        return;
    }
    if( locv = NULL)
    {
        printf("End vertex not present, first insert
                vertex %d\n", v);
        return;
    }
    tmp = (struct Edge *) malloc (sizeof(struct Edge));
    tmp->destVertex = locv;
    tmp -> nextEdge = NULL;
```



```
    if(locu->firstEdge==NULL)
    {
        locu->firstEdge=tmp;
        return;
    }
    ptr=locu->firstEdge;
    while(ptr->nextEdge!=NULL)
        ptr=ptr->nextEdge;
    ptr->nextEdge=tmp;
}/* End of insertEdge() */
void deleteEdge(int u, int v)
{
    struct Vertex *locu;
    struct Edge *tmp, *q;
    locu=findVertex(u);
    if(locu==NULL)
    {
        printf("Start vertex not present\n");
        return;
    }
    if(locu->firstEdge==NULL)
    {
        printf("Edge not present\n");
        return;
    }
    if(locu->firstEdge->destVertex->info==v)
    {
        tmp=locu->firstEdge;
        locu->firstEdge=locu->firstEdge->nextEdge;
        free(tmp);
        return;
    }
```



```
Date: / /

    q = loca -> firstEdge;
    while( q -> nextEdge != NULL)
    {
        if (q -> nextEdge -> destVertex -> info == v)
        {
            tmp = q -> nextEdge;
            q -> nextEdge = tmp -> nextEdge;
            free (tmp);
            return;
        }
        q = q -> nextEdge;
    }/* End of while */
    printf" This Edge not present in the graph\n";
} /* End of deleteEdge() */
void display()
{
    struct Vertex *ptr;
    struct Edge *q;
    ptr = start;
    while (ptr != NULL)
    {
        printf" %d ->", ptr -> info);
        q = ptr -> firstEdge;
        while (q != NULL)
        {
            printf" %d", q -> destVertex -> info);
            q = q -> nextEdge;
        }
        printf("\n");
        ptr = ptr -> nextVertex;
    }
} /* End of display() */
```



→ Python Code:

```
class Node:
    def __init__(self, v):
        self.vertex = v
        self.next = None
class Graph:
    def __init__(self, size):
        self.V = size
        self.graph = [None] * size
    def add_edge(self, s, d):
        node = Node(d)
        node.next = self.graph[s]
        self.graph[s] = node
        node = Node(s)
        node.next = self.graph[d]
        self.graph[d] = node
    def print_graph(self):
        for i in range(self.V):
            print(i, end=" → ")
            temp = self.graph[i]
            while temp:
                print(temp.vertex, end=" ")
                temp = temp.next
            print()
g = Graph(4)
g.add_edge(0, 1)
g.add_edge(0, 2)
g.add_edge(1, 2)
g.add_edge(2, 3)
g.print_graph()
```

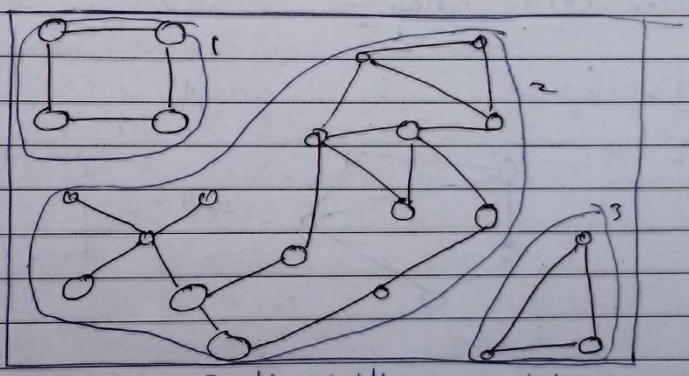
Page No.: 283
Date: 19/11/20

→ Connectivity in Undirected Graph-

- Connected- There exists a path from every vertex to every other vertex $\{ u_i \rightsquigarrow u_j \ \forall \ i, j \quad u_i \rightarrow u_j \}$

Note- $\rightsquigarrow$ (Path) and $\rightarrow$ (Edge)

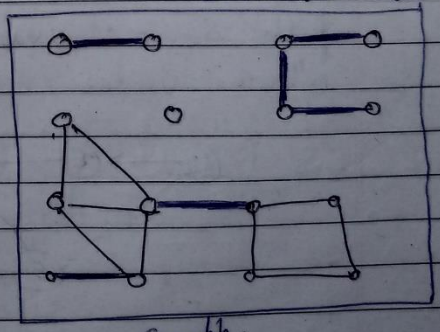
- Connected Component-



Graph with 3 connected component

- Bridge (Cut Edge or Cut arc)- Edge whose removal results in increase in the number of connected components a graph

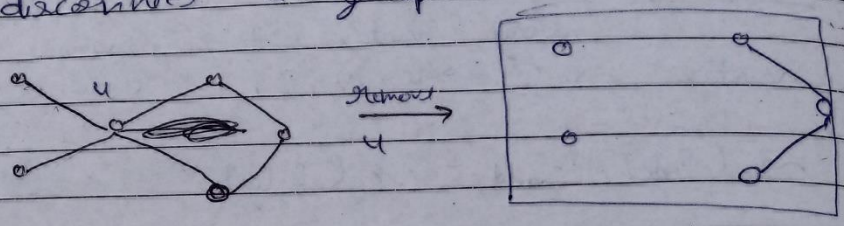
Note- — bridge



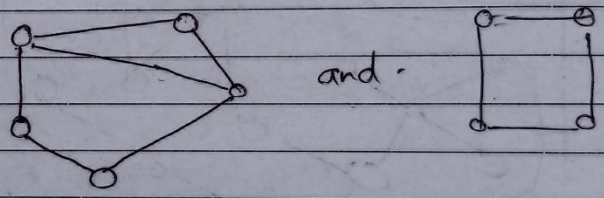
Graph.

Page No.: 284
Date: / /

- **Articulation Point (cut - vertex)** - Vertex whose removal will disconnect a graph.

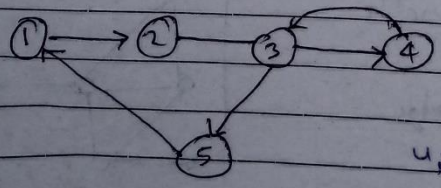


- **Biconnected Graph** - removal of one vertex does not disconnect the graph.



→ **Connectivity in Directed Graphs** -

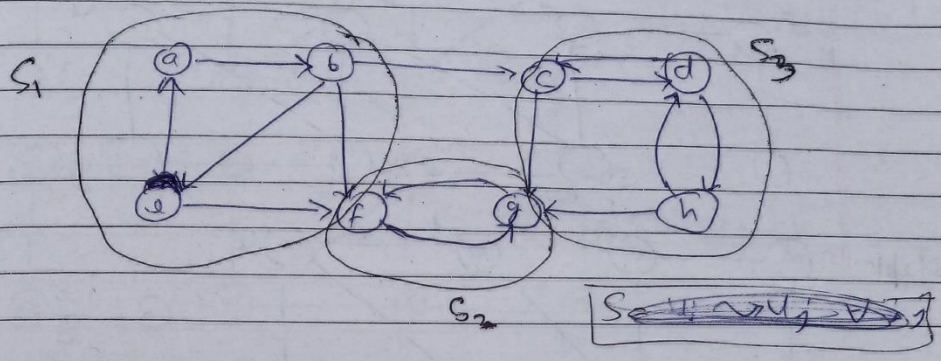
- **Strogly connected di-graph** - There exists a directed path from every vertex u_i to every other vertex u_j in G.



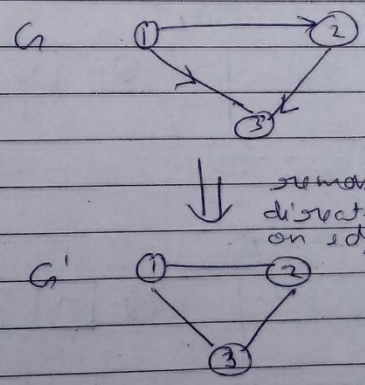
$u_i \rightarrow u_j \quad \forall\ i, j$

Page No.: 285
Date: / /

- **Strongly Connected Component-**



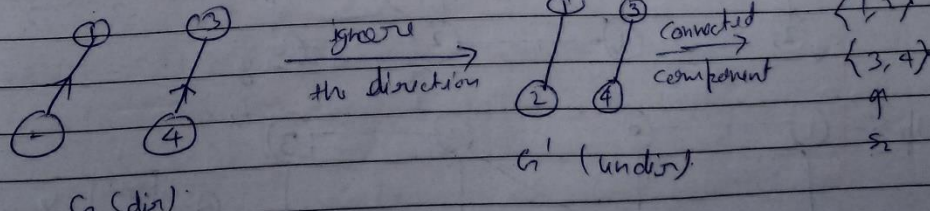
- **Weekly Connected di-graph-**

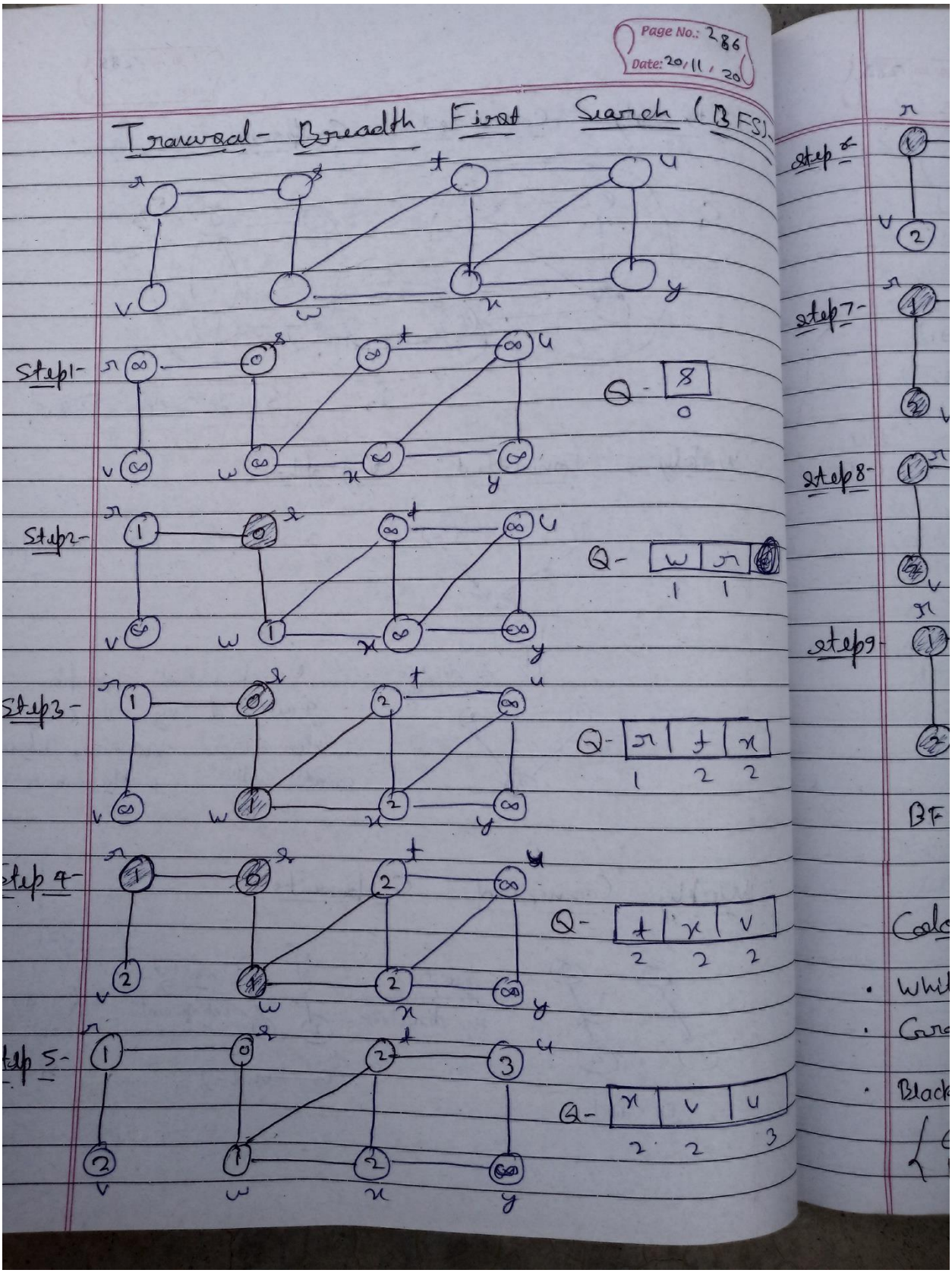


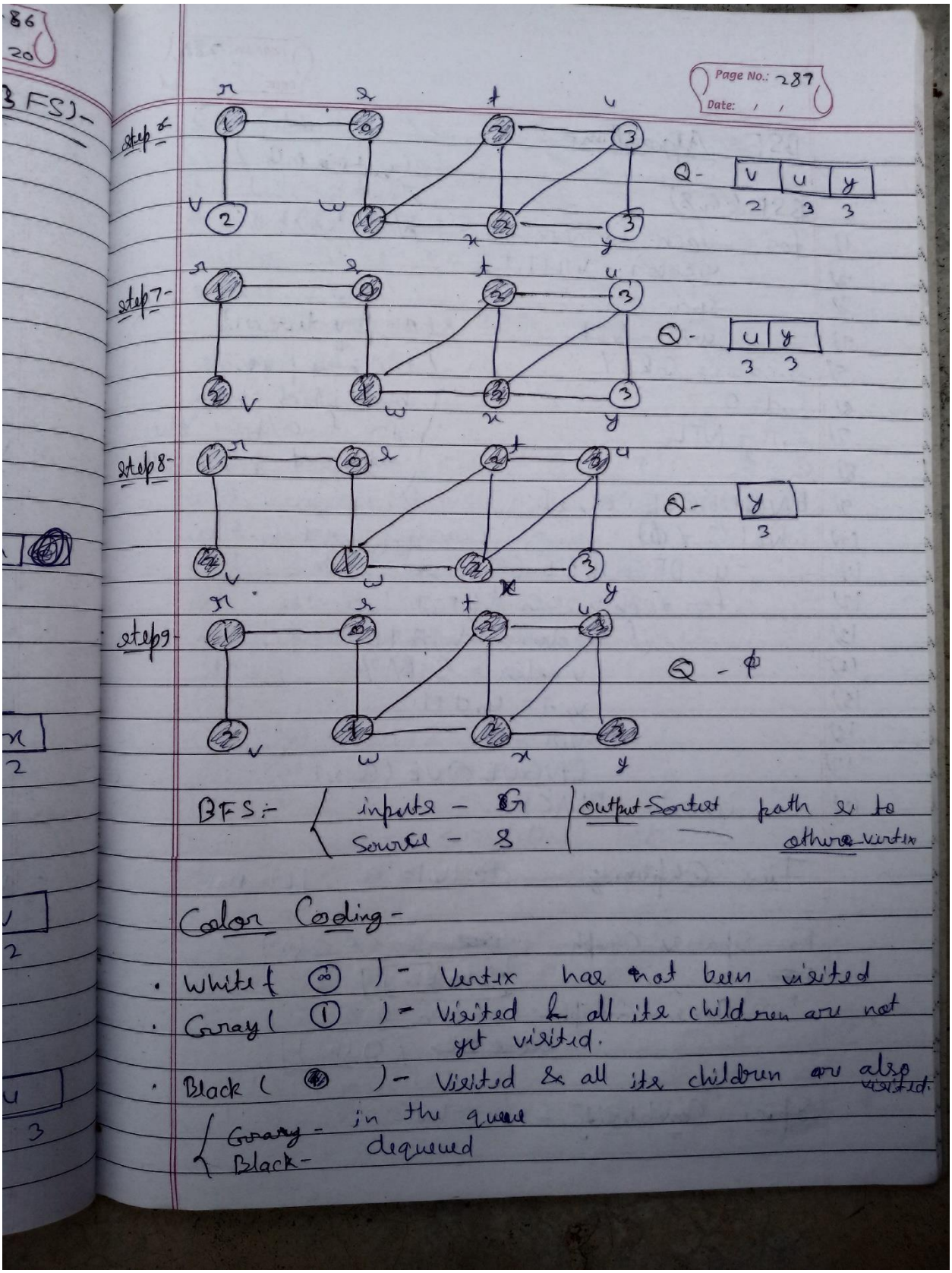
3 → 1 ⊗ Not directed path

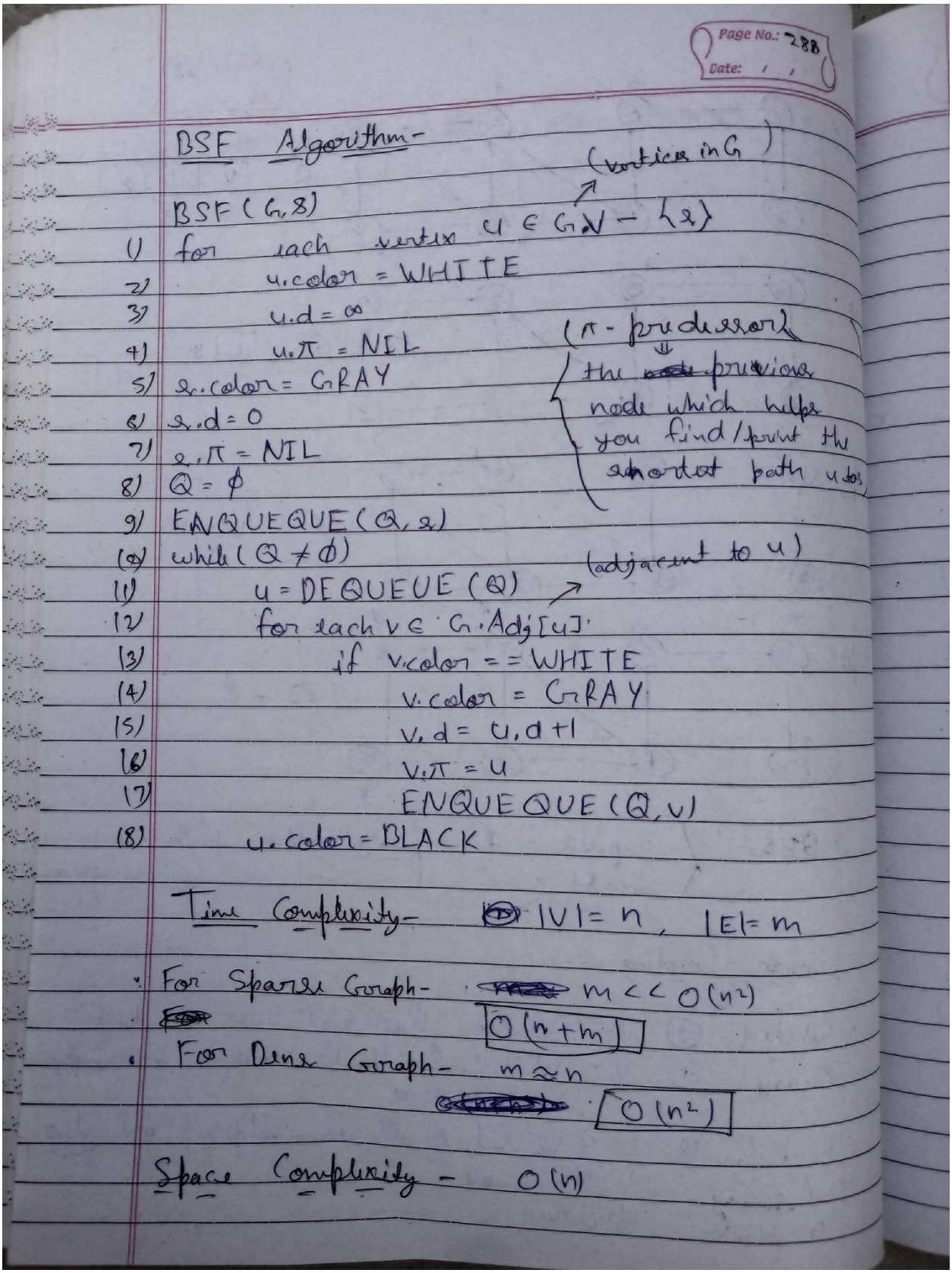
Un-directed graph generated by removing direction on my edges called weakly connected graph.

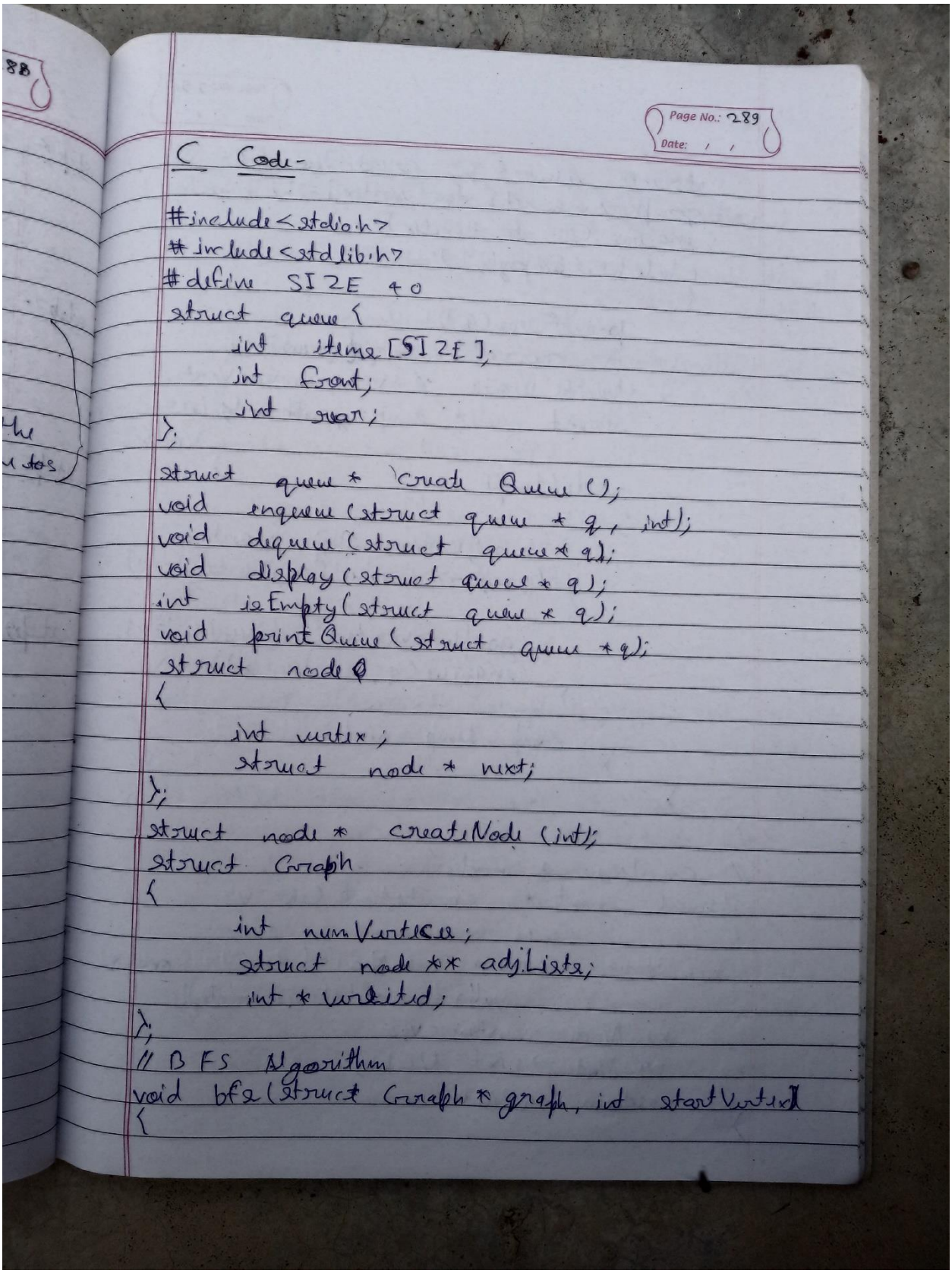
- **Weekly Connected Components-**




Page No.: 286
Date: 20/11/20
Traversal- Breadth First Search (BFS)
Step1-
Step2-
Step3-
Step 4-
Step 5-
step 6-
step 7-
step 8-
step 9-
Q- s
Q- w r
Q- r t x
Q- t x v
Q- x v u


Page No.: 287
Date: / /
step 6-
step 7-
step 8-
step 9-
Q- v u y
2 3 3
Q- u y
3 3
Q- y
3
Q - ϕ
BFS:- { input - G, source - s | output - Sortest path s to other vertex
Color Coding-
• White (∞) - Vertex has not been visited
• Gray (1) - Visited & all its children are not yet visited.
• Black ( ) - Visited & all its children are also visited.
{ Gray - in the queue
Black - dequeued

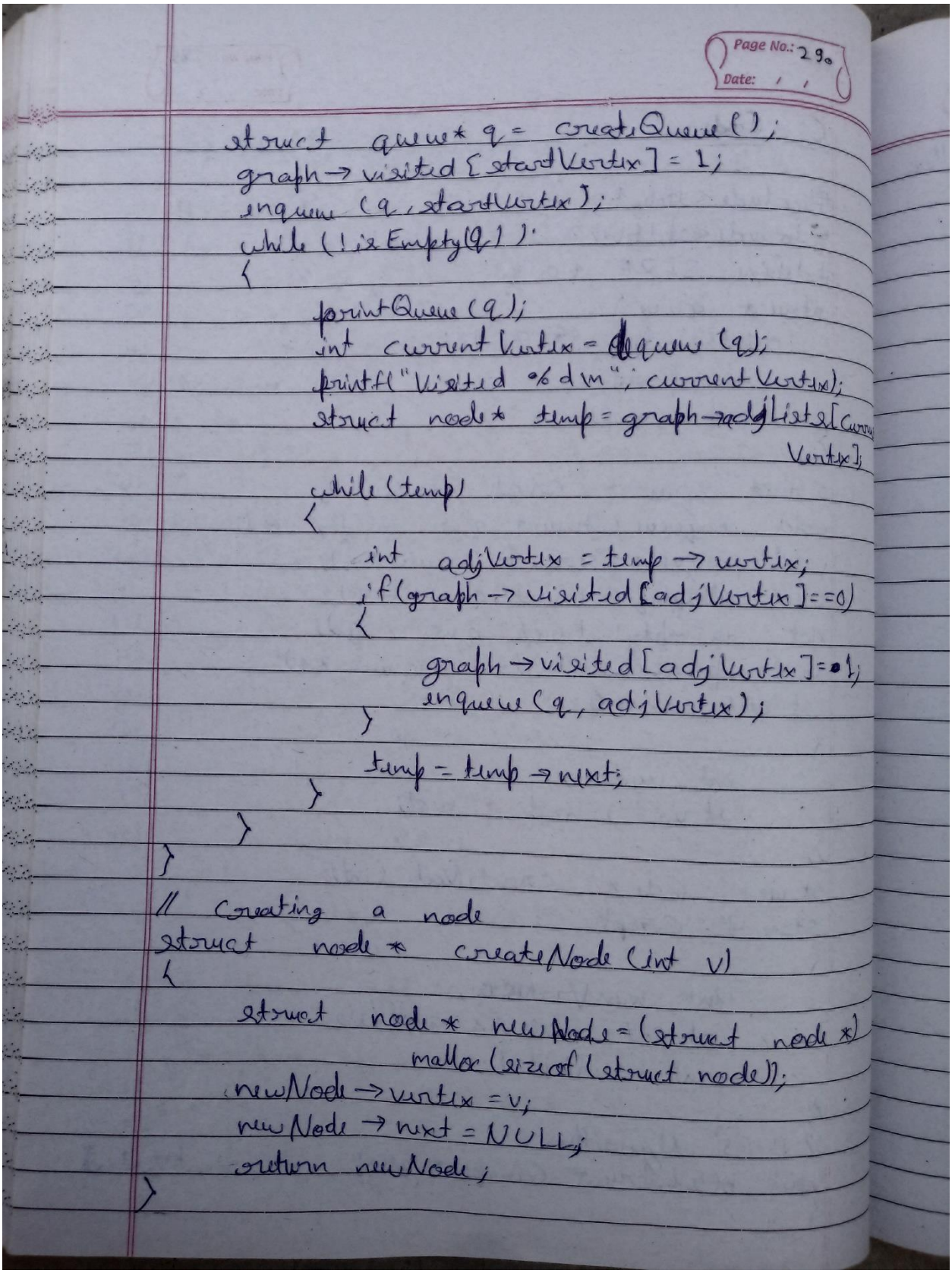

Page No.: 288
Date: / /
BSF Algorithm-
BSF(G,s)
(vertices in G)
1) for each vertex u ∈ G.V - {s}
2) u.color = WHITE
3) u.d = ∞
4) u.π = NIL
(π - predecessor)
the previous node which helps you find/print the shortest path u to s
5) s.color = GRAY
6) s.d = 0
7) s.π = NIL
8) Q = ϕ
9) ENQUEQUE(Q,s)
10) while(Q ≠ ϕ)
(adjacent to u)
11) u = DEQUEUE(Q)
12) for each v ∈ G.Adj[u]
13) if v.color == WHITE
14) v.color = GRAY
15) v.d = u.d + 1
16) v.π = u
17) ENQUEQUE(Q,v)
18) u.color = BLACK
Time Complexity- |V| = n, |E| = m
For Sparse Graph- m << O(n²)
O(n+m)
For Dense Graph- m ≈ n
O(n²)
Space Complexity - O(n)



Page No.: 289
Date: / /

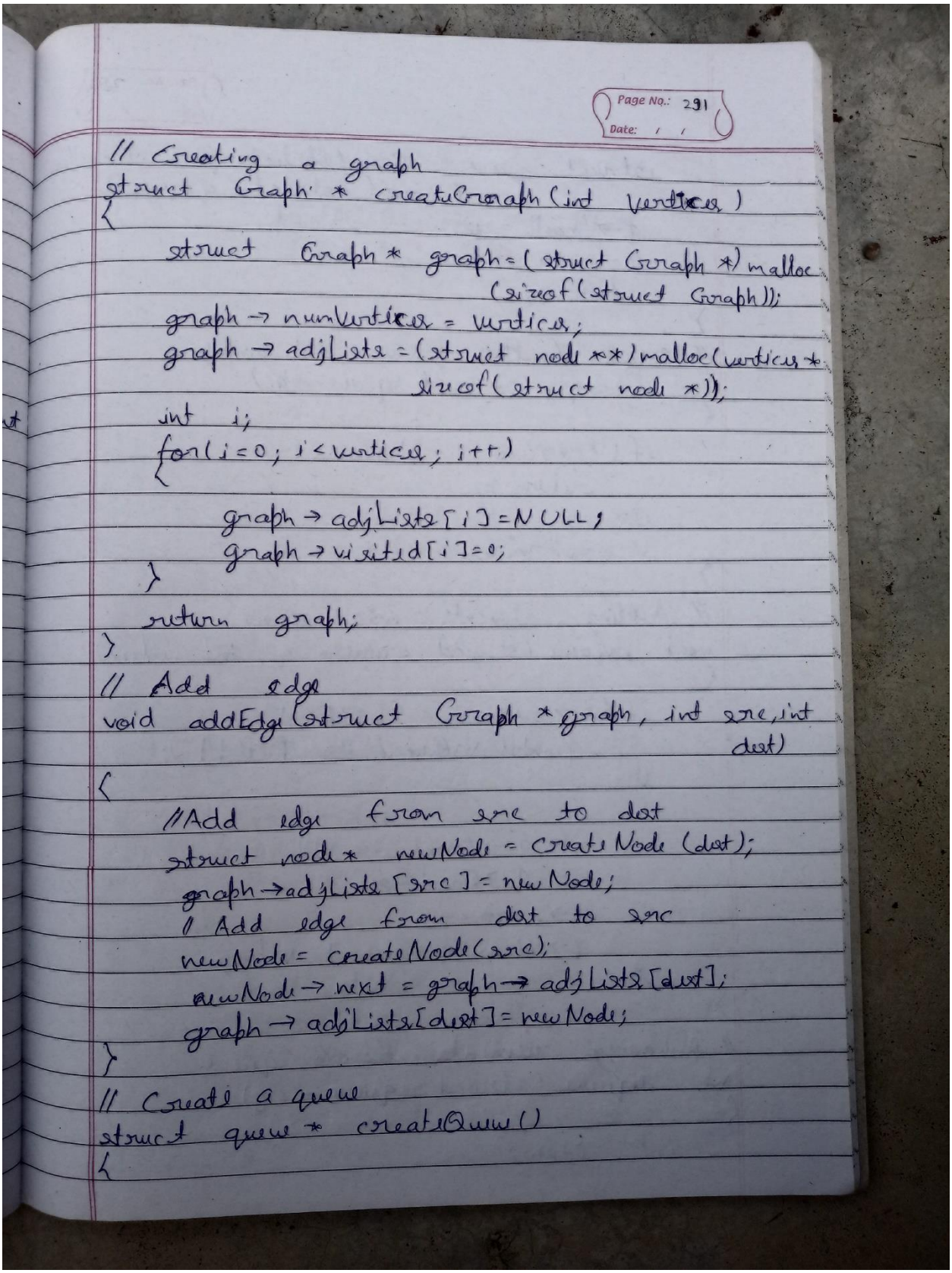
C Code-

```
#include<stdio.h>
#include<stdlib.h>
#define SIZE 40
struct queue {
    int items[SIZE];
    int front;
    int rear;
};
struct queue * createQueue();
void enqueue(struct queue * q, int);
void dequeue(struct queue * q);
void display(struct queue * q);
int isEmpty(struct queue * q);
void printQueue(struct queue *q);
struct node
{
    int vertex;
    struct node * next;
};
struct node * createNode(int);
struct Graph
{
    int numVertices;
    struct node ** adjLists;
    int * visited;
};
// BFS Algorithm
void bfs(struct Graph * graph, int startVertex)
{
```



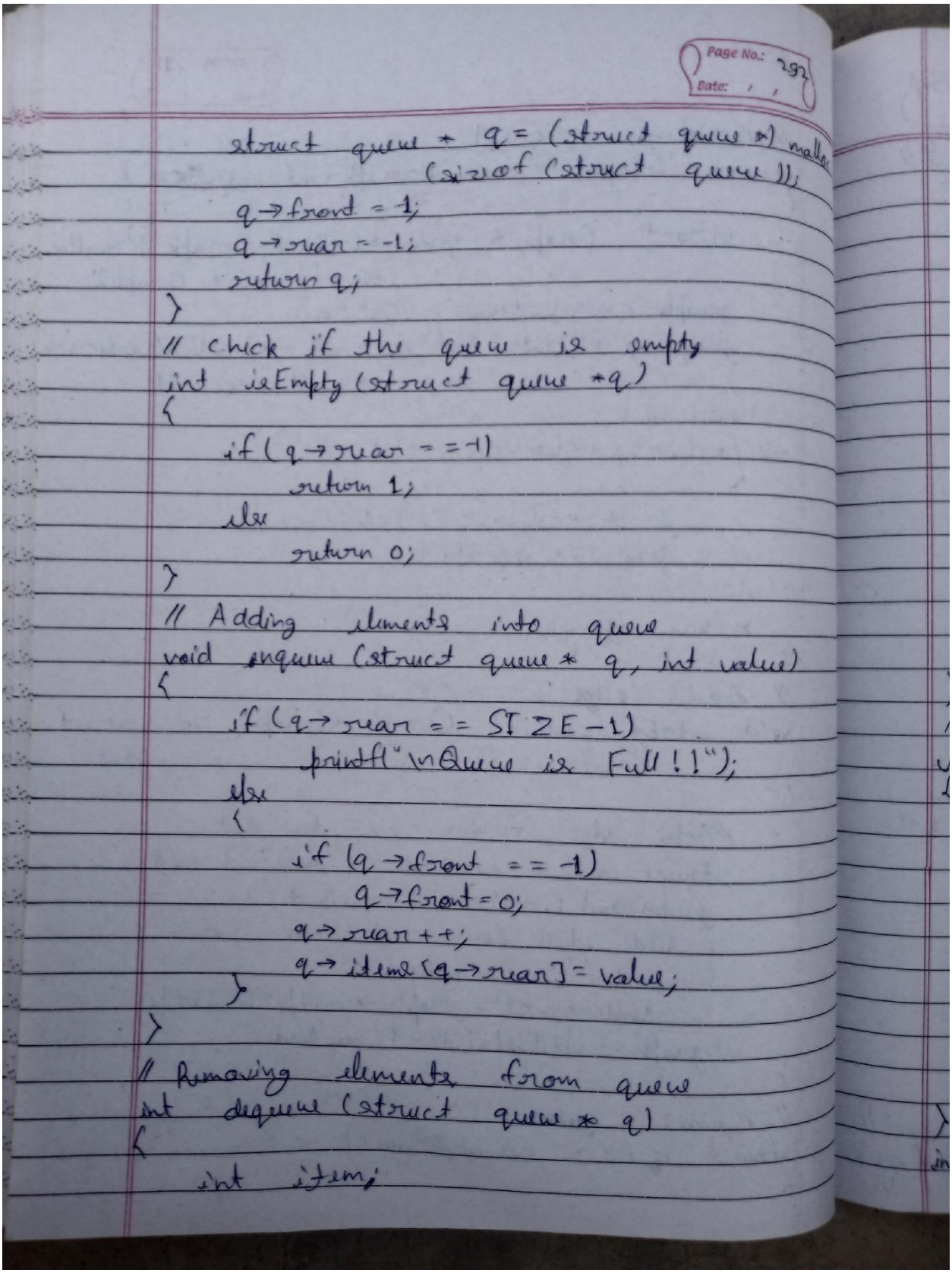
Page No.: 29
Date: / /

```
struct queue* q = createQueue();
graph->visited[startVertex] = 1;
enqueue(q, startVertex);
while(!isEmpty(q))
{
    printQueue(q);
    int currentVertex = dequeue(q);
    printf("Visited %d\n", currentVertex);
    struct node* temp = graph->adjLists[currentVertex];

    while(temp)
    {
        int adjVertex = temp->vertex;
        if(graph->visited[adjVertex]==0)
        {
            graph->visited[adjVertex]=1;
            enqueue(q, adjVertex);
        }
        temp = temp->next;
    }
}
}
// Creating a node
struct node* createNode(int v)
{
    struct node* newNode = (struct node*)
                malloc(sizeof(struct node));
    newNode->vertex = v;
    newNode->next = NULL;
    return newNode;
}
```

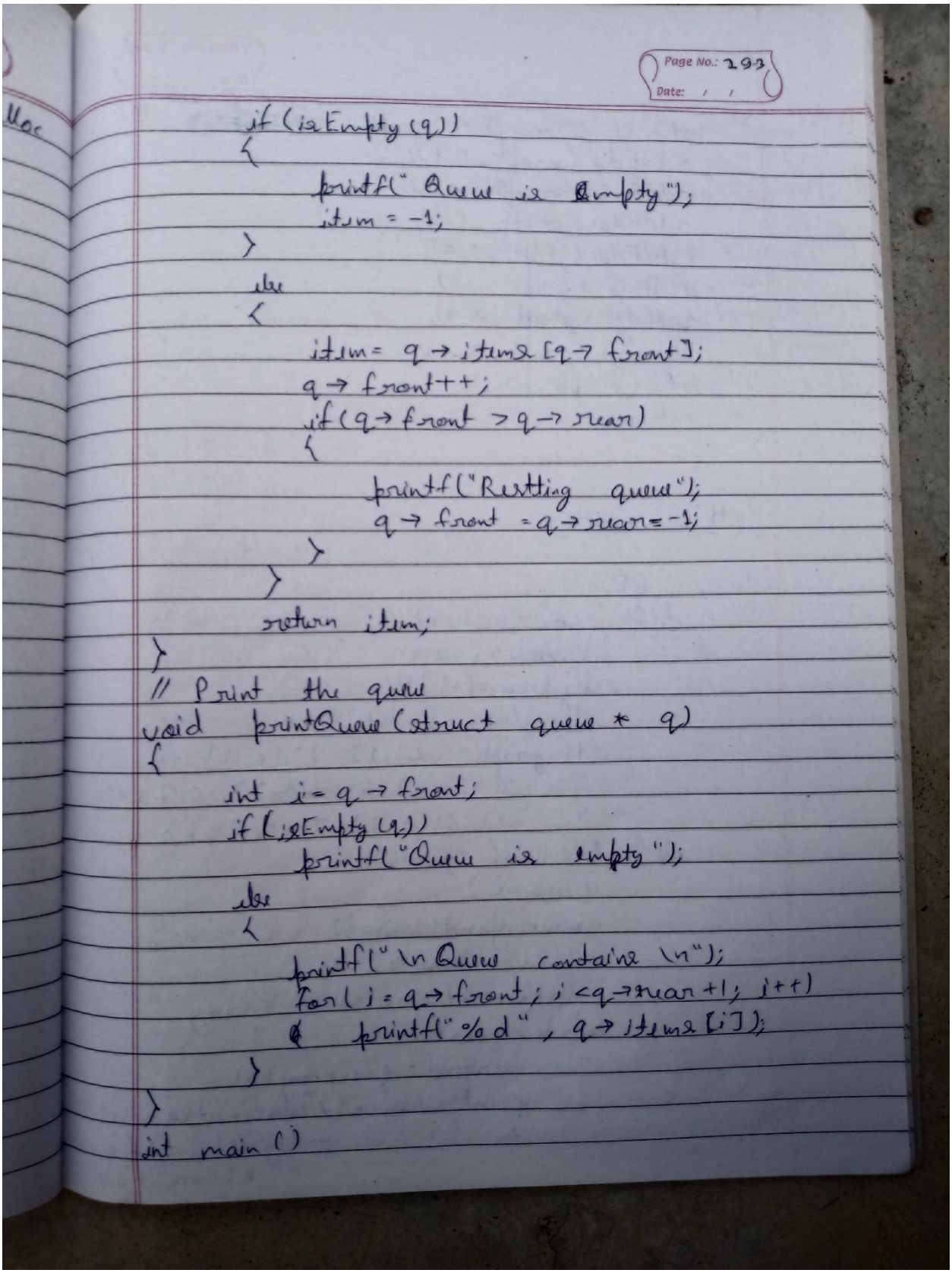


Page No.: 291
Date: / /

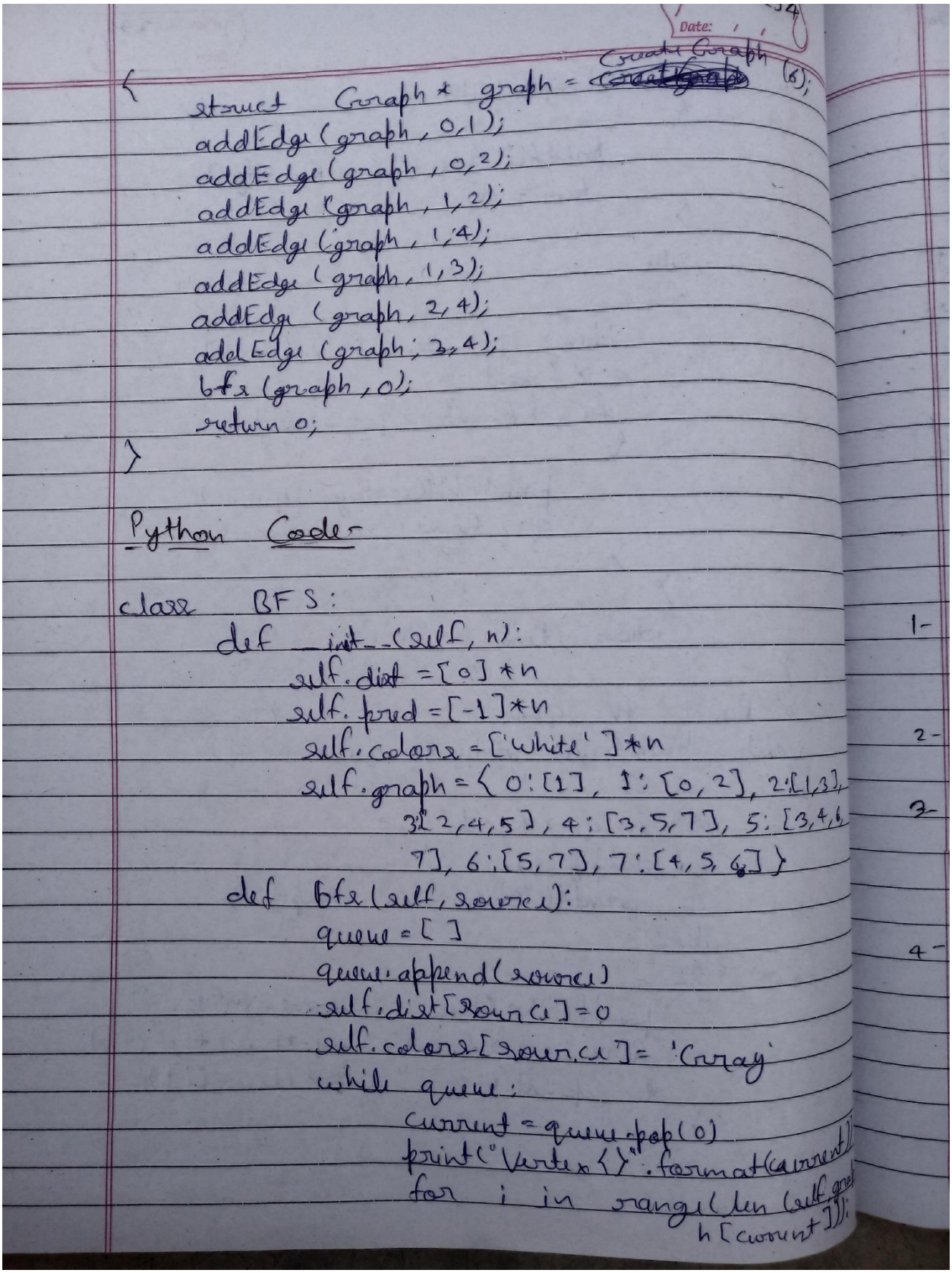
```
// Creating a graph
struct Graph * createGraph(int vertices)
{
    struct Graph * graph = (struct Graph *) malloc
                           (sizeof(struct Graph));
    graph -> numVertices = vertices;
    graph -> adjLists = (struct node **) malloc(vertices *
                        sizeof(struct node *));
    int i;
    for(i=0; i<vertices; i++)
    {
        graph -> adjLists[i] = NULL;
        graph -> visited[i] = 0;
    }
    return graph;
}
// Add edge
void addEdge(struct Graph * graph, int src, int
                                        dest)
{
    //Add edge from src to dest
    struct node * newNode = createNode(dest);
    graph -> adjLists[src] = newNode;
    // Add edge from dest to src
    newNode = createNode(src);
    newNode -> next = graph -> adjLists[dest];
    graph -> adjLists[dest] = newNode;
}
// Create a queue
struct queue * createQueue()
{
```



```
    struct queue * q = (struct queue *) malloc
                (sizeof (struct queue));
    q->front = -1;
    q->rear = -1;
    return q;
}
// check if the queue is empty
int isEmpty (struct queue *q)
{
    if (q->rear == -1)
        return 1;
    else
        return 0;
}
// Adding elements into queue
void enqueue (struct queue * q, int value)
{
    if (q->rear == SIZE-1)
        printf("\nQueue is Full!!");
    else
    {
        if (q->front == -1)
            q->front = 0;
        q->rear++;
        q->items[q->rear] = value;
    }
}
// Removing elements from queue
int dequeue (struct queue * q)
{
    int item;
```



```
Page No.: 293
Date:

    if (isEmpty(q))
    {
        printf("Queue is Empty");
        item = -1;
    }
    else
    {
        item = q->items[q->front];
        q->front++;
        if(q->front > q->rear)
        {
            printf("Resetting queue");
            q->front = q->rear = -1;
        }
    }
    return item;
}
// Print the queue
void printQueue(struct queue * q)
{
    int i = q->front;
    if(isEmpty(q))
        printf("Queue is empty");
    else
    {
        printf("\n Queue contains \n");
        for(i = q->front; i < q->rear+1; i++)
        {   printf("%d", q->items[i]);
        }
}
int main()
```



```
{
    struct Graph * graph = createGraph(6);
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 1, 2);
    addEdge(graph, 1, 4);
    addEdge(graph, 1, 3);
    addEdge(graph, 2, 4);
    addEdge(graph, 3, 4);
    bfs(graph, 0);
    return 0;
}
```

Python Code:-

```
class BFS:
    def __init__(self, n):
        self.dist = [0]*n
        self.pred = [-1]*n
        self.colors = ['white']*n
        self.graph = {0:[1], 1:[0,2], 2:[1,3],
                      3:[2,4,5], 4:[3,5,7], 5:[3,4,6,
                      7], 6:[5,7], 7:[4,5,6]}
    def bfs(self, source):
        queue = []
        queue.append(source)
        self.dist[source] = 0
        self.colors[source] = 'Gray'
        while queue:
            current = queue.pop(0)
            print("Vertex {}".format(current))
            for i in range(len(self.graph[current])):
```

Page No.: 295
Date: / /

```
if self.color[self.graph[current][i]
                == 'white':
    self.color[self.graph[current][i]
                = 'Gray'
    self.dist[self.graph[current][i]]
            = self.dist[current]+1
    self.pred[self.graph[current][i]]
                = current
    queue.append(self.graph[current][i])
self.color[current]='Black'
```

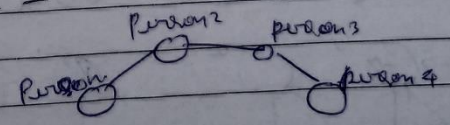
```
b= BFS(8)
b.bfs(2)
```

Applications of BFS-

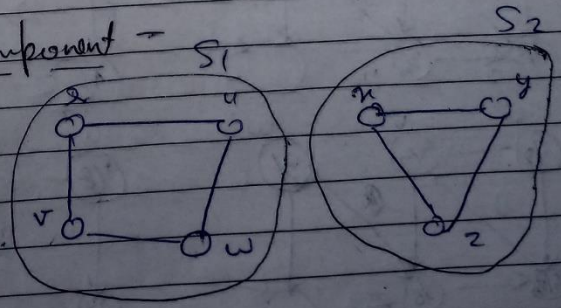
1- s: source - Short path from s to u in an unweighted Graph.

2- Web Crawler - (google) web graph

3- Social-network -



4- Connected - Component -



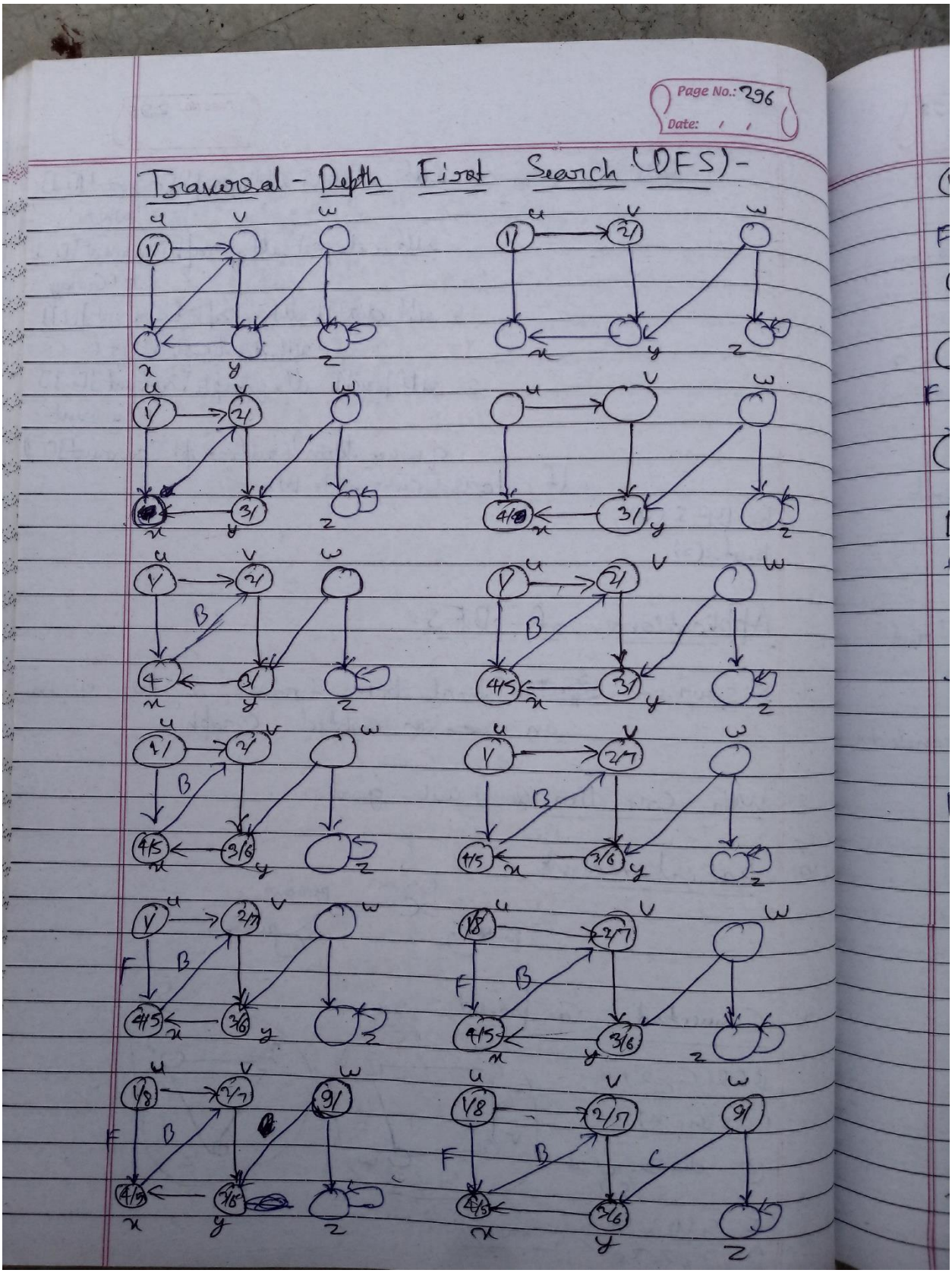
BFS(G, s)

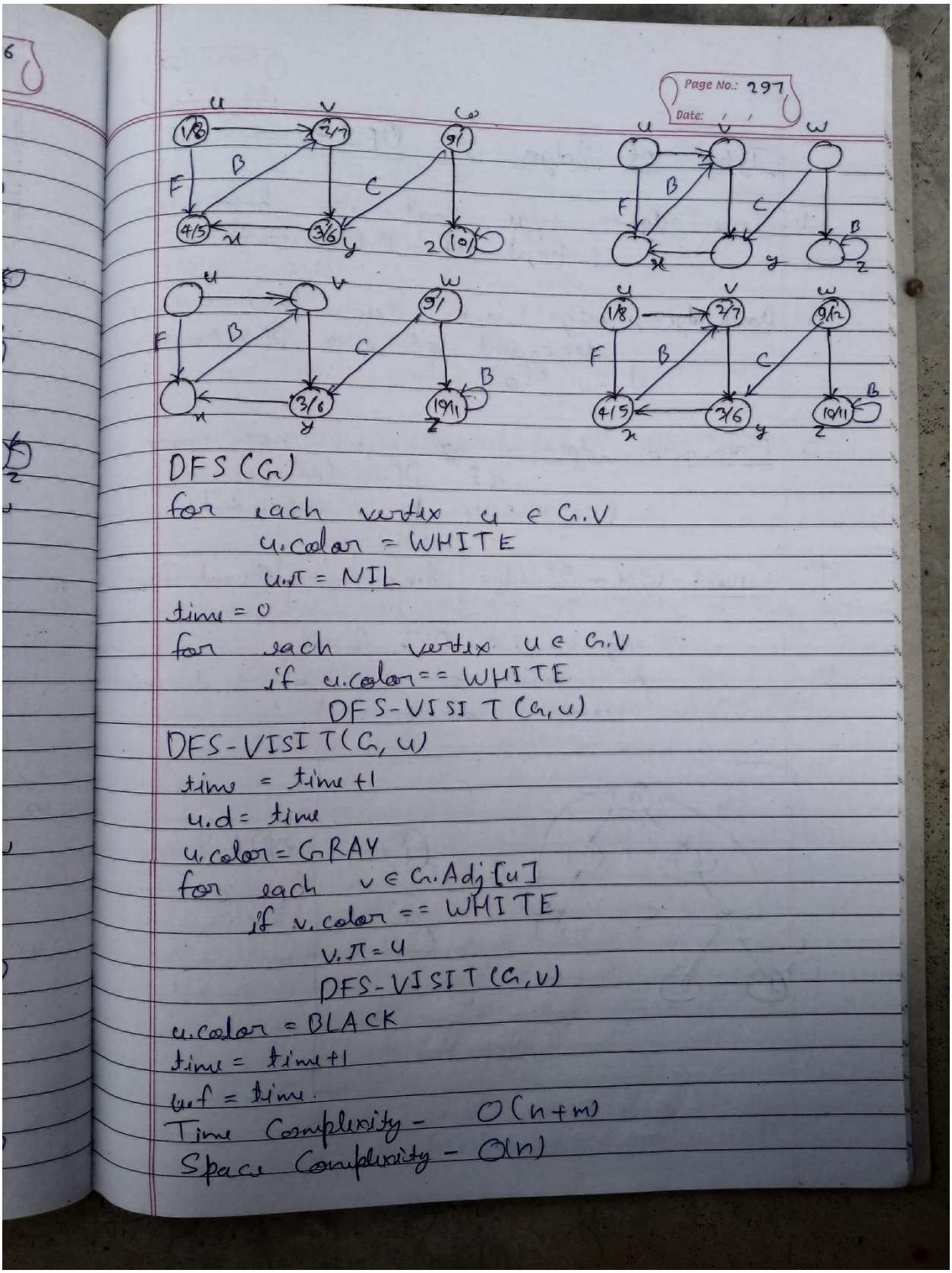
BFS(G, x)

① disconnected (S1, S2)

② s, u, v, w, → connected

③ x, y, z


Page No.: 296
Date: / /
Traversal - Depth First Search (DFS)-




```
DFS(G)
for each vertex u ∈ G.V
        u.color = WHITE
        u.π = NIL
time = 0
for each vertex u ∈ G.V
        if u.color == WHITE
                DFS-VISIT(G, u)
DFS-VISIT(G, u)
time = time + 1
u.d = time
u.color = GRAY
for each v ∈ G.Adj[u]
        if v.color == WHITE
                v.π = u
                DFS-VISIT(G, v)
u.color = BLACK
time = time + 1
u.f = time.
Time Complexity - O(n+m)
Space Complexity - O(n)
```

Page No.: 298
Date: / /

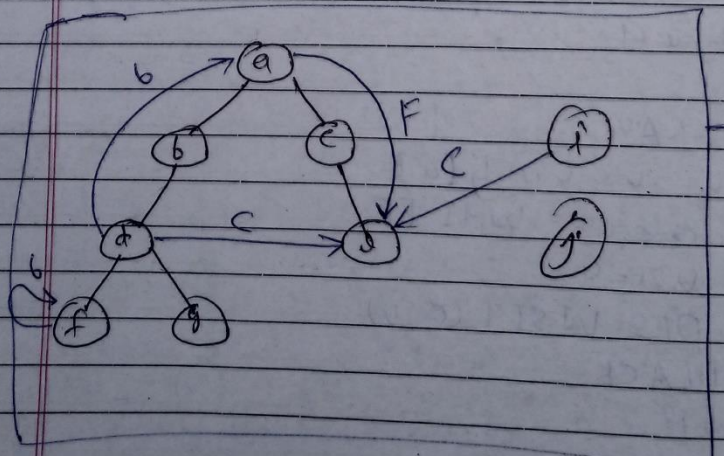
Type of edges in DFS -

1- Tree edge - edge that is predecessor part of DFS forest.

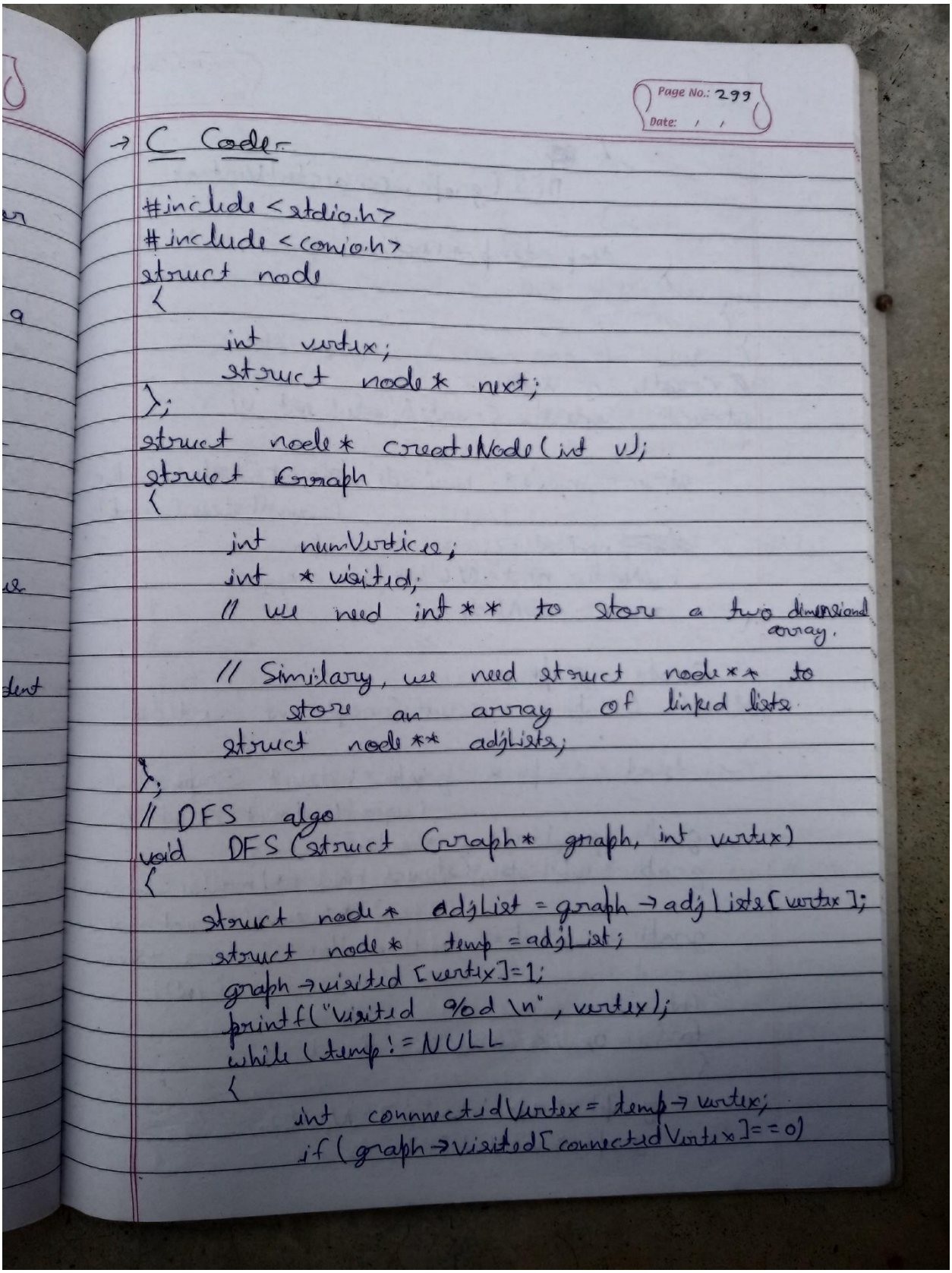
2- Back edge - a) edge (u, v) such that u is a decendent of v in DFS Forest
b) self loop

3- Forward edge - edge (u, v) not part of DFS - forest.
(u: ancestor ~~to~~ v: decendent)

4- Cross - edge - a) edge between different Tree
b) edge (u, v) ≠ DFS forest
no relation of ancestor & decendent between u & v



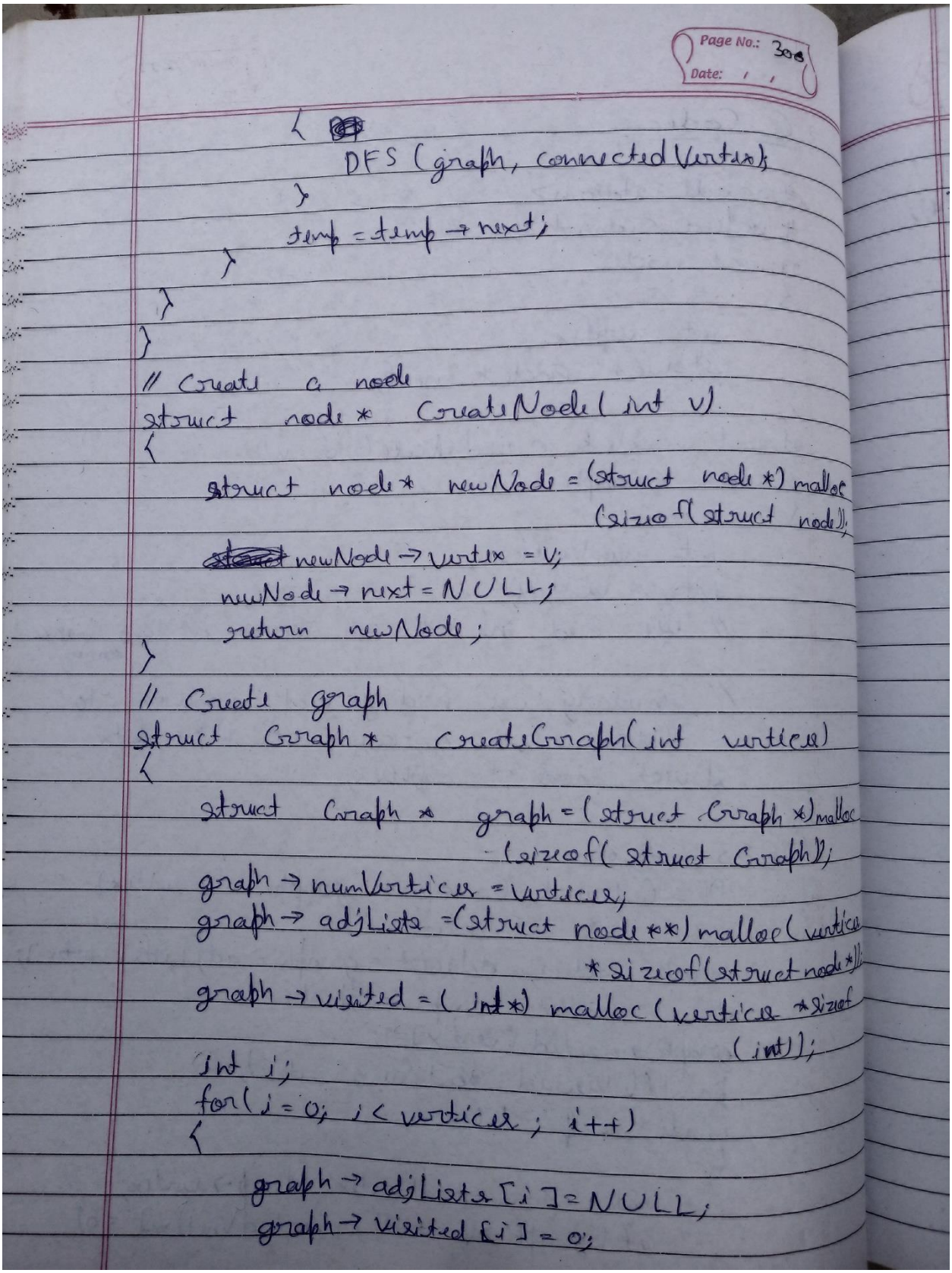
→ DFS Forest



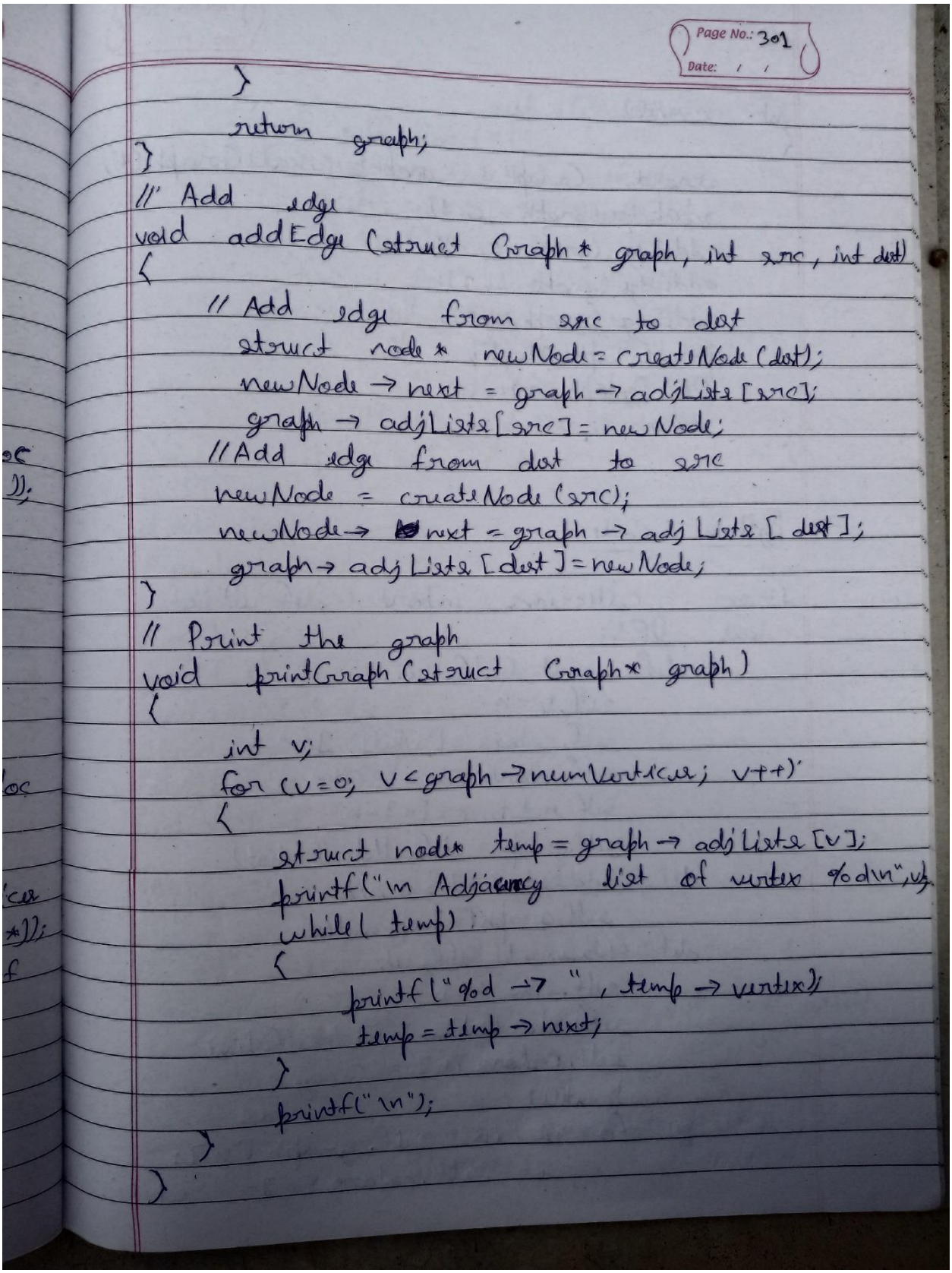
Page No.: 299
Date: / /

→ C Code:-

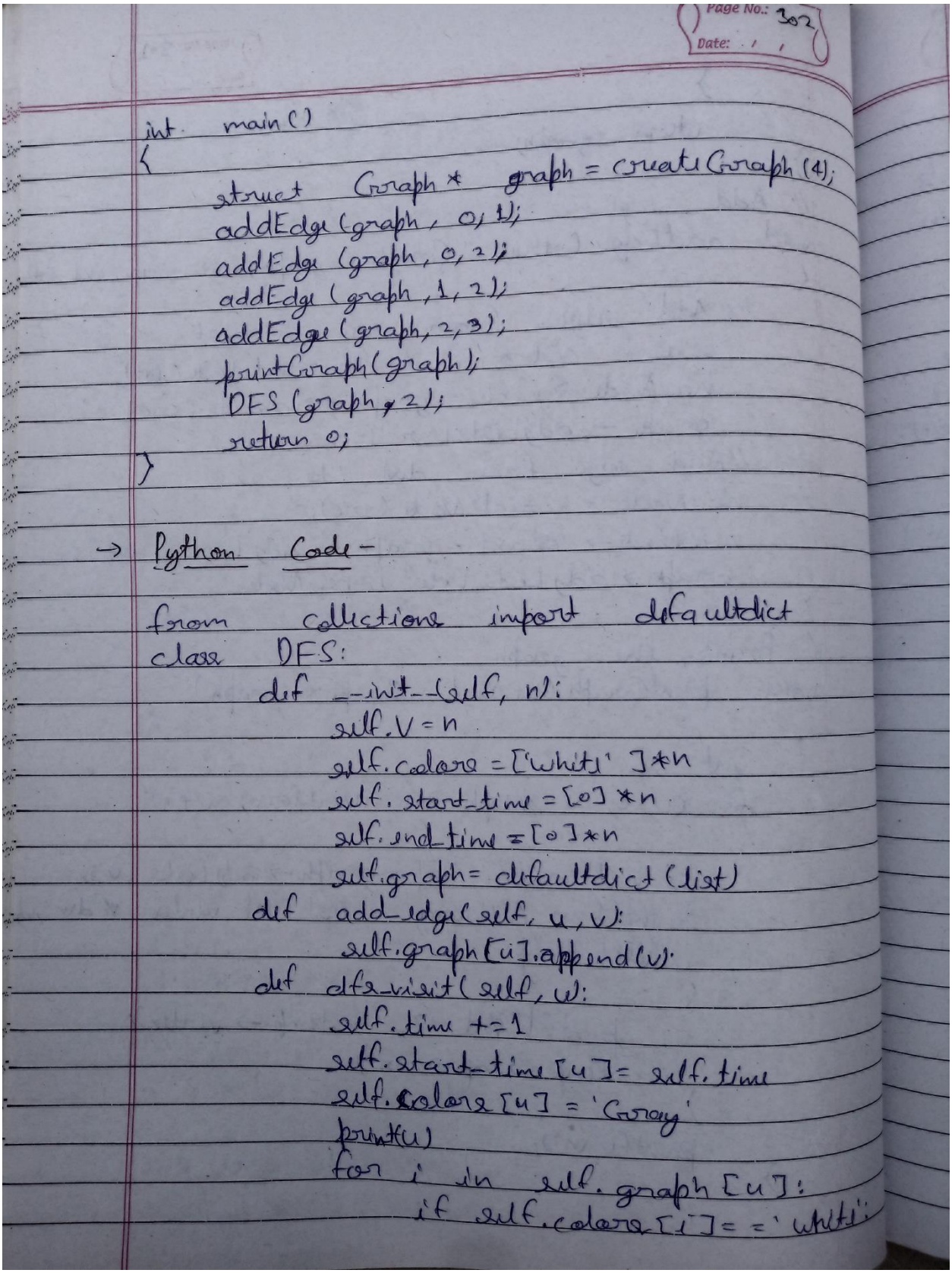
```
#include <stdio.h>
#include <conio.h>
struct node
{
    int vertex;
    struct node* next;
};
struct node* createNode(int v);
struct Graph
{
    int numVertices;
    int *visited;
    // we need int** to store a two dimensional array.
    // Similary, we need struct node** to
       store an array of linked lists.
    struct node** adjLists;
};
// DFS algo
void DFS(struct Graph* graph, int vertex)
{
    struct node* adjList = graph->adjLists[vertex];
    struct node* temp = adjList;
    graph->visited[vertex]=1;
    printf("visited %d \n", vertex);
    while(temp!=NULL
    {
        int connectedVertex = temp->vertex;
        if(graph->visited[connectedVertex]==0)
```



```
            {
                DFS(graph, connectedVertex);
            }
            temp = temp -> next;
        }
    }
}
// Create a node
struct node * createNode(int v)
{
    struct node* newNode = (struct node *) malloc
                           (sizeof(struct node));
    newNode -> vertex = v;
    newNode -> next = NULL;
    return newNode;
}
// Create graph
struct Graph * createGraph(int vertices)
{
    struct Graph * graph = (struct Graph *) malloc
                           (sizeof(struct Graph));
    graph -> numVertices = vertices;
    graph -> adjLists = (struct node **) malloc(vertices
                        * sizeof(struct node*));
    graph -> visited = (int*) malloc(vertices * sizeof
                       (int));
    int i;
    for(i = 0; i < vertices; i++)
    {
        graph -> adjLists[i] = NULL;
        graph -> visited[i] = 0;
```



```
    }
    return graph;
}
// Add edge
void addEdge(struct Graph* graph, int src, int dest)
{
    // Add edge from src to dest
    struct node* newNode = createNode(dest);
    newNode->next = graph->adjLists[src];
    graph->adjLists[src] = newNode;
    // Add edge from dest to src
    newNode = createNode(src);
    newNode->next = graph->adjLists[dest];
    graph->adjLists[dest] = newNode;
}
// Print the graph
void printGraph(struct Graph* graph)
{
    int v;
    for (v=0; v<graph->numVertices; v++)
    {
        struct node* temp = graph->adjLists[v];
        printf("\n Adjacency list of vertex %d\n", v);
        while(temp)
        {
            printf("%d -> ", temp->vertex);
            temp = temp->next;
        }
        printf("\n");
    }
}
```



```
int main()
{
    struct Graph* graph = createGraph(4);
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 1, 2);
    addEdge(graph, 2, 3);
    printGraph(graph);
    DFS(graph, 2);
    return 0;
}
```

→ Python Code -

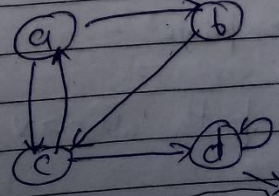
```
from collections import defaultdict
class DFS:
    def __init__(self, n):
        self.V = n
        self.colors = ['white' ]*n
        self.start_time = [0] *n
        self.end_time = [0]*n
        self.graph = defaultdict(list)
    def add_edge(self, u, v):
        self.graph[u].append(v)
    def dfs_visit(self, u):
        self.time += 1
        self.start_time[u] = self.time
        self.colors[u] = 'Gray'
        print(u)
        for i in self.graph[u]:
            if self.colors[i] == 'white':
```

Page No.: 303
Date: / /

```
                self.dfs_visit(i)
        self.time += 1
        self.end_time[u] = self.time
        self.colors[u] = 'Black'
    def dfs(self):
        for i in range(self.V):
            self.time = 0
            if self.colors[i] == 'white':
                self.dfs_visit(i)
        print(self.start_time)
        print(self.end_time)
d = DFS(6)
d.add_edge(0,1)
d.add_edge(0,2)
d.add_edge(1,2)
d.add_edge(2,3)
d.add_edge(3,1)
d.add_edge(4,3)
d.add_edge(4,5)
d.add_edge(5,5)
d.dfs()
```

Application of DFS - Finding cycle in a di-graph -

di-graph -



Cycle → u ~ v

di path u to itself

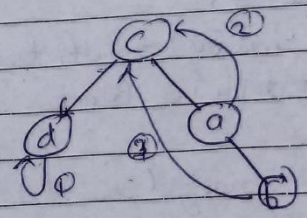
u ~ u

d → d ⓐ
a → c → a ⓑ
a → b → c → a ⓒ

No. of back edge = No. of cycle in digraph

Page No.: 304
Date: / /

DFS - Forest / Tree -

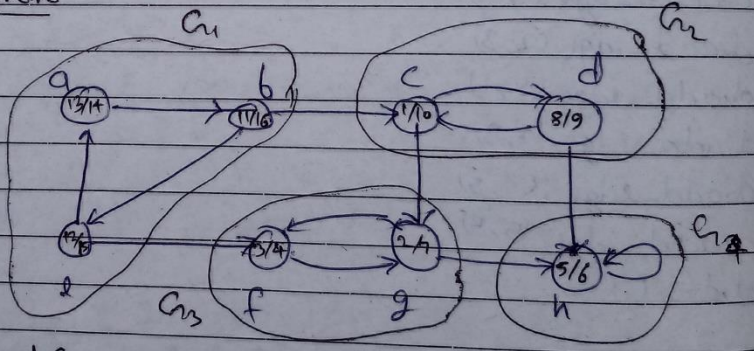


(u,v) where v is ancestor & u is a descendent in DFS - tree.

DFS: - ~~If~~ W, G, B the back - edge - (u,v) if v = Gray the (u,v) is a back edge (cycle).

Time Complexity - $O(V+E) = O(m+n)$

Application - Find ~~st~~ strongly Connected Component -



4 Strongly Connected Components.

Strongly Connected Component -

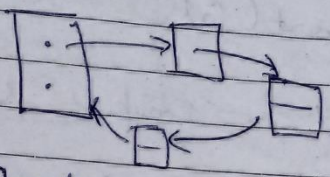
① G' which has path $u \rightsquigarrow v$ $\forall u,v \in G'$

② G' is maximal set of vertices that satisfy ① above

Page No.: 305
Date: 21/11/20

Need to detect strongly connected Component -

Web graph -

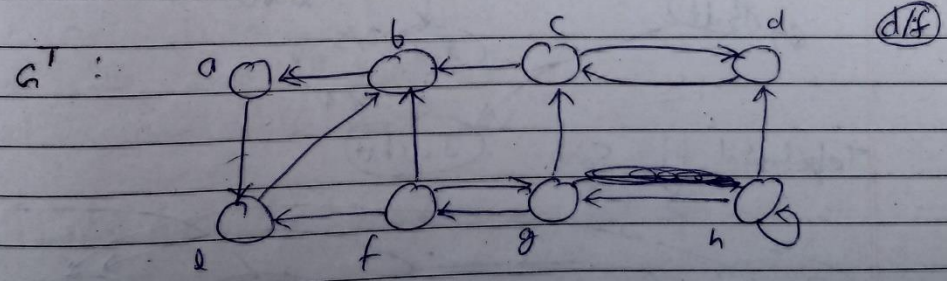


node: webpage
edge: hyperlinks (dir)
(very similar in context)

Strongly connected component in web graph

Kosaraju's Algo -

1) Compute DFS(G) & finishing times (u.f) for each $u \in G$
2) G^T: Transpose Graph (Construct)
   G: (u) → (v) G^T: (u) ← (v)
   (recursively)
3) Call DFS(G^T) with specific ordering of vertices → decr. order of u.f.



Output the vertices of each tree in the DFS-forest is strongly connected component.

Time Complexity - $O(n+m)$
Space Complexity - $O(n+m)$

Page No.: 306
Date: / /

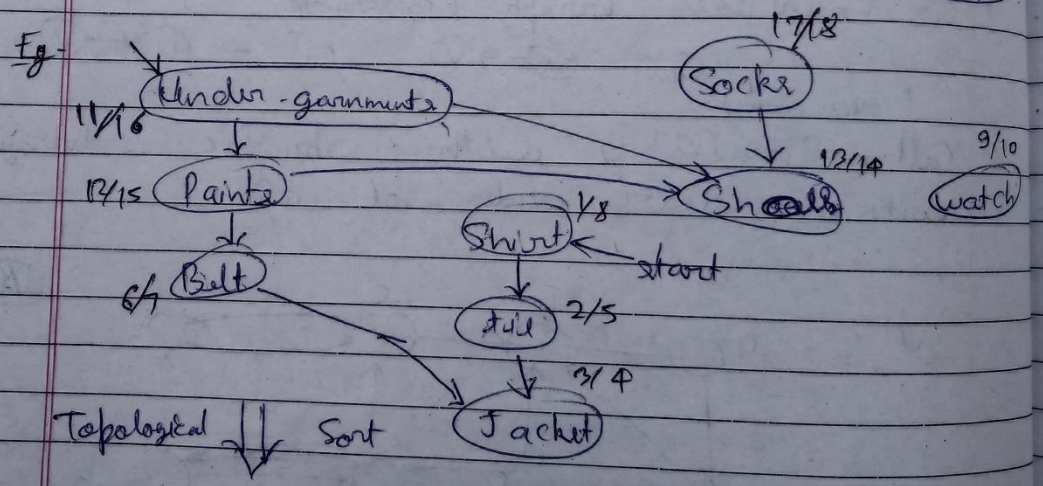
Topological Sort -

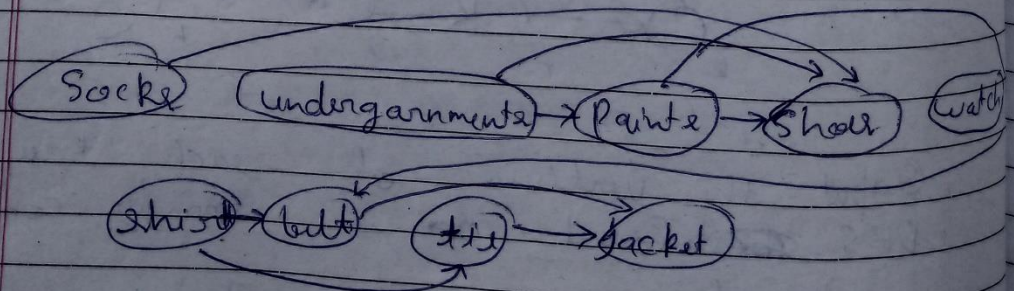
Topological_Sort(G)

1) Call DFS(G) & compute d.f for each vertex u
2) As each vertex finishes, insert the vertex at the front of a linked list.
3) return the linked list

Time Complexity - O(n+m)

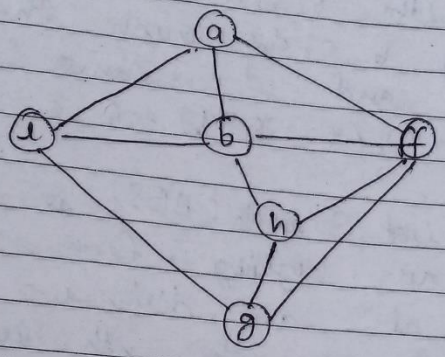
Eg -





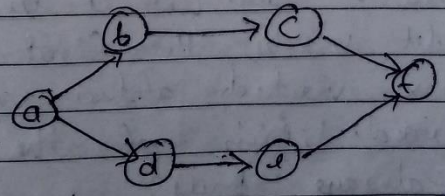
Page No.: 207
Date: / /

Q- Consider Graph-



Among the following sequences-
(i) a b e g h f
(ii) a b f e h g (there is no edge from f to e) ×
(iii) a b f h g e
(iv) a f g h b e
which are depth first traversals of the above graph?
(A) I, II and IV only
(B) I and IV only
(C) II, III and IV only
(D) I, III and IV only (✓)

Q- Consider the following directed graph -



b → c
d → e

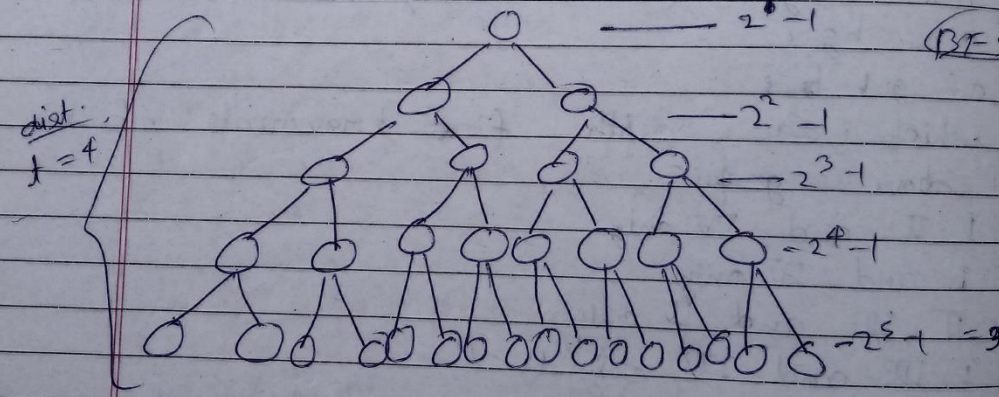
b c d e ✓	d e b c ✓
b d c e ✓	d b e c ×
b d e c ✓	d b c e ×
d d e c ✓	d

a - - - - f

The number of different topological orderings of the vertices of the graph is 6

a - - - - f . The blanck spaces are to be filled with b, c, d, e. Number of ways to arrange b, c, d, such that b come before c and d come before e, will be = 4! / (2! * 2!) = 6

Q- Breadth First Search (BFS) is started on a binary tree beginning from the root vertex t at a distance four the root. If t is the nth vertex in the BFS traversal, then the maximum possible value of n is 31



Q- Let G be an undirected graph. Consider a depth-first search tree. Let u be a vertex in G and let v be the first new unvisited vertex visited after visiting u in the traversal. Which of the following statements is always true?

(A) {u,v} must be an edge in G, and u is a descendant of v in T.

(B) {u,v} must be an edge in G, and v is a

descendant of u in T

(C) If {u,v} is not an edge in G then u is a leaf in T. (✓)

(D) If {u,v} is not an edge in G then u and v must have the same parent in T

Q- Let G be a graph with n vertices and m edges. What is the tightest upper bound on the running time on Depth First Search of G? Assume that the graph is represented using adjacency matrix.

(A) $O(n)$

(B) $O(m+n)$

(C) $\Theta(n^2)$ (✓)

(D) $O(m*n)$

Q- Consider the tree arcs (edges or links) of a BFS traversal from a source node W in an unweighted, connected, undirected graph. The tree T formed by the three arcs is a data structure for computing -

(A) the shortest path between every pair of vertices.

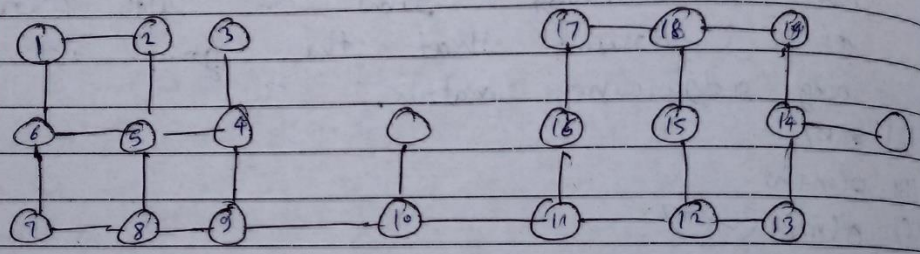
(B) the shortest path from W to every vertex in the graph. (✓)

(C) the shortest paths from W to only those node that are leaves of T

(D) the longest path in the graph.

Page No.: 310
Date: / /

Q- Suppose depth first search is executed on the graph below starting at some unknown vertex. Assume that a recursive call to visit a vertex is made only after first checking that the vertex has not been visited earlier. Then the maximum possible recursion depth (including the initial call) is-

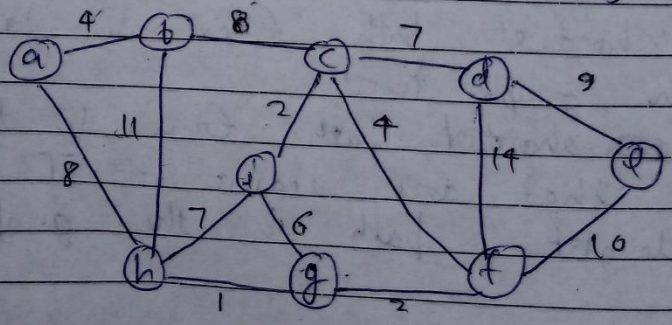


(A) 17
(B) 18
(C) 19 ✓
(D) 20

Weighted Graph (Undirected) -

(Every edge has a numerical weight)

Eg



Page No.: 311
Date: 22/11/20

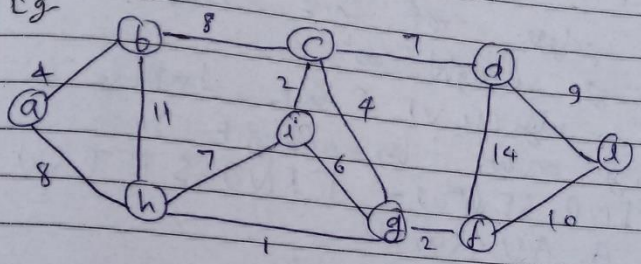
Spanning Tree -

A cyclic connected graph G is such that then Spanning tree T is such that-

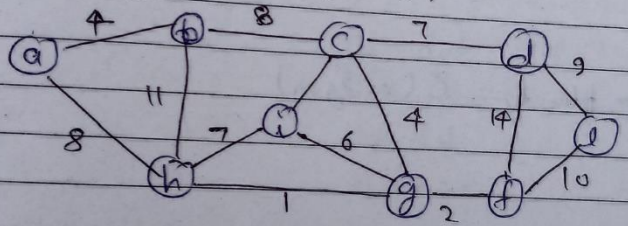
$T.V = G.V$

$T.E \subseteq G.E$

Eg-



: T



Length = 37

Minimah Spanning Tree -

Length of Spanning tree = Sum of edge weights in T.

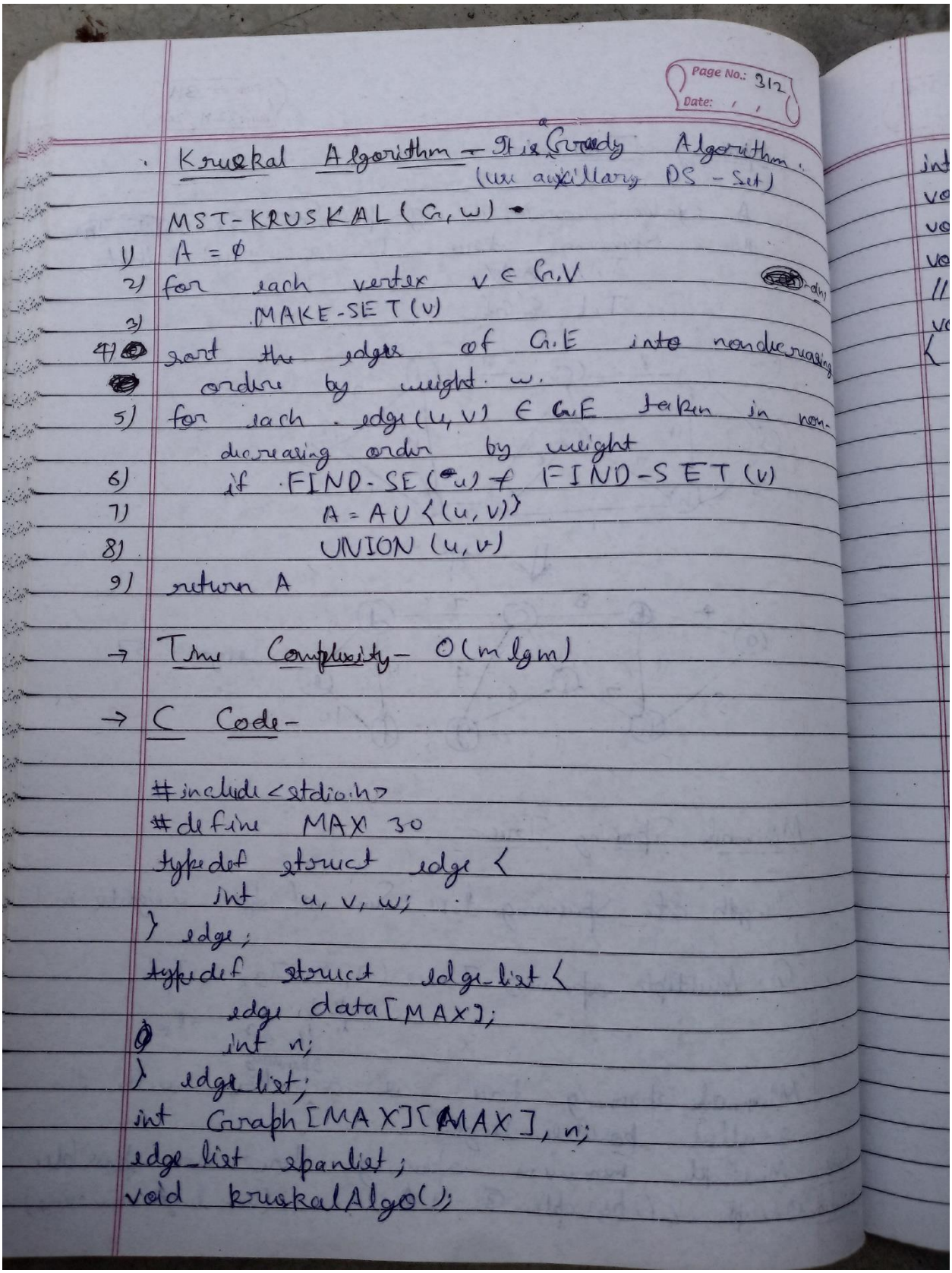
G: Multiple spanning Trees ($T_1, T_2, T_3, \dots T_K$)

$\downarrow l_1 \quad \downarrow l_2 \quad \downarrow l_3 \quad \dots \quad \downarrow l_K$

"Minimah spanning tree is a spanning tree with the smallest possible length."

Note- Multiple minimum spanning trees are possible.

Application ① Telephone / Fiber optic ② - Electronics (min length wiring)



Page No.: 312
Date: / /

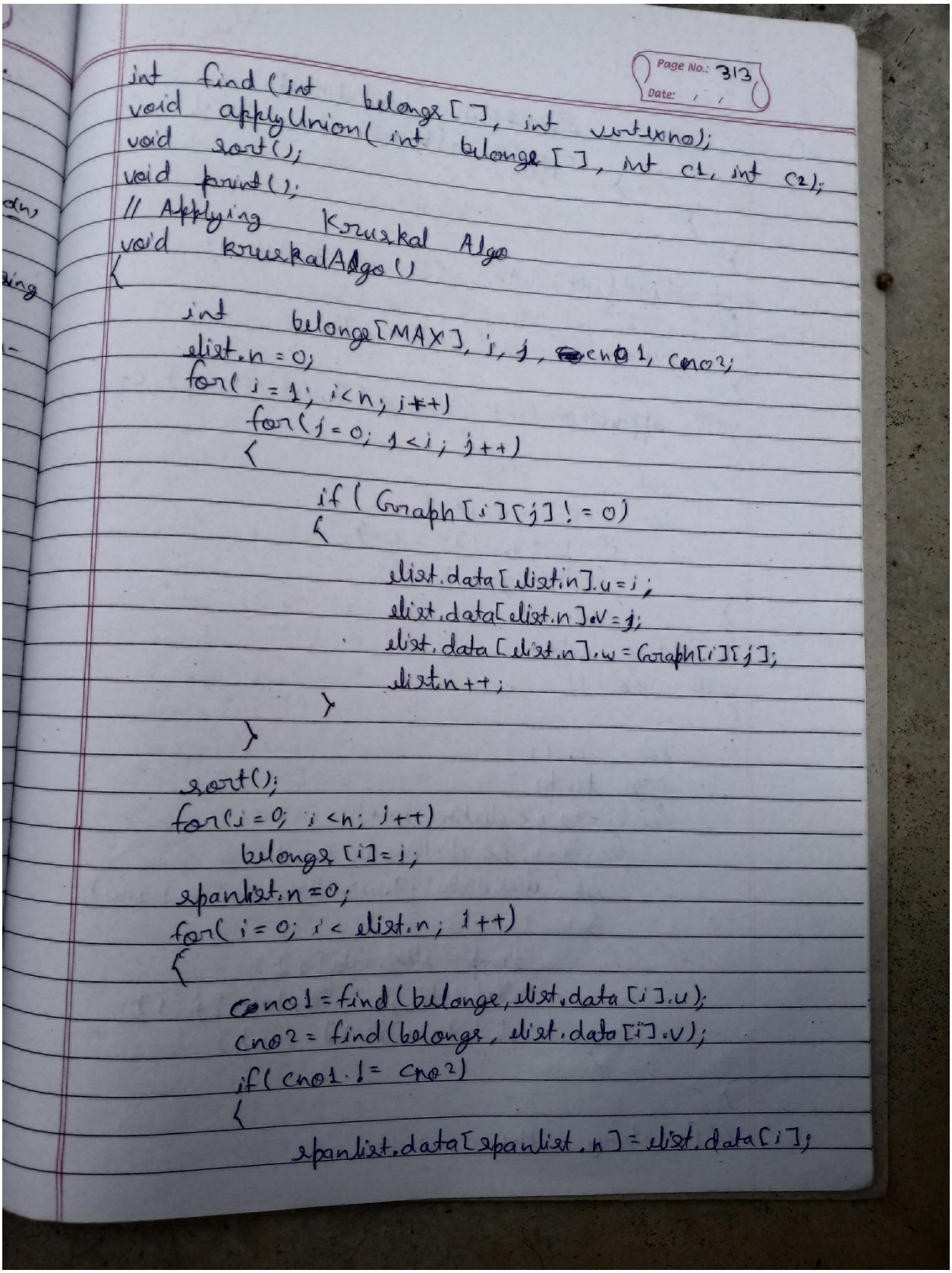
Kruskal Algorithm – It is a Greedy Algorithm.
(Use auxiliary DS – Set)

```
MST-KRUSKAL(G, w)
1) A = ∅
2) for each vertex v ∈ G.V
3)     MAKE-SET(v)
4) sort the edges of G.E into nondecreasing
   order by weight w.
5) for each edge(u, v) ∈ G.E taken in non-
   decreasing order by weight
6)     if FIND-SET(u) ≠ FIND-SET(v)
7)         A = A ∪ {(u, v)}
8)         UNION(u, v)
9) return A
```

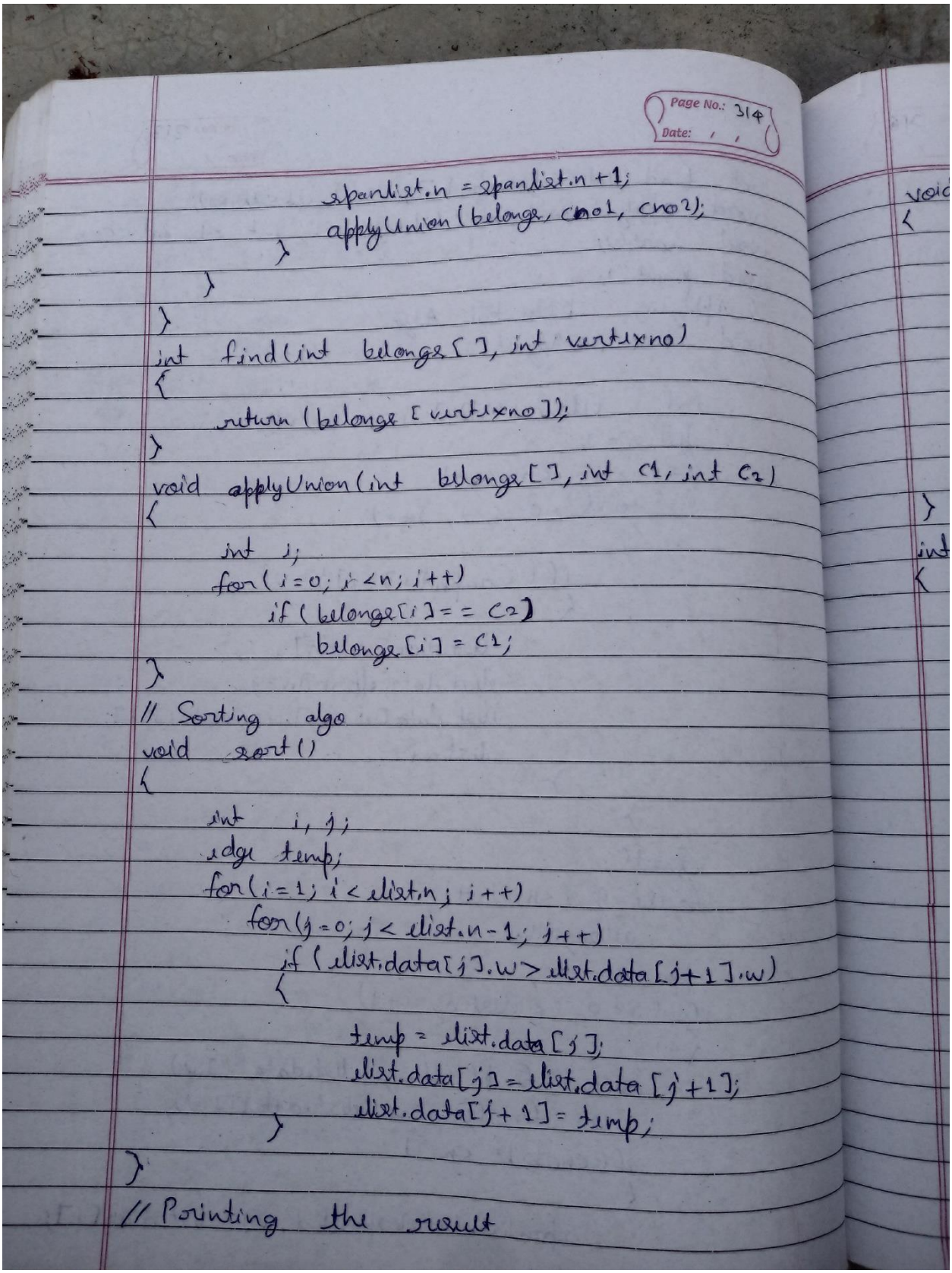
→ Time Complexity – O(m lg m)

→ C Code –

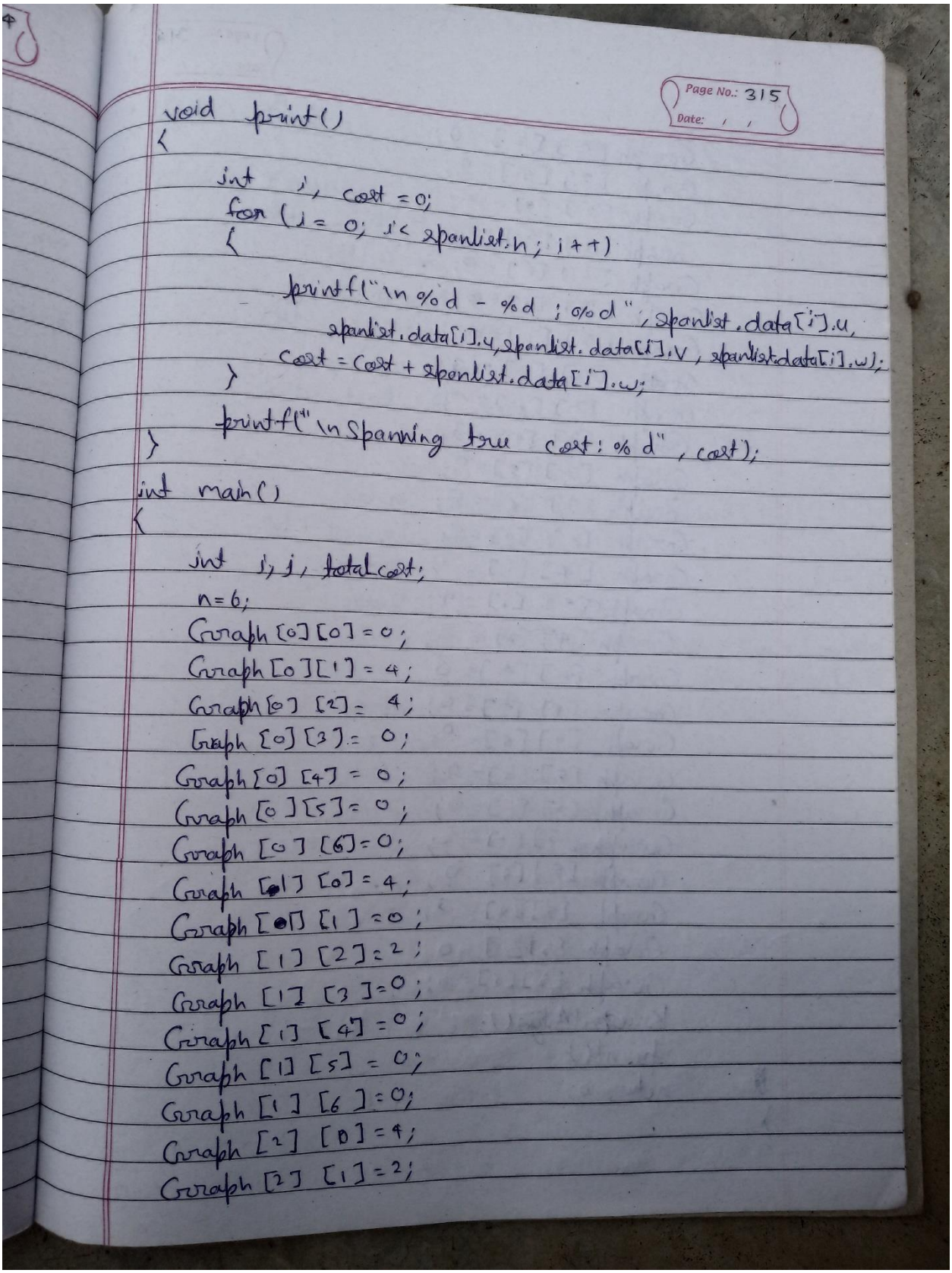
```
#include <stdio.h>
#define MAX 30
typedef struct edge {
    int u, v, w;
} edge;
typedef struct edge_list {
    edge data[MAX];
    int n;
} edge_list;
int Graph[MAX][MAX], n;
edge_list spanlist;
void kruskalAlgo();
```



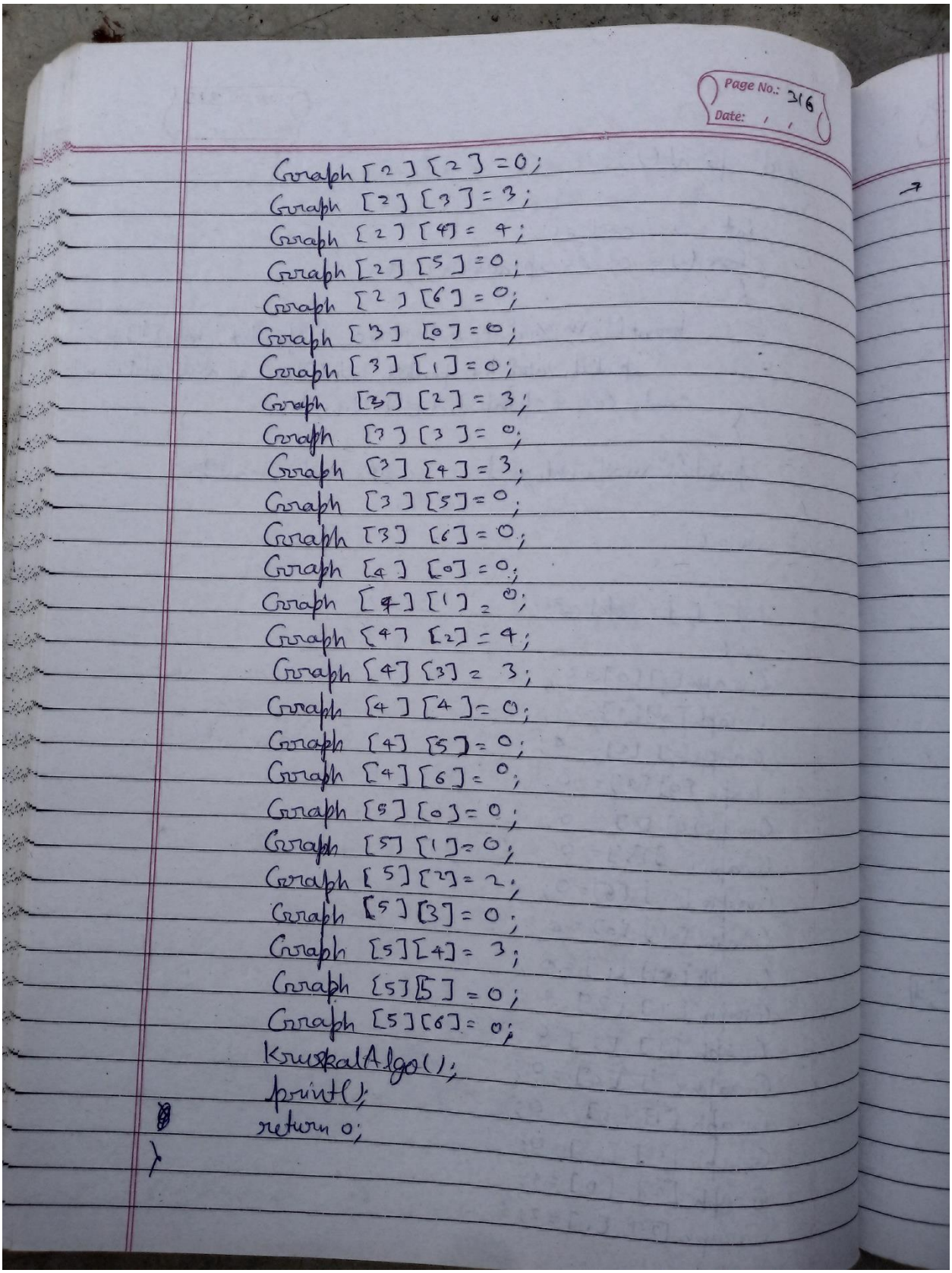
```
int find(int belongs[], int vertexno);
void applyUnion(int belongs[], int c1, int c2);
void sort();
void print();
// Applying Kruskal Algo
void kruskalAlgo()
{
    int belongs[MAX], i, j, cno1, cno2;
    elist.n = 0;
    for(i=1; i<n; i++)
        for(j=0; j<i; j++)
        {
            if(Graph[i][j] != 0)
            {
                elist.data[elist.n].u = i;
                elist.data[elist.n].v = j;
                elist.data[elist.n].w = Graph[i][j];
                elist.n++;
            }
        }
    sort();
    for(i=0; i<n; i++)
        belongs[i] = i;
    spanlist.n = 0;
    for(i=0; i<elist.n; i++)
    {
        cno1 = find(belongs, elist.data[i].u);
        cno2 = find(belongs, elist.data[i].v);
        if(cno1 != cno2)
        {
            spanlist.data[spanlist.n] = elist.data[i];
```



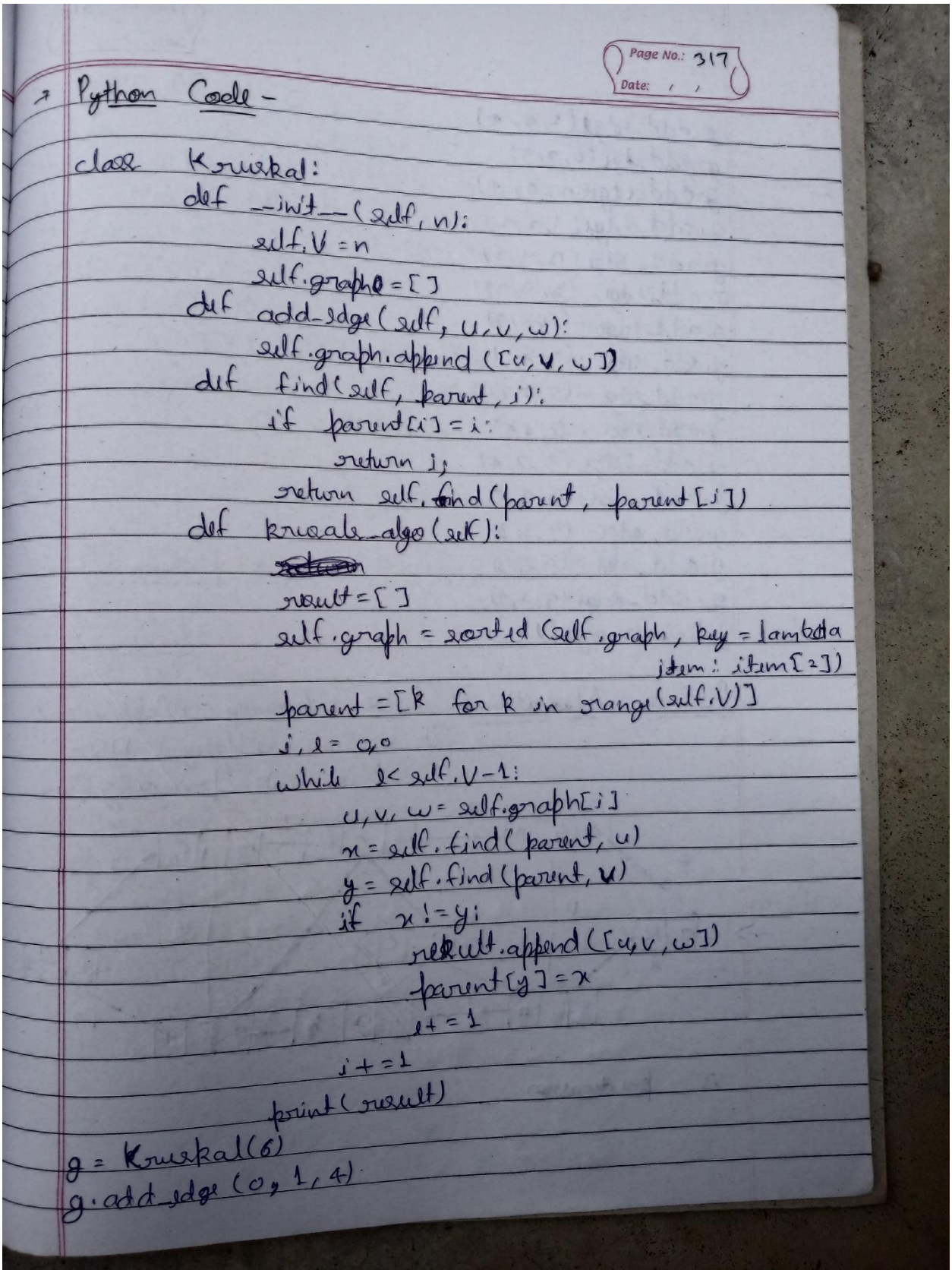
```
                spanlist.n = spanlist.n + 1;
                applyUnion(belongs, cno1, cno2);
            }
        }
    }
int find(int belongs[], int vertexno)
{
    return(belongs[vertexno]);
}
void applyUnion(int belongs[], int c1, int c2)
{
    int i;
    for(i=0; i<n; i++)
        if(belongs[i] == c2)
            belongs[i] = c1;
}
// Sorting algo
void sort()
{
    int i, j;
    edge temp;
    for(i=1; i<elist.n; i++)
        for(j=0; j<elist.n-1; j++)
            if(elist.data[j].w > elist.data[j+1].w)
            {
                temp = elist.data[j];
                elist.data[j] = elist.data[j+1];
                elist.data[j+1] = temp;
            }
}
// Printing the result
```



```
void print()
{
    int i, cost = 0;
    for (i = 0; i < spanlist.n; i++)
    {
        printf("\n %d - %d : %d", spanlist.data[i].u,
            spanlist.data[i].v, spanlist.data[i].w);
        cost = cost + spanlist.data[i].w;
    }
    printf("\n Spanning tree cost: %d", cost);
}
int main()
{
    int i, j, total cost;
    n = 6;
    Graph[0][0] = 0;
    Graph[0][1] = 4;
    Graph[0][2] = 4;
    Graph[0][3] = 0;
    Graph[0][4] = 0;
    Graph[0][5] = 0;
    Graph[0][6] = 0;
    Graph[1][0] = 4;
    Graph[1][1] = 0;
    Graph[1][2] = 2;
    Graph[1][3] = 0;
    Graph[1][4] = 0;
    Graph[1][5] = 0;
    Graph[1][6] = 0;
    Graph[2][0] = 4;
    Graph[2][1] = 2;
```



```
Graph[2][2]=0;
Graph[2][3]=3;
Graph[2][4]=4;
Graph[2][5]=0;
Graph[2][6]=0;
Graph[3][0]=0;
Graph[3][1]=0;
Graph[3][2]=3;
Graph[3][3]=0;
Graph[3][4]=3;
Graph[3][5]=0;
Graph[3][6]=0;
Graph[4][0]=0;
Graph[4][1]=0;
Graph[4][2]=4;
Graph[4][3]=3;
Graph[4][4]=0;
Graph[4][5]=0;
Graph[4][6]=0;
Graph[5][0]=0;
Graph[5][1]=0;
Graph[5][2]=2;
Graph[5][3]=0;
Graph[5][4]=3;
Graph[5][5]=0;
Graph[5][6]=0;
KruskalAlgo();
print();
return 0;
}
```



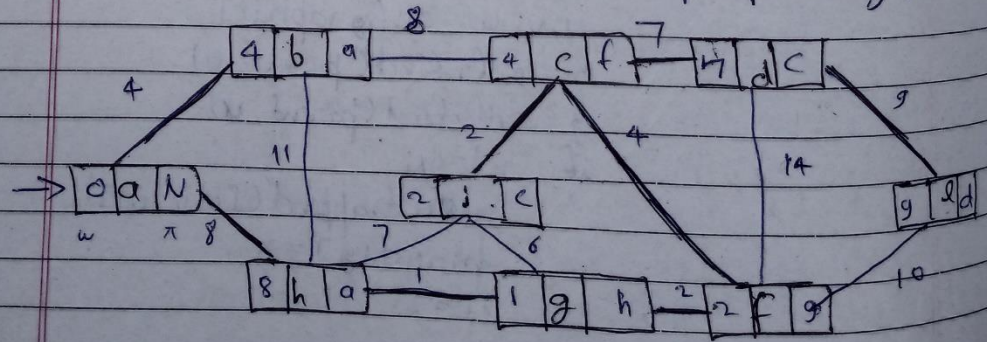
Page No.: 317
Date: / /

→ Python Code -

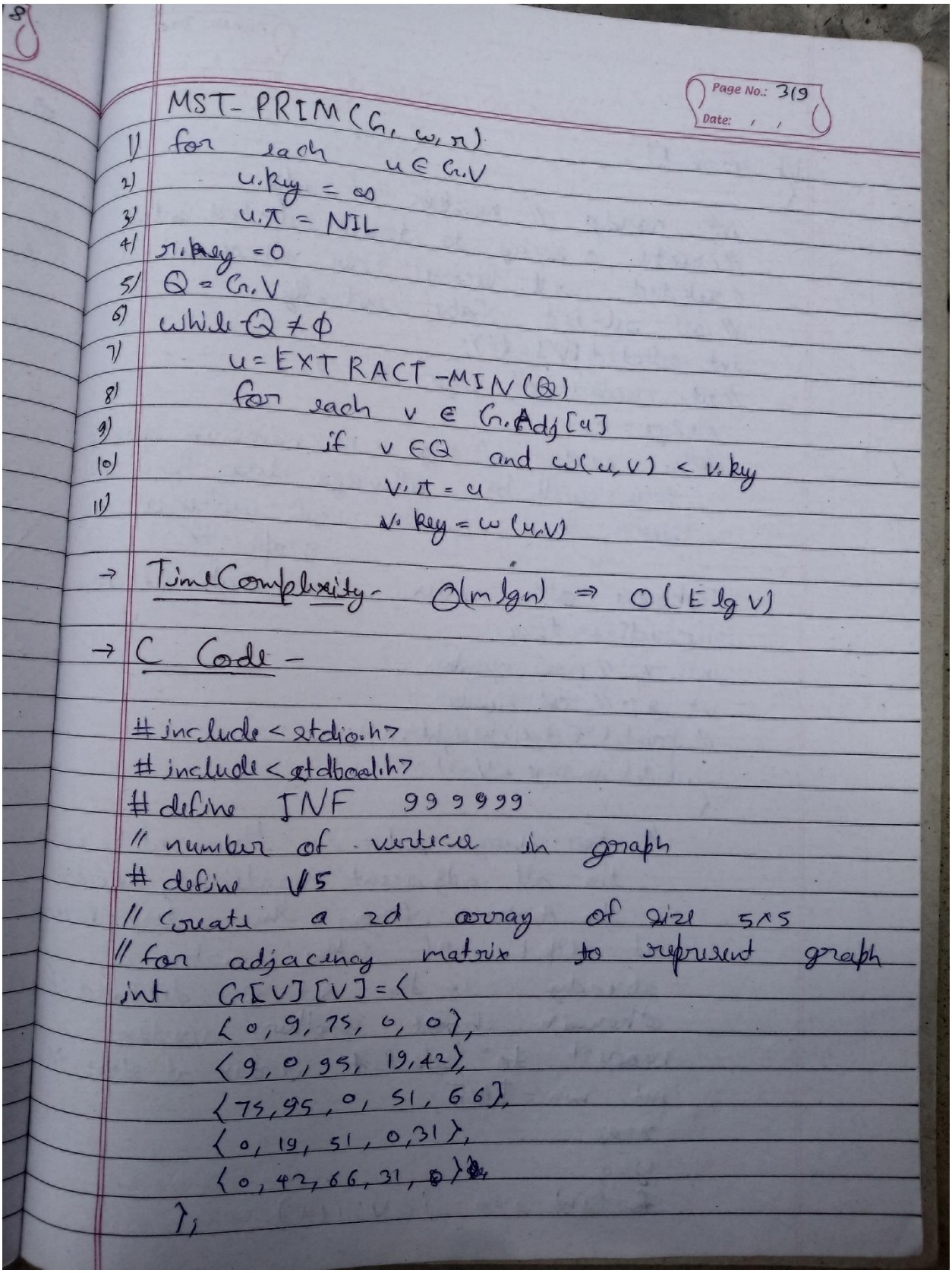
```
class Kruskal:
    def __init__(self, n):
        self.V = n
        self.graph = []
    def add_edge(self, u, v, w):
        self.graph.append([u, v, w])
    def find(self, parent, i):
        if parent[i] = i:
            return i,
        return self.find(parent, parent[i])
    def kruskal_algo(self):
        result = []
        self.graph = sorted(self.graph, key = lambda
                                item : item[2])
        parent = [k for k in range(self.V)]
        i, e = 0, 0
        while e < self.V-1:
            u, v, w = self.graph[i]
            x = self.find(parent, u)
            y = self.find(parent, v)
            if x != y:
                result.append([u, v, w])
                parent[y] = x
                e += 1
            i += 1
        print(result)
g = Kruskal(6)
g.add_edge(0, 1, 4)
```

```
g.add_edge(1,0,4)
g.add_edge(0,2,4)
g.add_edge(2,0,4)
g.add_edge(1,2,2)
g.add_edge(2,1,2)
g.add_edge(2,3,3)
g.add_edge(3,2,3)
g.add_edge(2,5,2)
g.add_edge(5,2,2)
g.add_edge(2,4,4)
g.add_edge(4,2,4)
g.add_edge(3,4,3)
g.add_edge(4,3,3)
g.add_edge(4,5,3)
g.add_edge(5,4,3)
g.kruskals_algo()
```

Prim's Algorithm- (i.e. Greedy Approach)
It use auxillary DS-
Min Heap / priority queue



π = predecessor.



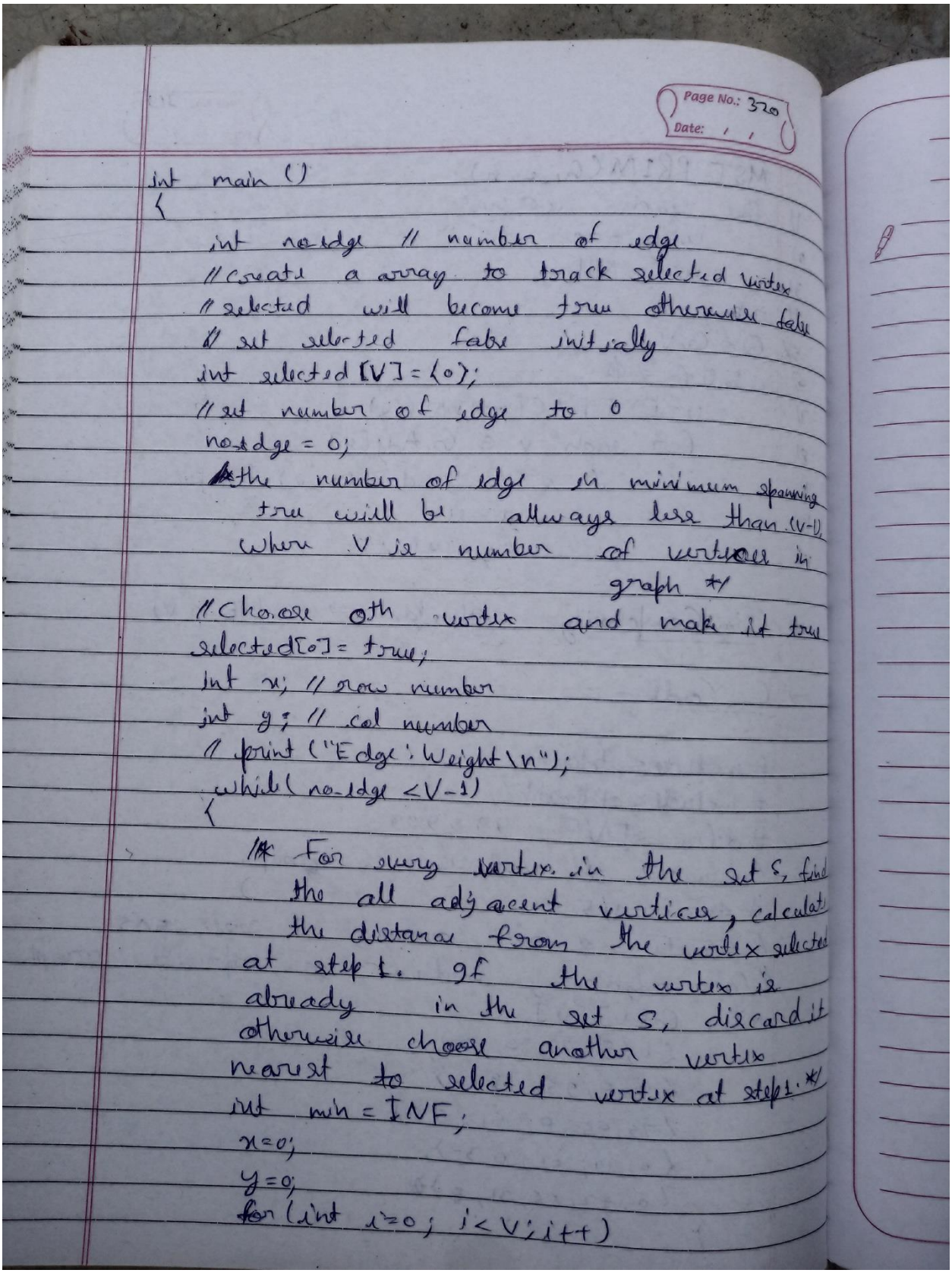
Page No.: 319
Date: / /

```
MST-PRIM(G, w, r)
1) for each u ∈ G.V
2)     u.key = ∞
3)     u.π = NIL
4) r.key = 0
5) Q = G.V
6) while Q ≠ ϕ
7)     u = EXTRACT-MIN(Q)
8)     for each v ∈ G.Adj[u]
9)         if v ∈ Q and w(u,v) < v.key
10)            v.π = u
11)            v.key = w(u,v)
```

→ Time Complexity - $O(m \lg n) \Rightarrow O(E \lg V)$

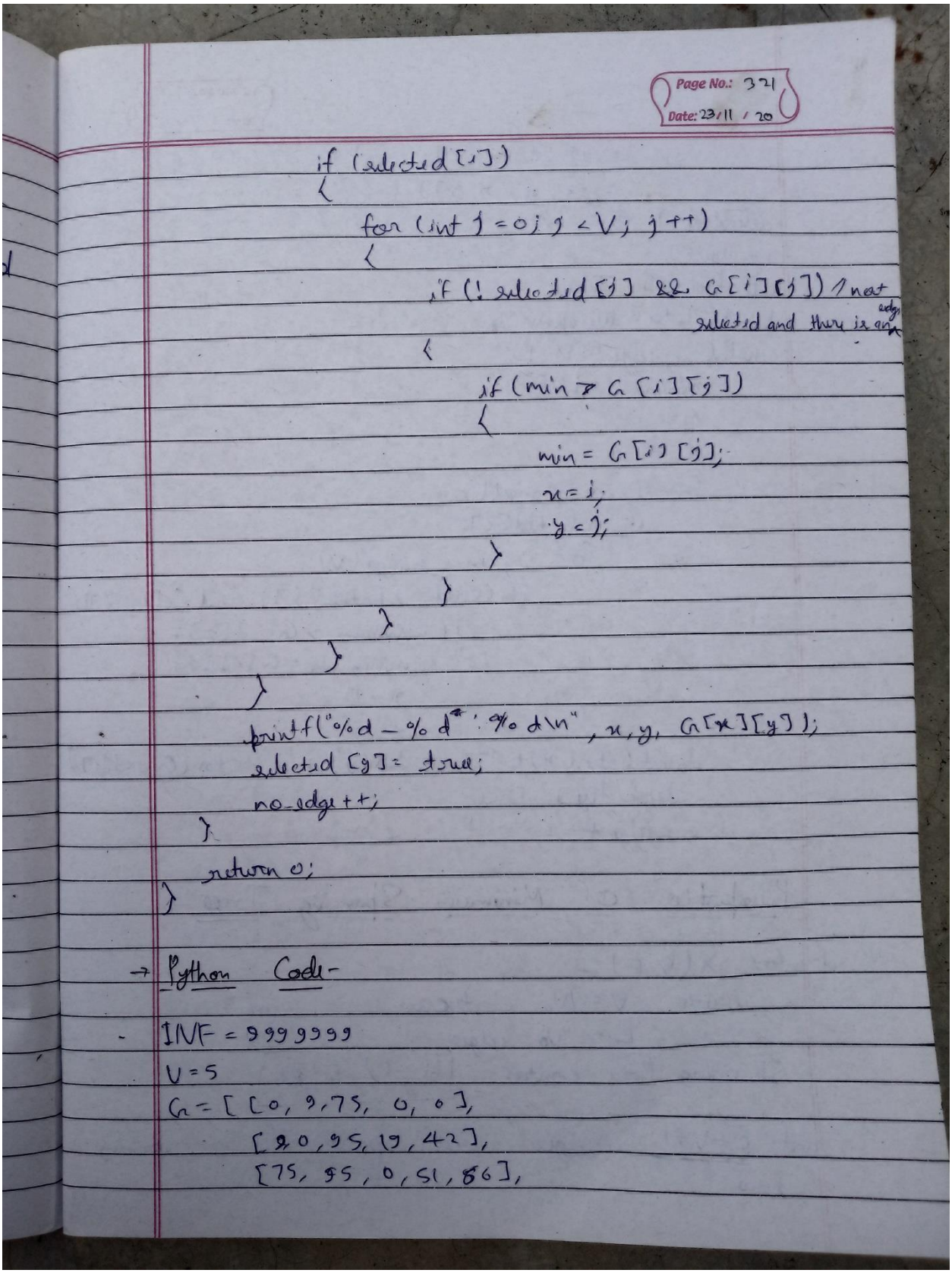
→ C Code -

```
#include <stdio.h>
#include <stdbool.h>
#define INF 999999
// number of vertices in graph
#define V 5
// Create a 2d array of size 5x5
// for adjacency matrix to represent graph
int G[V][V] = {
    {0, 9, 75, 0, 0},
    {9, 0, 95, 19, 42},
    {75, 95, 0, 51, 66},
    {0, 19, 51, 0, 31},
    {0, 42, 66, 31, 0}
};
```



Page No.: 320
Date: / /

```
int main ()
{
    int no_edge // number of edge
    // create a array to track selected vertex
    // selected will become true otherwise false
    // set selected false initially
    int selected[V] = {0};
    // set number of edge to 0
    no_edge = 0;
    /* the number of edge in minimum spanning
       tree will be always less than (V-1)
       where V is number of vertices in
                                    graph */
    // Choose 0th vertex and make it true
    selected[0] = true;
    int x; // row number
    int y; // col number
    // print ("Edge : Weight \n");
    while( no_edge < V-1)
    {
        /* For every vertex in the set S, find
           the all adjacent vertices, calculate
           the distance from the vertex selected
           at step 1. If the vertex is
           already in the set S, discard it
           otherwise choose another vertex
           nearest to selected vertex at step 1. */
        int min = INF;
        x=0;
        y=0;
        for (int i=0; i<V; i++)
```

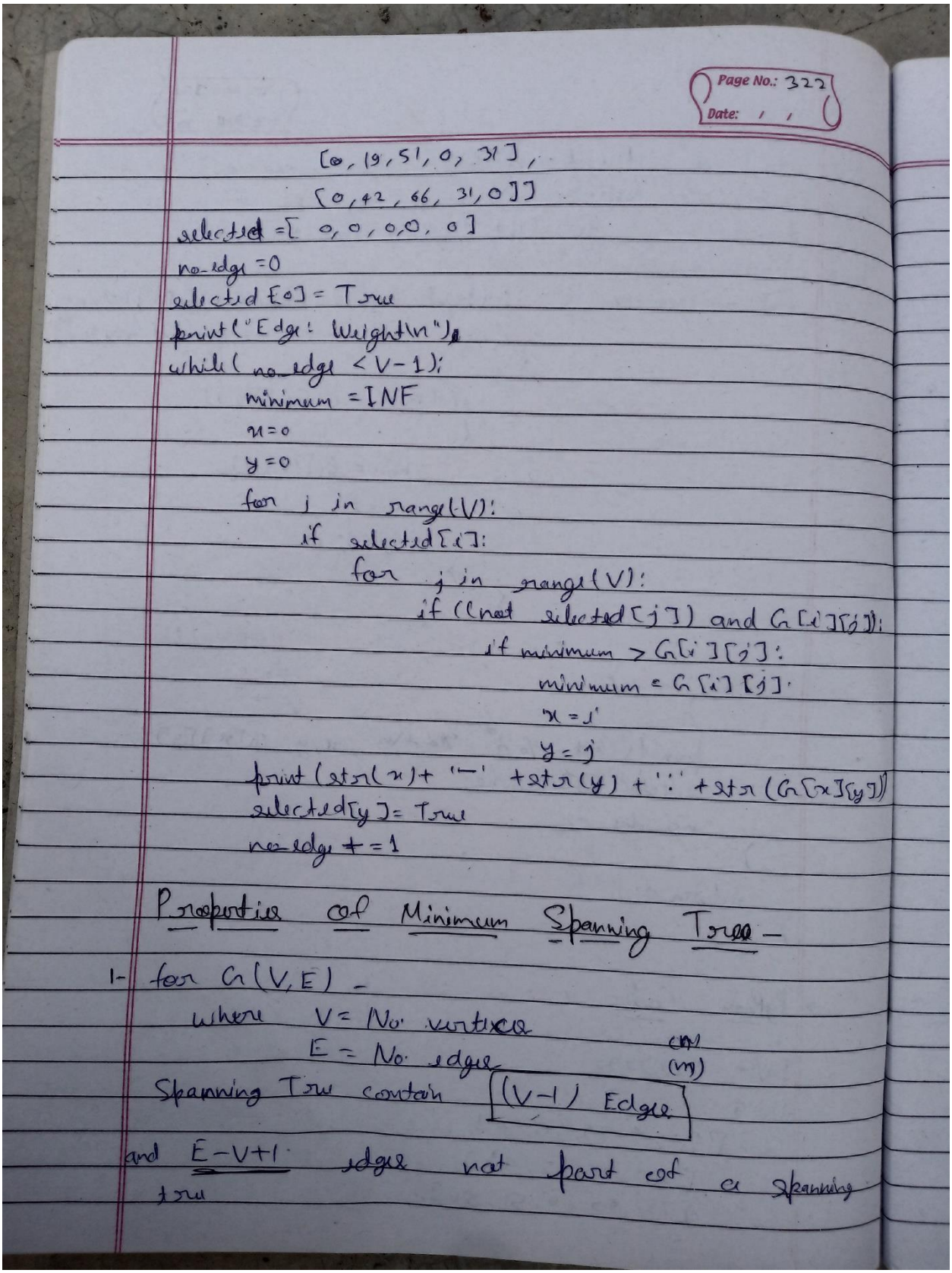


Page No.: 321
Date: 23/11/20

```
if (selected[i])
{
    for (int j=0; j<V; j++)
    {
        if (!selected[j] && G[i][j]) // not selected and there is an edge
        {
            if (min > G[i][j])
            {
                min = G[i][j];
                x = i;
                y = j;
            }
        }
    }
}
printf("%d - %d : %d\n", x, y, G[x][y]);
selected[y] = true;
no_edge++;
}
return 0;
}
```

→ Python Code-

```
INF = 9999999
V = 5
G = [[0, 9, 75, 0, 0],
     [9, 0, 95, 19, 42],
     [75, 95, 0, 51, 66],
```



```
        [0, 19, 51, 0, 31],
        [0, 42, 66, 31, 0]]
selected =[ 0, 0, 0, 0, 0]
no_edge =0
selected[0] = True
print("Edge : Weight\n")
while( no_edge < V-1):
    minimum =INF
    x=0
    y=0
    for i in range(V):
        if selected[i]:
            for j in range(V):
                if ((not selected[j]) and G[i][j]):
                    if minimum > G[i][j]:
                        minimum = G[i][j].
                        x = i
                        y = j
    print(str(x)+ '-' +str(y) + ':' +str(G[x][y]))
    selected[y]= True
    no_edge += 1
```

Properties of Minimum Spanning Tree -

1- for G(V, E) -

where V = No. vertices (n)
E = No. edges (m)

Spanning Tree contain (V-1) Edges

and E-V+1 edges not part of a spanning tree

Page No.: 323
Date: / /

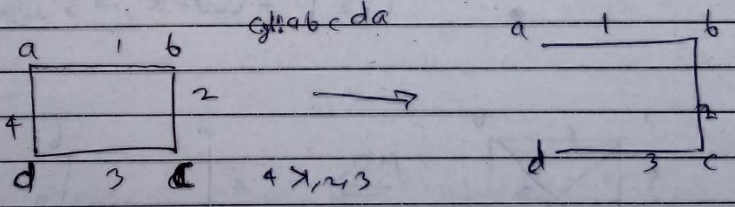
2- If you add any edge to a ST, the new graph becomes cyclic. Hence spanning Tree is maximally acyclic

3- Every spanning Tree is minimally connected that is removal of an edge will disconnect the graph.

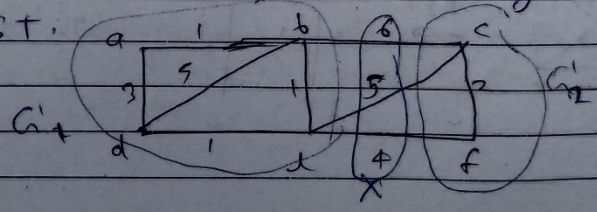
4- There may be several MST of same weight.

5- If each edge has a distinct weight then you will have exactly one unique MST.

6- Cyclic Property - For any cycle C in Graph, if the edge weight (e) is larger than all other edges in C, then edge cannot belong to MST.



7- Cut property - For any cut C, if the weight of an edge in the Cut-set is strictly smaller all other weights of edges in the cut-Set then, this edge must be part of MST.

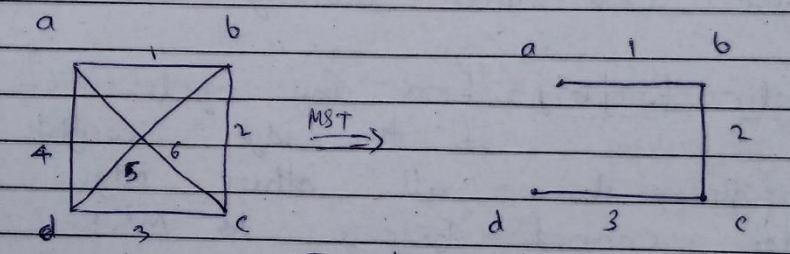


Page No.: 324
Date: / /

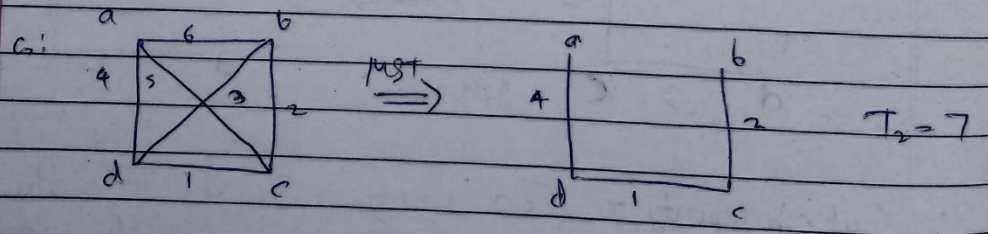
8- Min-Cost-edge- If the min-weighted edge in G is unique then this edge ∈ MST

Complete Graph- edge (u,v) exists ∀ u,v ∈ G
Eg ⊠ △ —

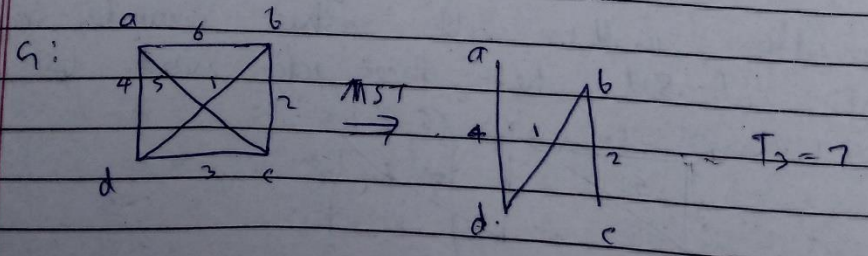
Q- Let G be a complete undirected graph on 4 vertices, having 6 edges with weights being 1, 2, 3, 4, 5 and 6. The maximum possible weight that a minimum weight spanning tree of G can have is. ?



$T_1 = 6$ | 4 Vertices, → 3 edge



$T_2 = 7$

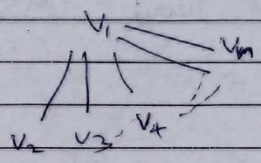


$T_3 = 7$

Page No.: 325
Date: / /

Q- What is the maximum number of undirected graphs with n-vertices?

Graph of n vertices → $m = \frac{n(n-1)}{2}$



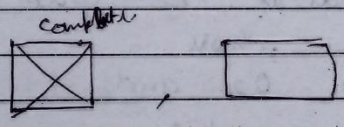
$V_1 \rightarrow (n-1)$ edges

$V_2 \rightarrow (n-1)$ edges

⋮

$V_n \rightarrow 0$ edges

$\frac{n(n-1)}{2} = m$



In Graph any edge
→ Present
→ Absent

No. edge possibility → 2^m = no. undirected Graphs using n vertices.

$= 2^{\frac{n(n-1)}{2}}$

Simple Graph- Not any multiple edge for any vertex set (u, v) and Not contain any self loop.

Q- $G = (V, E)$ is an undirected simple graph in which each edge has a distinct weight, and particular edge of G. which of the following statements about the minimum spanning trees (MSTs) of G is/are TRUE

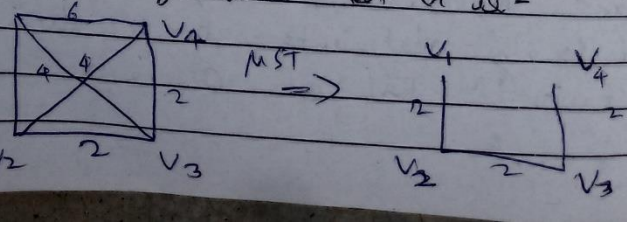
Page No.: 376
Date: / /

(i) If e is the lightest edge of some cycle in G, then every MST of G includes e.
(ii) If e is the heaviest edge of some cycle in G then every MST of G excludes e

(A) I only
(B) II only (✓)
(C) both I and II
(D) neither I nor II

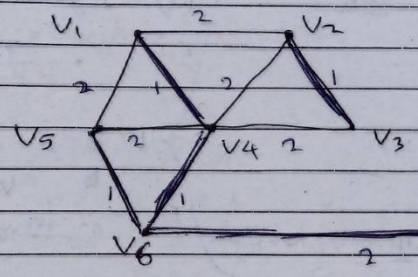
Q- An undirected graph G has n nodes. Its adjacency matrix is given by an n×n square matrix where
(i) diagonal elements are 0's and
(ii) non-diagonal elements are 1's
which one of the following is TRUE?
(A) Graph G has no minimum spanning tree (MST)
(B) Graph G has unique MST of cost n-1
(C) Graph G has multiple distinct MST, each of cost n-1. (✓)
(D) Graph G has multiple spanning trees of different costs.

Q- Consider a weighted complete graph G on the vertex set $\{v_1, v_2, \ldots v_n\}$ such that weight of the edge (v_i, v_j) is $2|i-j|$. The weight of a minimum spanning tree of G is -
(A) n-1
(B) 2n-2 (✓)
(C) nC_2
(D) 2



Page No.: 327
Date: / /

Q- The number of distinct minimum spanning trees for the weighted graph below is - 6



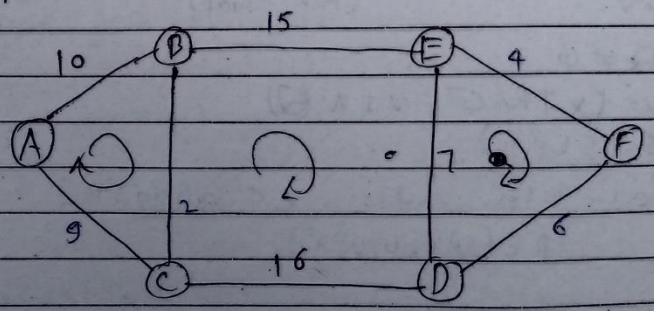
Using Cut property

- means must be that edge part of MST.

3 ways → (V2, V3)
2 ways → (V8 V9)

Total way = Total No. MST = 3×2 = 6

Q- The graph shown below has 8 edges with distinct integer edge weights. The minimum spanning tree (MST) is weight 36 and contains the edges: {(A, C), (B, C), (B, E), (E, F), (D, F)} The edge-weights of only those edges which are in the MST are given in the figure shown below. The minimum possible sum of weights of all 8 edges of this graph is 69

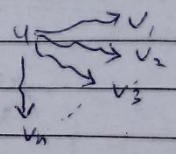


Ans Cycle Property — {36 + 10 + 7 + 16} = 69.

Page No.: 328
Date: 24/11/20

Graph- Shortest Paths-

1- Single Source Sorted Path-



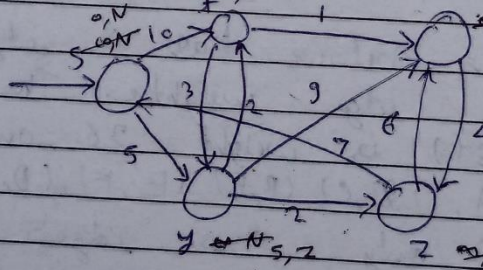
(a) edge weights- +ve
Eg u ——— v

(b) edge weight may be +ve or -ve
Eg: A —+ve→ B —-ve→ C

2- All pair Shortest Path - $V_i \rightsquigarrow V_j$

• SSSP- DIJKSTRA'S ALGORITM-

apply on Graph only if all edge weights are non-negative.



Auxilary DS-
use Priority Queue/ min heap.

```
DIJKSTRA(G, w, s)
1) INITIALIZE-SINGLE-SOURCE(G, s)
2) S = ∅
3) Q = G.V                    (min-heap)
4) while Q ≠ ∅
5)     u = EXTRACT-MIN(Q)
6)     S = S ∪ {u}
7)     for each vertex v ∈ G.Adj[u]
8)         RELAX(u, v, w)
```

Page No.: 329
Date: / /

INITIALIZE-SINGLE-SOURCE(G, s) — O(V)

1) for each vertex $v \in G.V$
2) $v.d = \infty$
3) $v.\pi = NIL$
4) $s.d = 0$

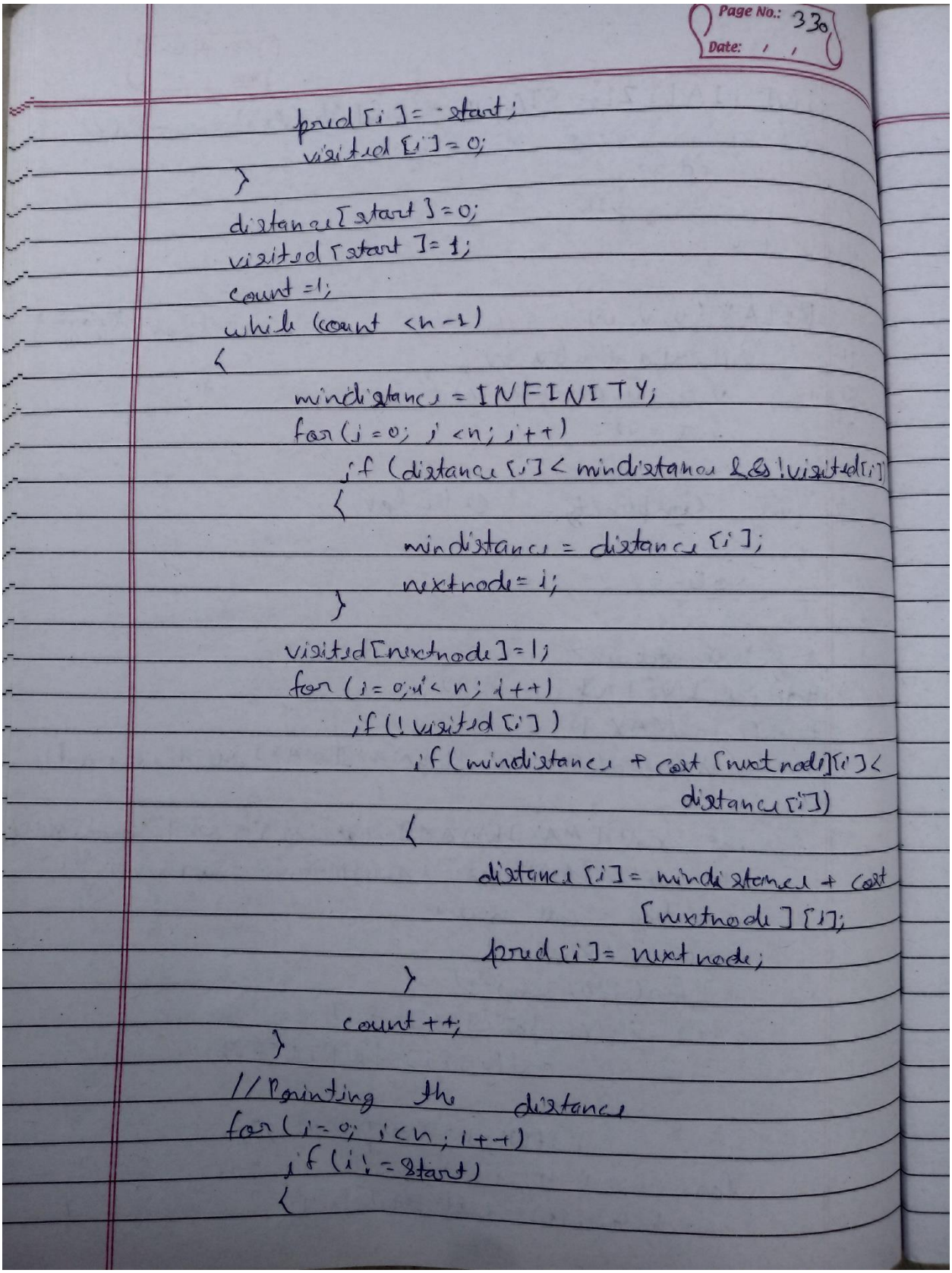
RELAX(u, v, w) — O(lgV) (Priority queue)

1) if $v.d > u.d + w(u,v)$
2) $v.d = u.d + w(u,v)$
3) $v.\pi = u$

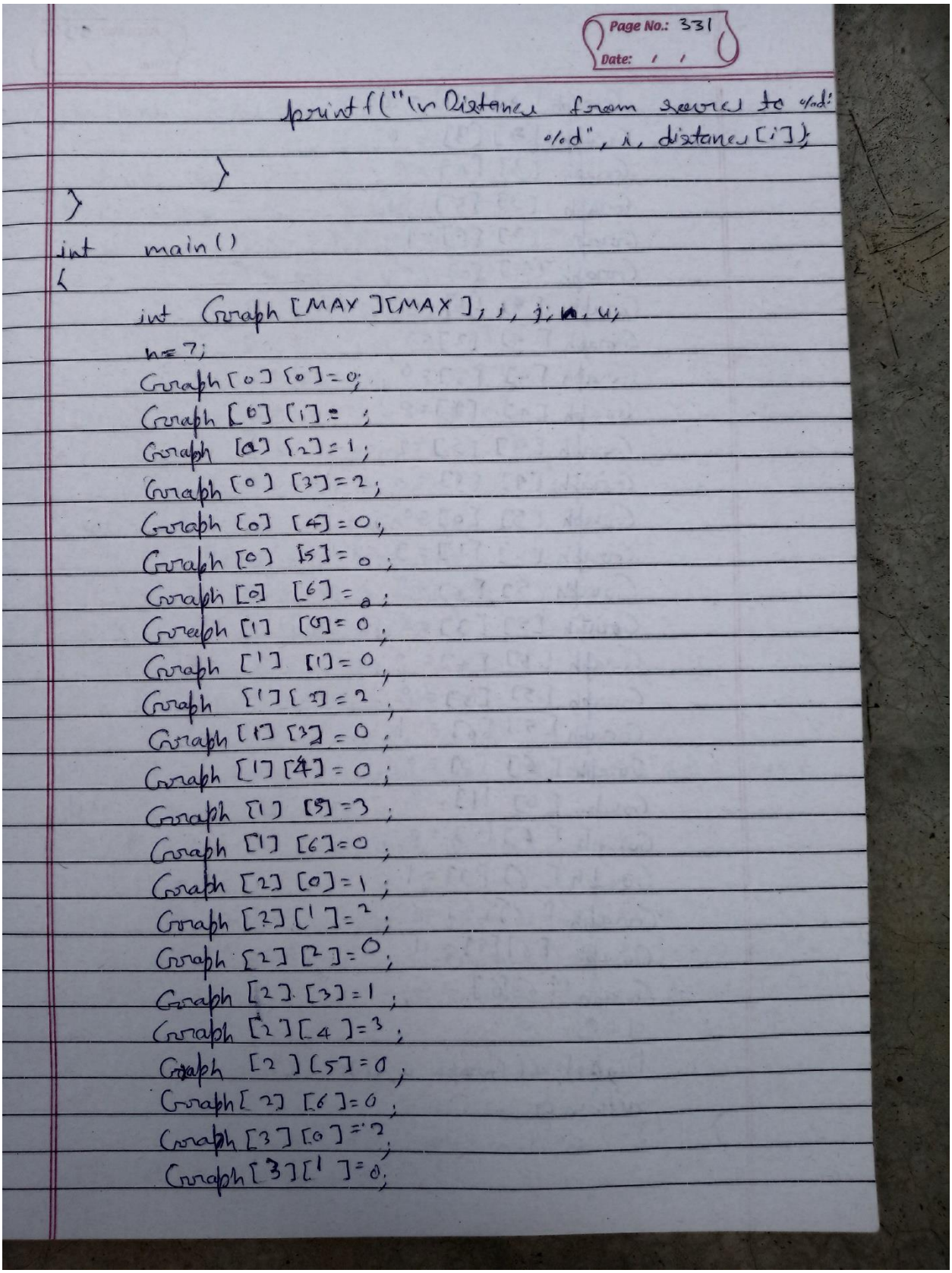
→ Time Complexity - O(E lgV)

C Code-

```
#include <stdio.h>
#define INFINITY 9999
#define MAX 10
void Dijkstra(int Graph[MAX][MAX], int n, int start);
{
    int cost[MAX][MAX], distance[MAX], pred[MAX];
    int visited[MAX], count, mindistance, nextnode, i, j;
    // Creating cost matrix
    for(i=0; i<n; i++)
        for(j=0; j<n; j++)
            if(Graph[i][j]==0)
                cost[i][j]=INFINITY;
            else
                cost[i][j] = Graph[i][j];
    for(i=0; i<n; i++)
        distance[i]= cost[start][i];
```

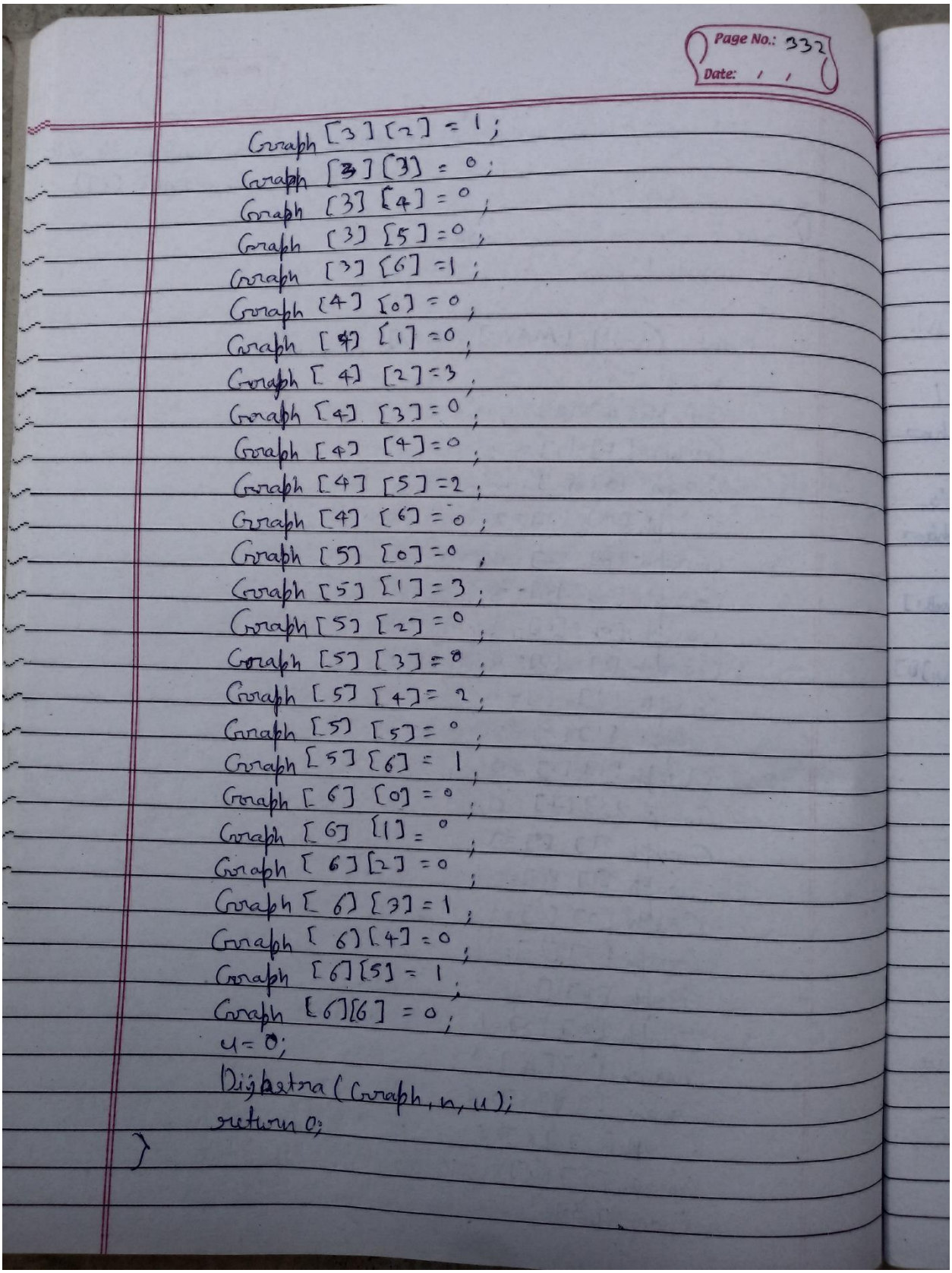


```
Page No.: 330
Date: / /

        pred[i] = start;
        visited[i] = 0;
    }
    distance[start] = 0;
    visited[start] = 1;
    count = 1;
    while (count < n-1)
    {
        mindistance = INFINITY;
        for (i = 0; i < n; i++)
            if (distance[i] < mindistance && !visited[i])
            {
                mindistance = distance[i];
                nextnode = i;
            }
        visited[nextnode] = 1;
        for (i = 0; i < n; i++)
            if (!visited[i])
                if (mindistance + cost[nextnode][i] <
                                        distance[i])
                {
                    distance[i] = mindistance + cost
                                [nextnode][i];
                    pred[i] = nextnode;
                }
        count++;
    }
    // Printing the distance
    for (i = 0; i < n; i++)
        if (i != start)
        {
```



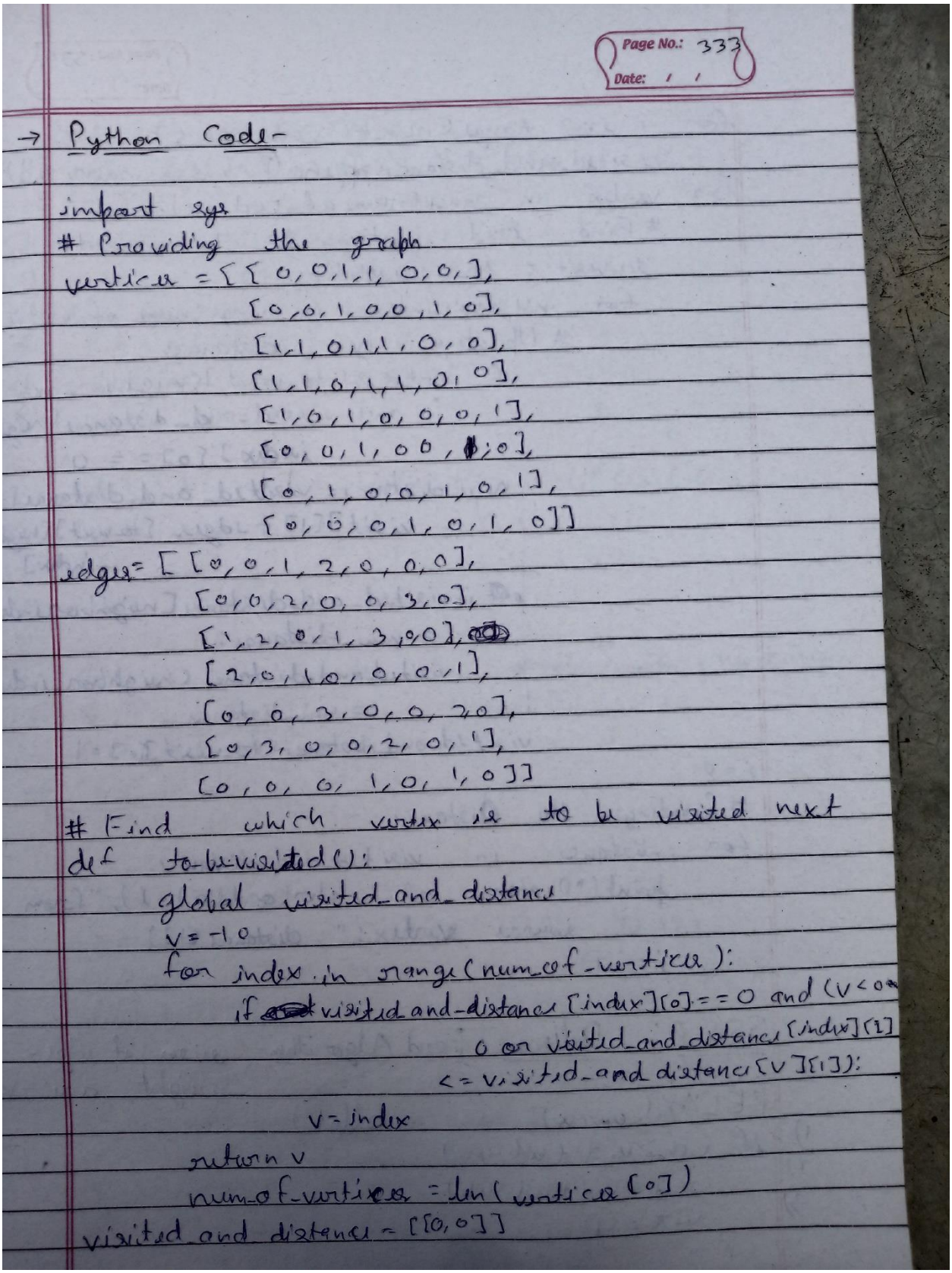
Page No.: 331
Date: / /

```
        printf("\n Distance from source to %d: %d", i, distance[i]);
    }
}
int main()
{
    int Graph[MAX][MAX], i, j, n, u;
    n=7;
    Graph[0][0]=0;
    Graph[0][1]= ;
    Graph[0][2]=1;
    Graph[0][3]=2;
    Graph[0][4]=0;
    Graph[0][5]=0;
    Graph[0][6]=0;
    Graph[1][0]=0;
    Graph[1][1]=0;
    Graph[1][2]=2;
    Graph[1][3]=0;
    Graph[1][4]=0;
    Graph[1][5]=3;
    Graph[1][6]=0;
    Graph[2][0]=1;
    Graph[2][1]=2;
    Graph[2][2]=0;
    Graph[2][3]=1;
    Graph[2][4]=3;
    Graph[2][5]=0;
    Graph[2][6]=0;
    Graph[3][0]=2;
    Graph[3][1]=0;
```



Page No.: 332
Date: / /

```
Graph [3] [2] = 1;
Graph [3] [3] = 0;
Graph [3] [4] = 0;
Graph [3] [5] = 0;
Graph [3] [6] = 1;
Graph [4] [0] = 0;
Graph [4] [1] = 0;
Graph [4] [2] = 3;
Graph [4] [3] = 0;
Graph [4] [4] = 0;
Graph [4] [5] = 2;
Graph [4] [6] = 0;
Graph [5] [0] = 0;
Graph [5] [1] = 3;
Graph [5] [2] = 0;
Graph [5] [3] = 0;
Graph [5] [4] = 2;
Graph [5] [5] = 0;
Graph [5] [6] = 1;
Graph [6] [0] = 0;
Graph [6] [1] = 0;
Graph [6] [2] = 0;
Graph [6] [3] = 1;
Graph [6] [4] = 0;
Graph [6] [5] = 1;
Graph [6] [6] = 0;
u = 0;
Dijkstra (Graph, n, u);
return 0;
}
```



Page No.: 333
Date: / /

→ Python Code-

```
import sys
# Providing the graph
vertices = [[0,0,1,1,0,0,],
            [0,0,1,0,0,1,0],
            [1,1,0,1,1,0,0],
            [1,1,0,1,1,0,0],
            [1,0,1,0,0,0,1],
            [0,0,1,0,0,0,0],
            [0,1,0,0,1,0,1],
            [0,0,0,1,0,1,0]]
edges= [[0,0,1,2,0,0,0],
        [0,0,2,0,0,3,0],
        [1,2,0,1,3,0,0],
        [2,0,1,0,0,0,1],
        [0,0,3,0,0,2,0],
        [0,3,0,0,2,0,1],
        [0,0,0,1,0,1,0]]
# Find which vertex is to be visited next
def to_be_visited():
    global visited_and_distance
    v=-10
    for index in range(num_of_vertices):
        if visited_and_distance[index][0]==0 and (v<0
                0 or visited_and_distance[index][1]
                <=visited_and_distance[v][1]):
            v=index
    return v
num_of_vertices = len(vertices[0])
visited_and_distance=[[0,0]]
```

Page No.: 334
Date: / /

```
for i in range(num_of_vertices - 1):
    visited_and_distance.append([0, sys.maxsize])
for vertex in range(num_of_vertices):
    # Find next vertex to be visited.
    to_visit = to_be_visited()
    for neighbor_index in range(num_of_vertices):
        # Updating new distances
        if vertices[to_visit][neighbor_index]!=0
                and visited_and_distance[neighbor
                    index][0] == 0:
            new_distance = visited_and_distance[to_
                visit][1] + edges[to_visit][neighbor
                    _index]

            if visited_and_distance[neighbor_index][1]
                > new_distance:
                visited_and_distance[neighbor_index][1]
                    = new_distance
            visited_and_distance[to_visit][0] = 1
i = 0
# Printing the distances
for distance in visited_and_distance:
    print("Distance of ", chr(ord('a') + i), "from
        source vertex:", distance[1])
    i = i + 1
```

SSSP :- Bellman - Ford Algorithm - even if edge-weight are -ve

RELAX(u, v, w) - O(lgv)

```
1) if v.d > u.d + w(u,v)
2)     v.d = u.d + w(u,v)
3)     v.π = u
```

Page No.: 335
Date: / /

BELLMAN-FORD (G, w, s)
1) INITIALIZE-SINGLE-SOURCE (G, s)
2) for i = 1 to |G.V| - 1
3) for each edge (u, v) ∈ G.E
4) RELAX (u, v, w)
5) for each edge (u, v) ∈ G.E
6) if v.d > u.d + w(u, v)
7) return FALSE
8) return TRUE

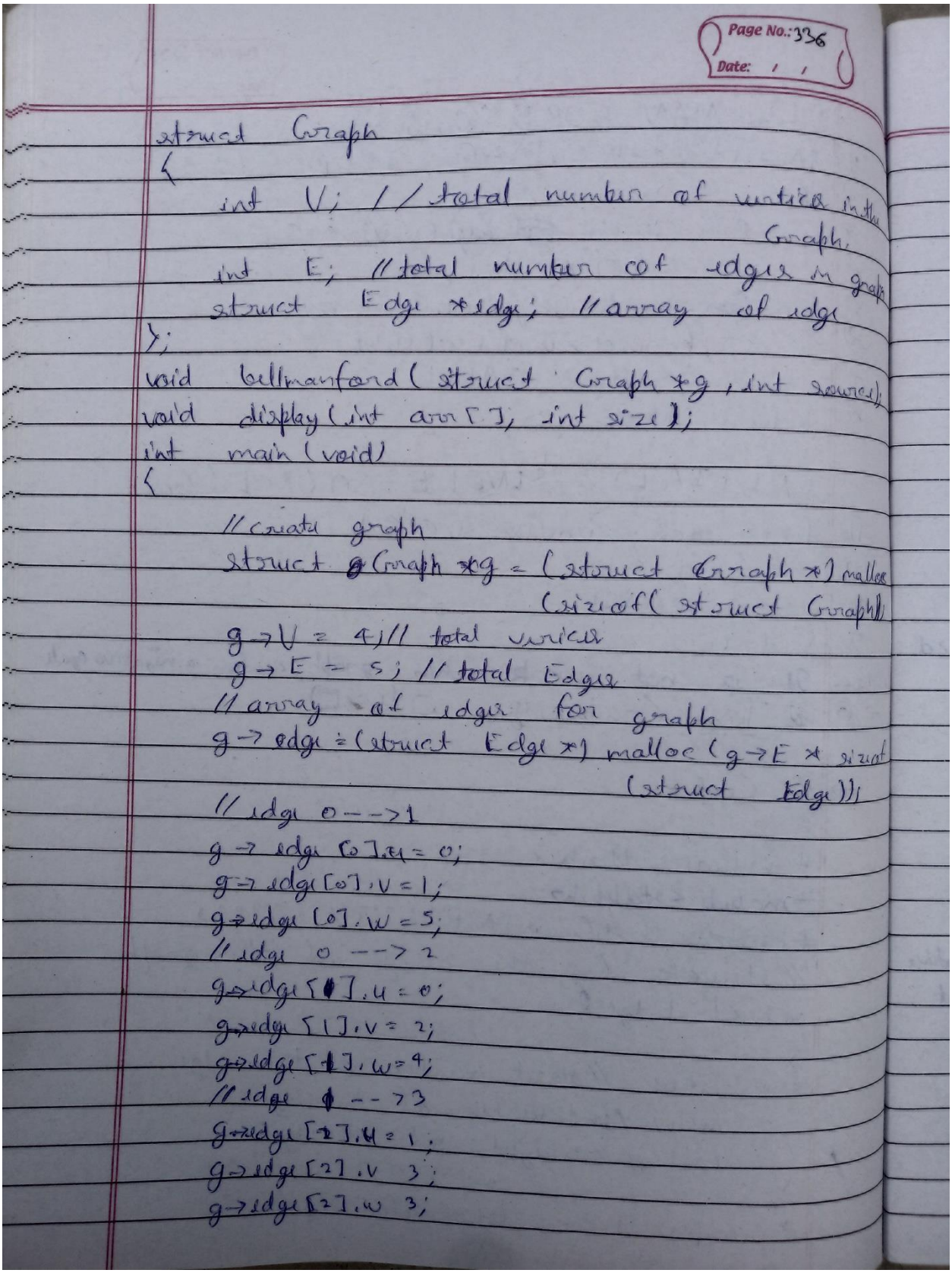
INITIALIZ-SINGLE-SOURCE (G, s)
1) for each vertex v ∈ G.V
2) v.d = ∞
3) v.π = NIL
4) s.d = 0

Note:- It is not work when graph contain a negative cycle.

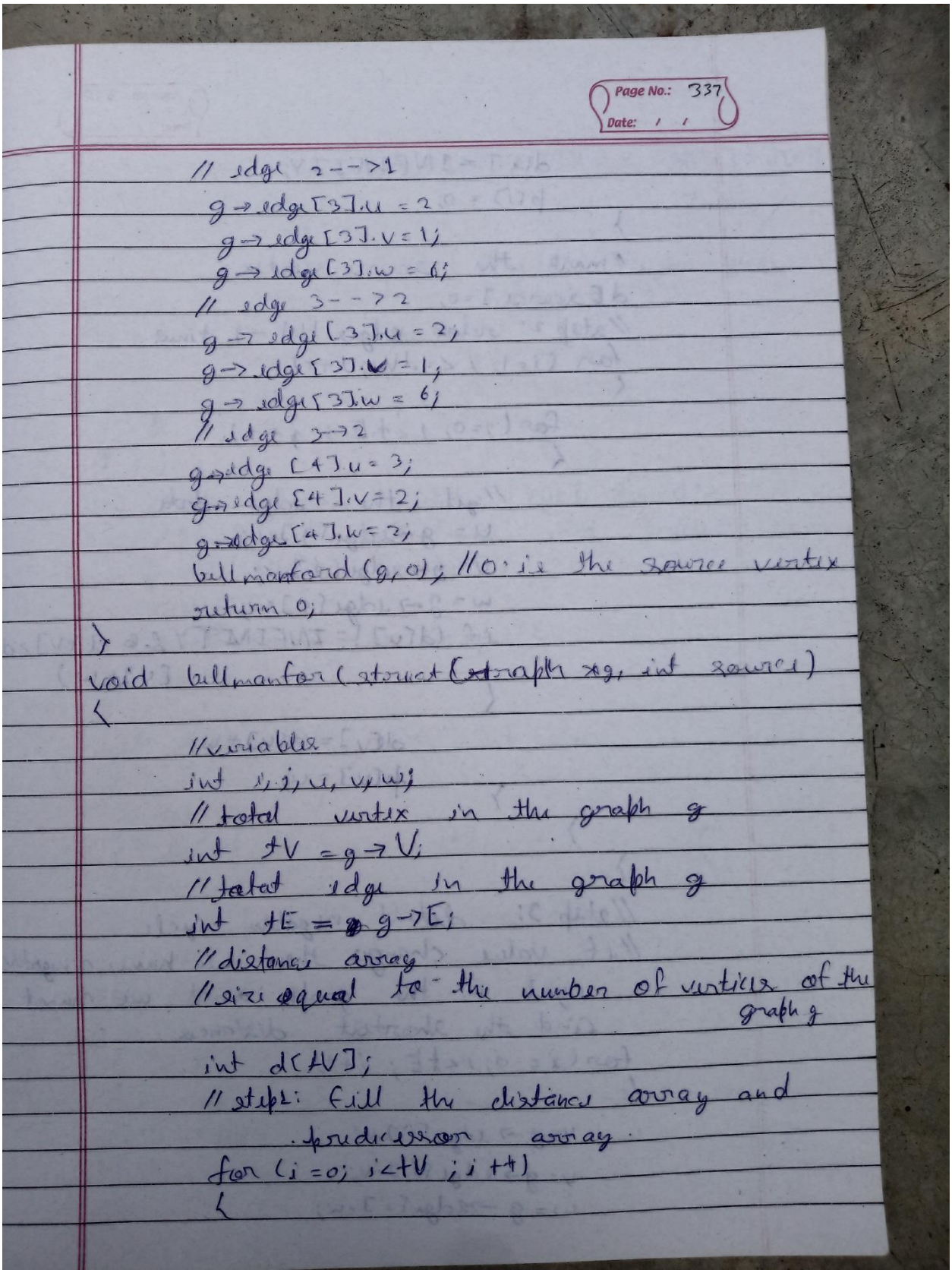
→ Time Complexity- O(V*E)

→ C Code-

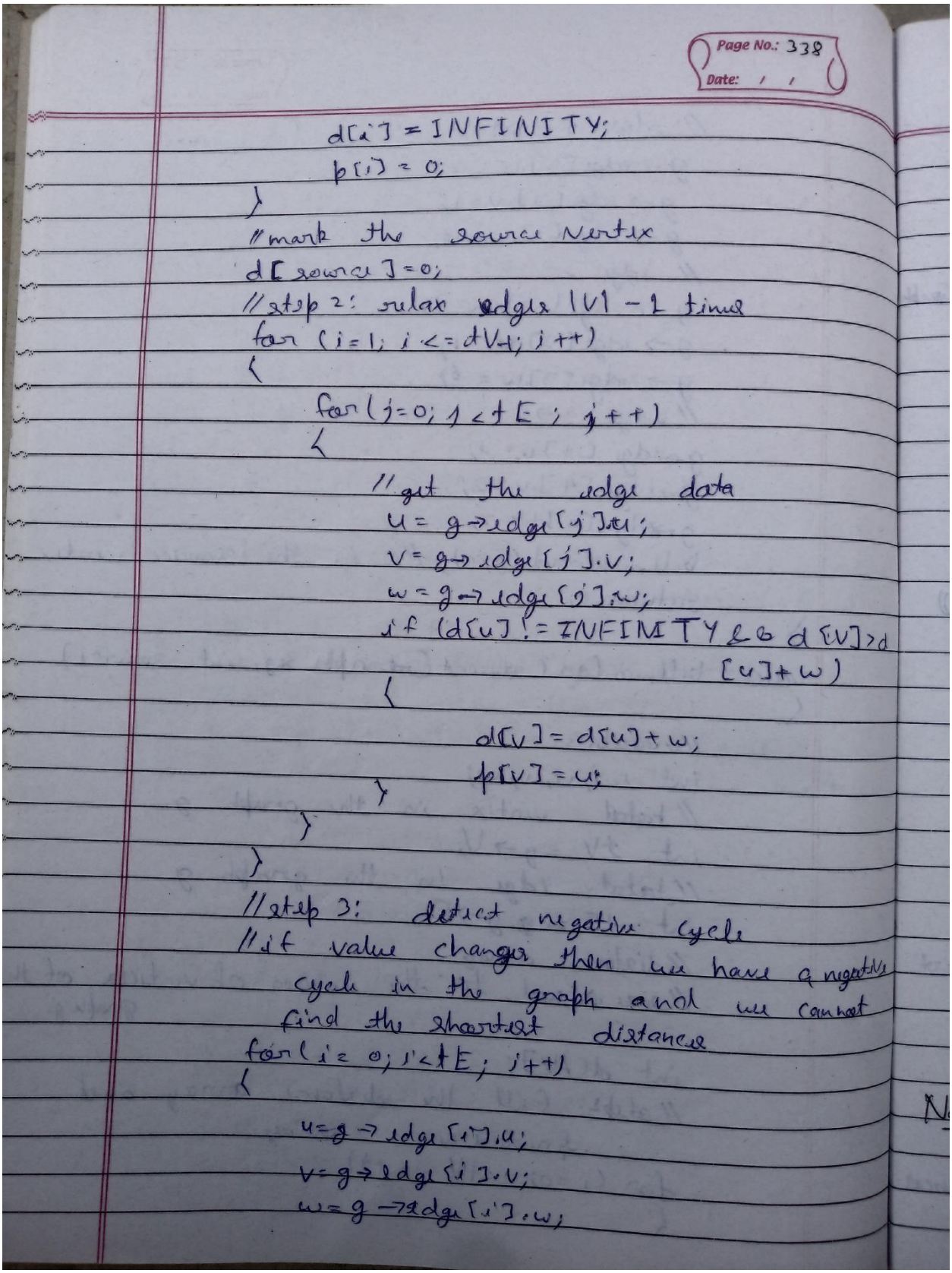
```
#include <stdio.h>
#include <stdlib.h>
#define INFINITY 99999
// struct for the edge of the graph
struct Edge {
    int u; // start vertex of the edge
    int v; // end vertex of the edge
    int w; // weight of the edge
};
// Graph - it consists of edges
```



```
struct Graph
{
    int V; // total number of vertices in the
                                    Graph.
    int E; // total number of edges in graph
    struct Edge *edge; // array of edge
};
void bellmanford( struct Graph *g, int source);
void display(int arr[], int size);
int main(void)
{
    // create graph
    struct Graph *g = (struct Graph *) malloc
                        (sizeof( struct Graph));
    g->V = 4; // total vertex
    g->E = 5; // total Edges
    // array of edge for graph
    g->edge = (struct Edge *) malloc (g->E * sizeof
                        (struct Edge));
    // edge 0 --> 1
    g->edge[0].u = 0;
    g->edge[0].v = 1;
    g->edge[0].w = 5;
    // edge 0 --> 2
    g->edge[1].u = 0;
    g->edge[1].v = 2;
    g->edge[1].w = 4;
    // edge 1 --> 3
    g->edge[2].u = 1;
    g->edge[2].v = 3;
    g->edge[2].w = 3;
```



```
// edge 2 --> 1
g->edge[3].u = 2
g->edge[3].v = 1;
g->edge[3].w = 6;
// edge 3 --> 2
g->edge[3].u = 2;
g->edge[3].v = 1;
g->edge[3].w = 6;
// edge 2 --> 2
g->edge[4].u = 3;
g->edge[4].v = 2;
g->edge[4].w = 2;
bellmanford(g, 0); // 0: is the source vertex
return 0;
}
void bellmanford(struct Graph *g, int source)
{
// variables
int i, j, u, v, w;
// total vertex in the graph g
int tV = g->V;
// total edge in the graph g
int tE = g->E;
// distance array
// size equal to the number of vertices of the
                                    graph g
int d[tV];
// step1: fill the distance array and
        predecessor array.
for (i = 0; i < tV ; i++)
{
```



Page No.: 338
Date: / /

```
        d[i] = INFINITY;
        p[i] = 0;
    }
    // mark the source vertex
    d[source] = 0;
    // step 2: relax edges |V| - 1 times
    for (i = 1; i <= dV-1; i++)
    {
        for (j = 0; j < tE; j++)
        {
            // get the edge data
            u = g->edge[j].u;
            v = g->edge[j].v;
            w = g->edge[j].w;
            if (d[u] != INFINITY && d[v] > d
                                    [u] + w)
            {
                d[v] = d[u] + w;
                p[v] = u;
            }
        }
    }
    // step 3: detect negative cycle
    // if value changes then we have a negative
       cycle in the graph and we cannot
       find the shortest distance
    for (i = 0; i < tE; i++)
    {
        u = g->edge[i].u;
        v = g->edge[i].v;
        w = g->edge[i].w;
```

Page No.: 339
Date: / /

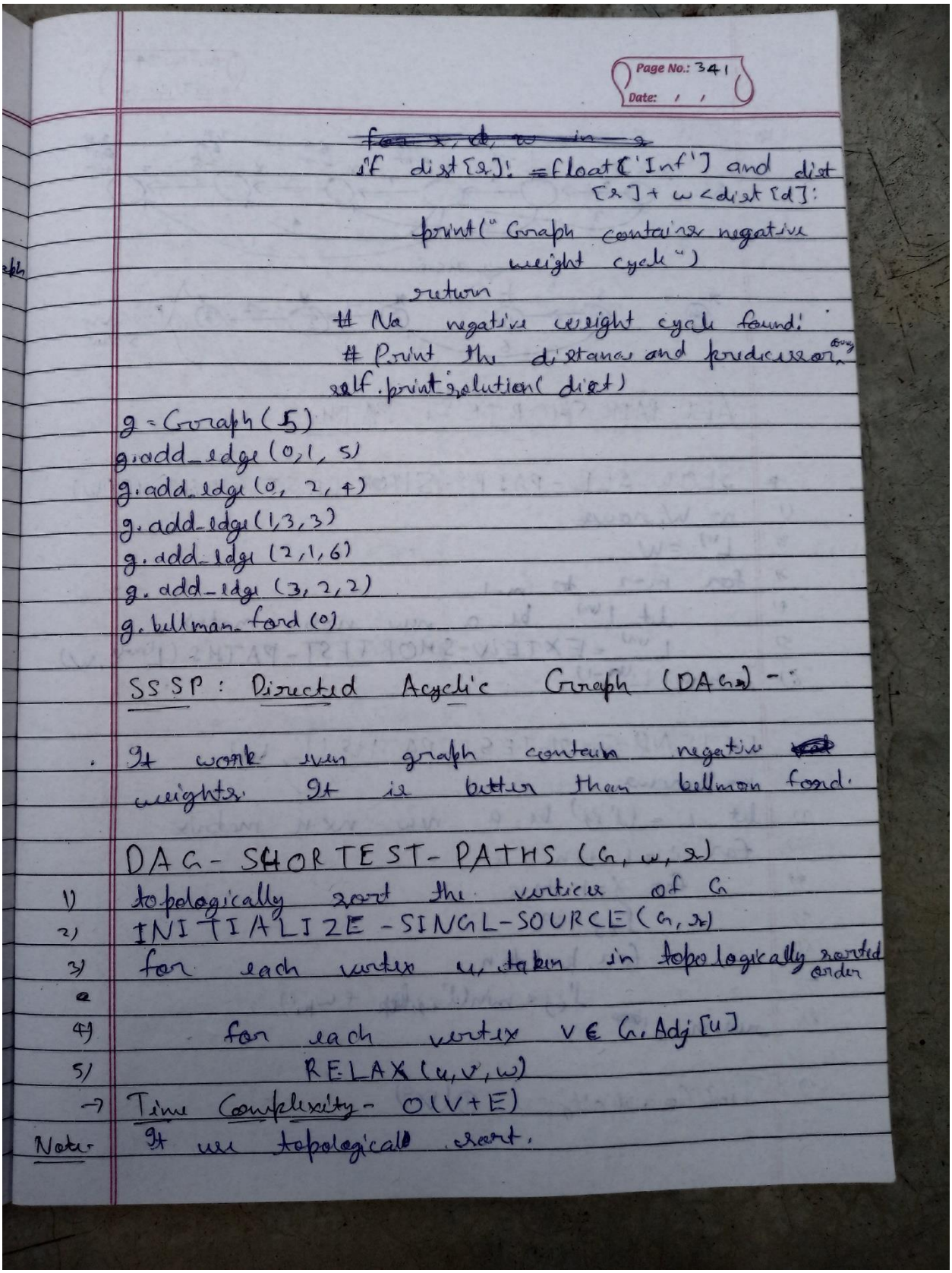
```
            if(d[u] != INFINITY && d[v] > d[u]
                                        + w)
            {
                printf("Negative weight cycle
                        detected!\n");
                return;
            }
        }
    }
}
// No negative weight cycle found!
// Print the distance and predecessor array
printf("Distance array: ");
display(d, tV);
printf("Predecessor array: ");
display(p, tV);
}
void display(int arr[], int size)
{
    int i;
    for(i=0; i<size; i++)
    {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
```

Note:- If your graph not contain negative cycle then Dijkstra is better than Bellman ford.

Page No.: 340
Date: / /

→ Python Code -

```
class Graph:
    def __init__(self, vertices):
        self.V = vertices # total Number of vertices in Graph
        self.graph = [] # Array of edges
    # Add edges
    def add_edge(self, s, d, w):
        self.graph.append([s, d, w])
    # Print the solution
    def print_solution(self, dist):
        print("Vertex Distance from Source")
        for i in range(self.V):
            print("{0}\t\t{1}".format(i, dist[i]))
    def bellman_ford(self, src):
        # step1: fill the distance array and
        #        predecessor array
        dist = [float("Inf")] * self.V
        # Mark the source vertex
        dist[src] = 0
        # step2: relax edges |V| - 1 times
        for _ in range(self.V - 1):
            for s, d, w in self.graph:
                if dist[s] != float('Inf') and dist
                        [s] + w < dist[d]:
                    dist[d] = dist[s] + w
        # step 3: detect negative cycle
        # if value changes then we have a
        # negative cycle in the graph and
        # we cannot find the shortest distances
        for s, d, w in self.graph:
```



Page No.: 341
Date: / /

~~for x, d, w in s~~

if dist[x]! =float('Inf') and dist[x] + w < dist[d]:

print("Graph contains negative weight cycle")

return

No negative weight cycle found!

Print the distance and predecessor

self.printSolution(dist)

```
g = Graph(5)
g.add_edge(0,1,5)
g.add_edge(0,2,4)
g.add_edge(1,3,3)
g.add_edge(2,1,6)
g.add_edge(3,2,2)
g.bellman_ford(0)
```

SSSP : Directed Acyclic Graph (DAG) -:

. It work even graph contain negative weights. It is better than bellman ford.

DAG-SHORTEST-PATHS (G, w, s)

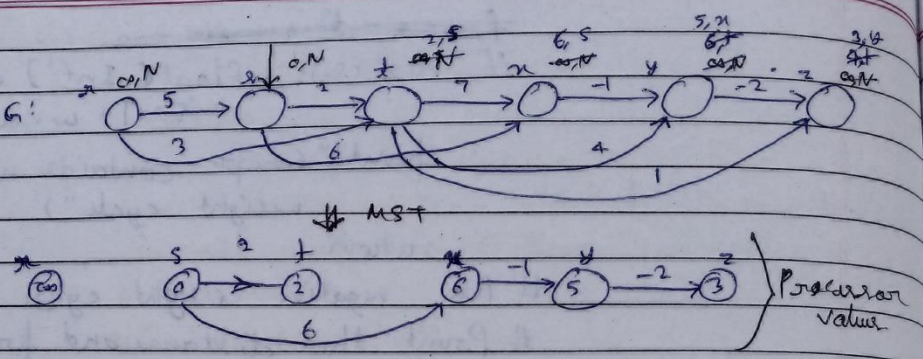
1) topologically sort the vertices of G
2) INITIALIZE-SINGLE-SOURCE (G, s)
3) for each vertex u, taken in topologically sorted order
4) for each vertex v ∈ G.Adj[u]
5) RELAX (u, v, w)

→ Time Complexity - O(V+E)

Note: It use topological sort.

Page No.: 342
Date: 25/11/20

Eg-

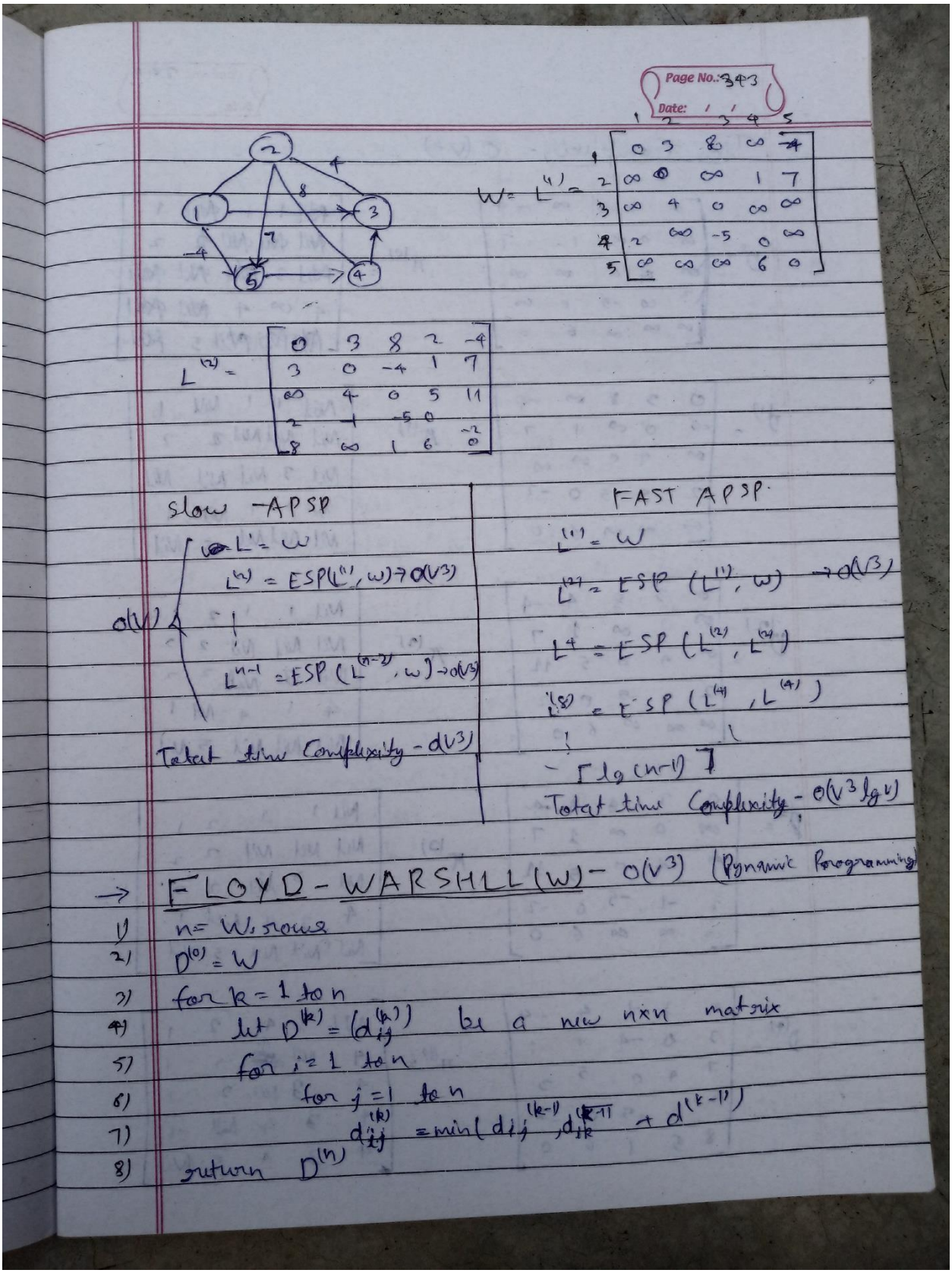


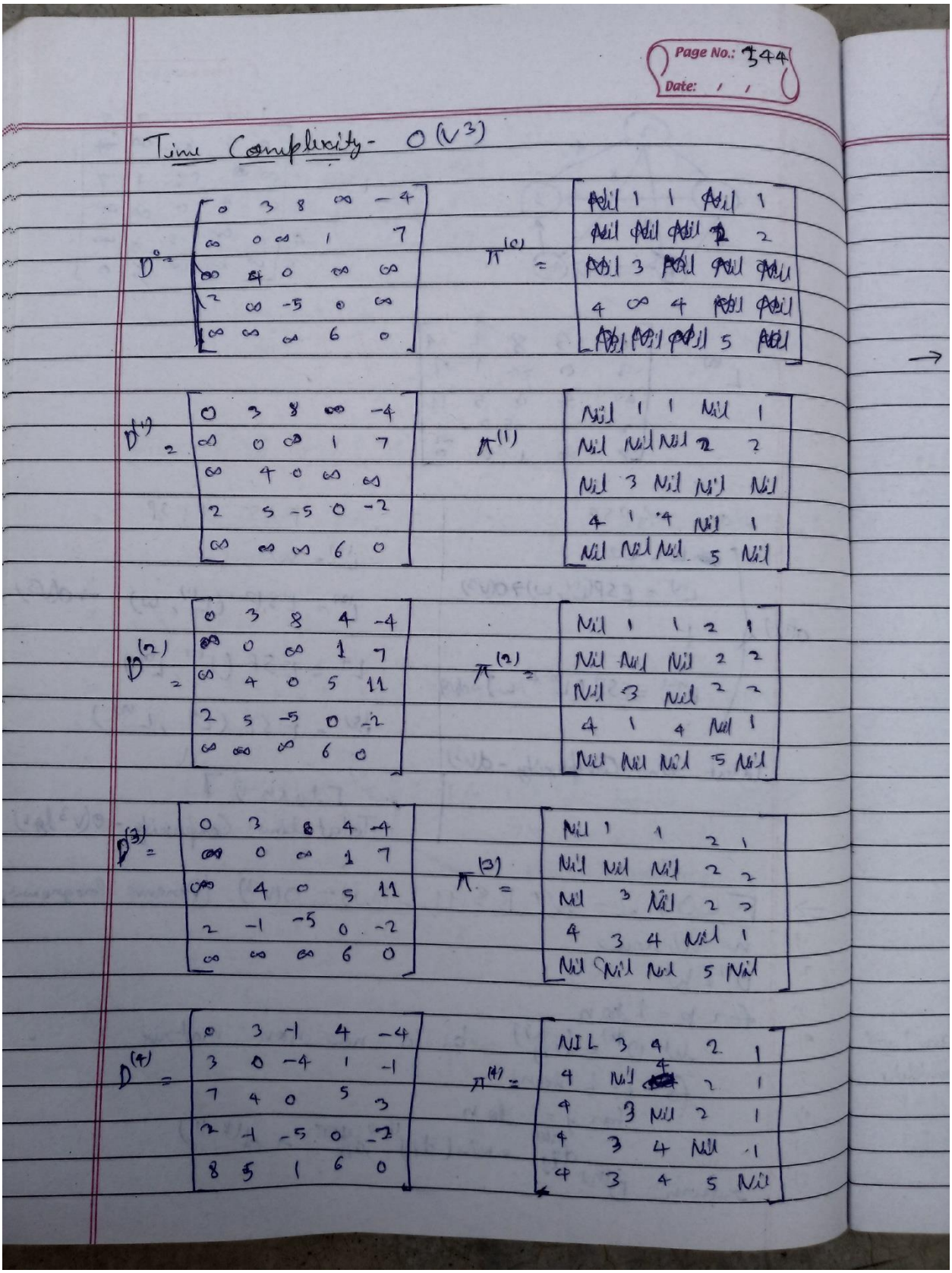
ALL PAIR SHORTEST PATH-

```
SLOW-ALL-PAIRS-SHORTEST-PATHS(W)
1) n = W.rows
2) L^(1) = W
3) for m = 2 to n-1
4)     let L^(m) be a new nxn matrix
5)     L^(m) = EXTEND-SHORTEST-PATHS(L^(m-1), W)
6) return L^(n-1)
```

```
EXTEND-SHORTEST-PATHS(L, W)
1) n = L.rows
2) let L' = (l'_ij) be a new nxn matrix
3) for i = 1 to n
4)     for j = 1 to n
5)         l'_ij = ∞
6)         for k = 1 to n
7)             l'_ij = min(l'_ij, l_ik + w_kj)
8) return L'
```

→ Time Complexity - $O(V^4)$


Page No.: 343
Date: / /
1 2 3 4 5
W = L(1) =
1 0 3 8 ∞ -4
2 ∞ 0 ∞ 1 7
3 ∞ 4 0 ∞ ∞
4 2 ∞ -5 0 ∞
5 ∞ ∞ ∞ 6 0
L(2) =
0 3 8 2 -4
3 0 -4 1 7
∞ 4 0 5 11
2 -1 -5 0 -2
8 ∞ 1 6 0
Slow - APSP
L1 = W
L(2) = ESP(L(1), W) → O(V3)
L(n-1) = ESP(L(n-2), W) → O(V3)
O(V)
Total time Complexity - O(V3)
FAST APSP
L(1) = W
L(2) = ESP (L(1), W) → O(V3)
L4 = ESP (L(2), L(2))
L(8) = ESP (L(4), L(4))
- ⌈lg(n-1)⌉
Total time Complexity - O(V3 lg V)
→ FLOYD-WARSHALL(W) - O(V3) (Dynamic Programming)
1) n = W.rows
2) D(0) = W
3) for k = 1 to n
4) let D(k) = (d(k)ij) be a new nxn matrix
5) for i = 1 to n
6) for j = 1 to n
7) d(k)ij = min(d(k-1)ij, d(k-1)ik + d(k-1)kj)
8) return D(n)



Page No.: 144
Date: / /

Time Complexity- $O(V^3)$

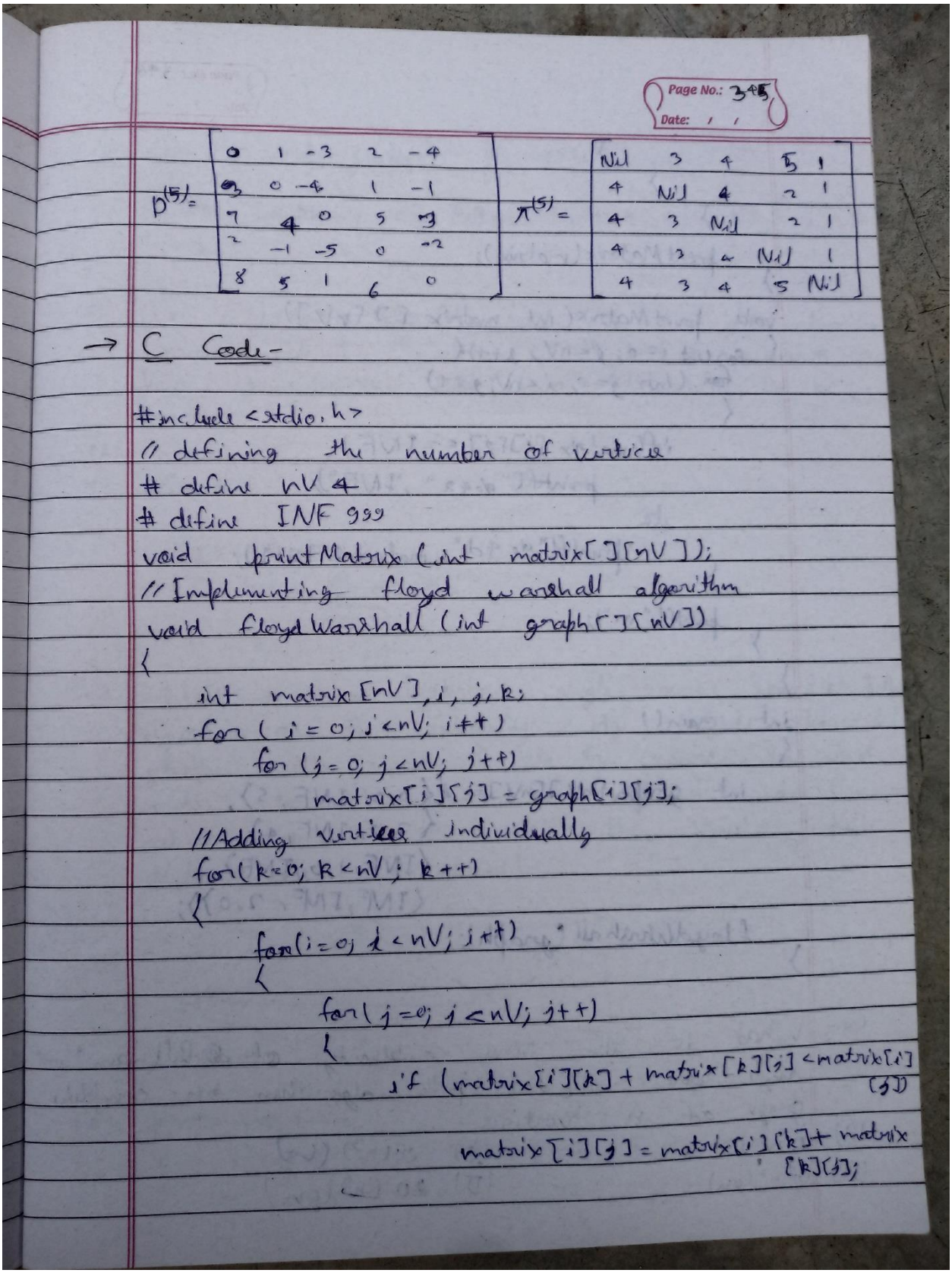
$D^{(0)} = \begin{bmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix}$ $\pi^{(0)} = \begin{bmatrix} Nil & 1 & 1 & Nil & 1 \\ Nil & Nil & Nil & 2 & 2 \\ Nil & 3 & Nil & Nil & Nil \\ 4 & \infty & 4 & Nil & Nil \\ Nil & Nil & Nil & 5 & Nil \end{bmatrix}$

$D^{(1)} = \begin{bmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix}$ $\pi^{(1)} = \begin{bmatrix} Nil & 1 & 1 & Nil & 1 \\ Nil & Nil & Nil & 2 & 2 \\ Nil & 3 & Nil & Nil & Nil \\ 4 & 1 & 4 & Nil & 1 \\ Nil & Nil & Nil & 5 & Nil \end{bmatrix}$

$D^{(2)} = \begin{bmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix}$ $\pi^{(2)} = \begin{bmatrix} Nil & 1 & 1 & 2 & 1 \\ Nil & Nil & Nil & 2 & 2 \\ Nil & 3 & Nil & 2 & 2 \\ 4 & 1 & 4 & Nil & 1 \\ Nil & Nil & Nil & 5 & Nil \end{bmatrix}$

$D^{(3)} = \begin{bmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix}$ $\pi^{(3)} = \begin{bmatrix} Nil & 1 & 1 & 2 & 1 \\ Nil & Nil & Nil & 2 & 2 \\ Nil & 3 & Nil & 2 & 2 \\ 4 & 3 & 4 & Nil & 1 \\ Nil & Nil & Nil & 5 & Nil \end{bmatrix}$

$D^{(4)} = \begin{bmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{bmatrix}$ $\pi^{(4)} = \begin{bmatrix} NIL & 3 & 4 & 2 & 1 \\ 4 & Nil & 4 & 2 & 1 \\ 4 & 3 & Nil & 2 & 1 \\ 4 & 3 & 4 & Nil & 1 \\ 4 & 3 & 4 & 5 & Nil \end{bmatrix}$



Page No.: 345
Date: / /

$D^{(5)} =$

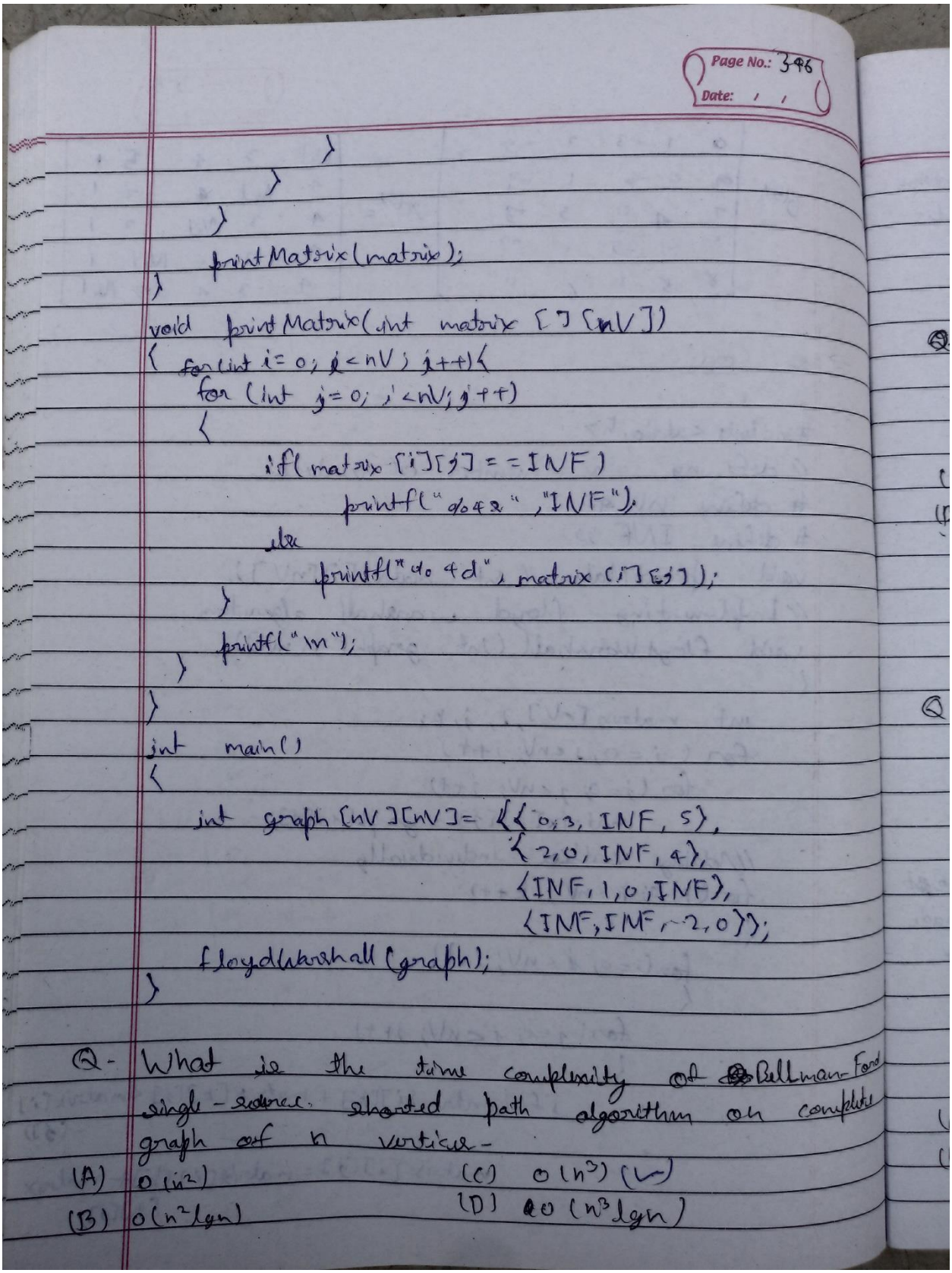
0	1	-3	2	-4
3	0	-4	1	-1
7	4	0	5	3
2	-1	-5	0	-2
8	5	1	6	0

$\pi^{(5)} =$

Nil	3	4	5	1
4	Nil	4	2	1
4	3	Nil	2	1
4	3	4	Nil	1
4	3	4	5	Nil

→ C Code-

```
#include <stdio.h>
// defining the number of vertices
# define nV 4
# define INF 999
void printMatrix (int matrix[][nV]);
// Implementing floyd warshall algorithm
void floydWarshall (int graph[][nV])
{
    int matrix[nV], i, j, k;
    for ( i = 0; j<nV; i++)
        for (j = 0; j<nV; j++)
            matrix[i][j] = graph[i][j];
    //Adding vertices individually
    for(k=0; k<nV; k++)
    {
        for(i=0; i<nV; i++)
        {
            for( j=0; j<nV; j++)
            {
                if (matrix[i][k] + matrix[k][j] < matrix[i]
                [j])

                matrix[i][j] = matrix[i][k]+ matrix
                [k][j];
```



```
        }
      }
    }
    printMatrix(matrix);
}
void printMatrix(int matrix[][nV])
{ for(int i=0; i<nV; i++){
    for (int j=0; j<nV; j++)
    {
        if(matrix[i][j]==INF)
            printf("%4s", "INF");
        else
            printf("%4d", matrix[i][j]);
    }
    printf("\n");
  }
}
int main()
{
    int graph[nV][nV]= {{ 0,3, INF, 5},
                        { 2,0, INF, 4},
                        {INF,1,0,INF},
                        {INF,INF,-2,0}};
    floydWarshall(graph);
}
```

Q- What is the time complexity of Bellman-Ford single-source shortest path algorithm on complete graph of n vertices-

(A) $O(n^2)$ (C) $O(n^3)$

(B) $O(n^2 \lg n)$ (D) $O(n^3 \lg n)$

Page No.: 347
Date: / /

In complete graph $E = O(V^2)$

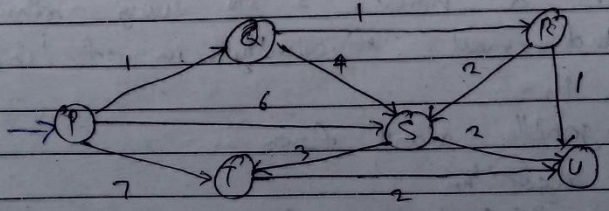
Time Complexity of Bellman-Ford = $O(V * E) = O(V^3)$
$= O(n^3)$

Q. To implement Dijkstra's shortest path algorithm on unweighted graphs so that it runs in linear time, the data structure to be used is.

(A) Queue (✓)
(B) Stack
(C) Heap
(D) B-Tree

Dijkstra use min-heap.
Dijkstra $\xrightarrow[\text{Queue}]{\text{use}}$ BFS

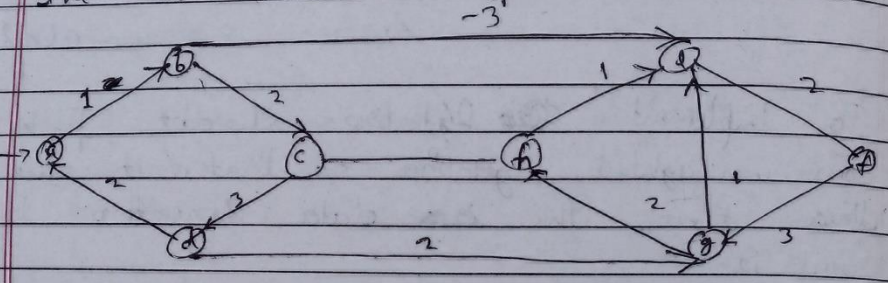
Q. Suppose we run Dijkstra single source shortest-path algorithm on the following edge weighted directed graph with vertex P as the source. In what order do the nodes get included into the set of vertices for which the shortest path distances are finalized?



(A) P, Q, R, S, T, U
(B) P, Q, R, U, S, T (✓)
(C) P, Q, R, U, T, S
(D) P, Q, T, R, U, S

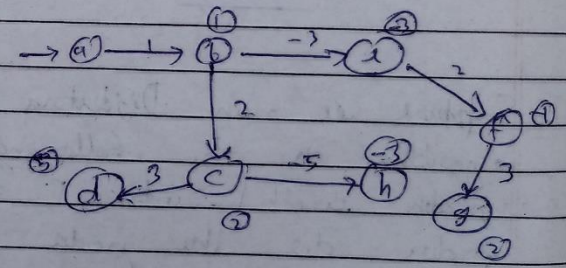
Page No.: 348
Date: / /

Q- Dijkstra's single source shortest path algorithm when vertex a is the below graph, computes the correct shortest path distance to-



(A) only vertex a
(B) only vertices a, e, f, g, h
(C) only vertices a, b, c, d
(D) all the vertices (✓)

Dijkstra

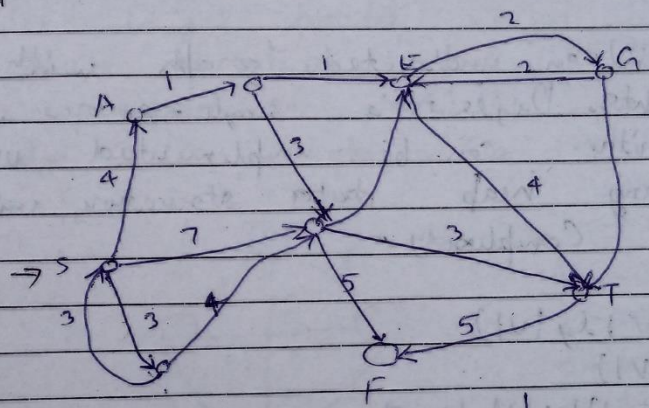


Q- In an unweighted, undirected graph, the shortest path from a node S to every other node is computed most efficiently, in terms of time complexity by-

(A) Dijkstra's algorithm start from S.
(B) Warshall's algorithm
(C) Performing a DFS starting from S
(D) Performing a BFS starting from S (✓)

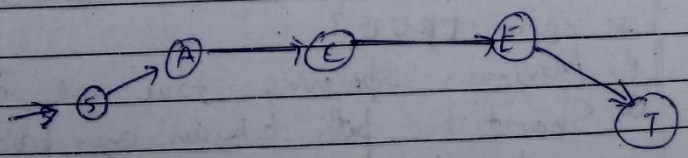
Page No.: 349
Date: / /

Q- Consider the directed graph shown in the figure below. There are multiple shortest paths between vertices S and T. Which one will be reported by Dijkstra's shortest path algorithm? Assume that, in any iteration, the shortest path to a vertex v is updated only when a strictly shorter path to v is discovered.



(A) SDT
(B) SBDT
(C) SACDT
(D) SACET

Dijkstra



Q- Which of the following statement(s) is/are correct regarding Bellman-Ford shortest path algorithm?

~~P: Atgar~~

Page No.: 350
Date: / /

P: Always finds a negative weighted cycle, if exists s.
Q: Finds whether any negative weighted cycle is reachable from the source.

(A) P Only
(B) Q only (✓)
(C) Both P and Q.
(D) Neither P nor Q

Q- Let G(V, E) an undirected graph with positive edge weights. Dijkstra's single-source shortest path algorithm can be implemented using the binary heap data structure with ~~for~~ time Complexity -

(A) O(|V|)
(B) O(|E| + |V| lg |V|)
(C) O(|V| lg |V|)
(D) O((|E| + |V|) lg |V|) (✓)

Q- Let G be a weighted connected graph with distinct positive edge weights. If every edge weight is increased by the same value, then which of the following statements is/are TRUE?
P: Minimum spanning tree of G does not change
Q: Shortest path between any pair of vertices does not change.

(A) P only (✓)
(B) Q only
(C) Neither P nor Q
(D) Both P and Q

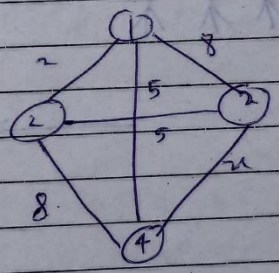
Page No.: 351
Date: / /

Q- Consider the weighted undirected graph with 4 vertices, where the weight of edge $\{i, j\}$ is given by the entry w_{ij} in the matrix W.

$$W = \begin{bmatrix} 0 & 2 & 8 & 5 \\ 2 & 0 & 5 & 8 \\ 8 & 5 & 0 & x \\ 5 & 8 & x & 0 \end{bmatrix}$$

The largest possible integer value of x, for which at least one shortest path between some pair of vertices will contain the edge with weight x is 12

Ans:



1 → 2 : 2
1 → 3 : 8 or 5+x (x<3)
1 → 4 : 5
2 → 3 : 5
2 → 4 : 8 or 5+x (x<3)
3 → 4 : x or 5+8 (x<13)

x < 5+8 = 13

x = 12