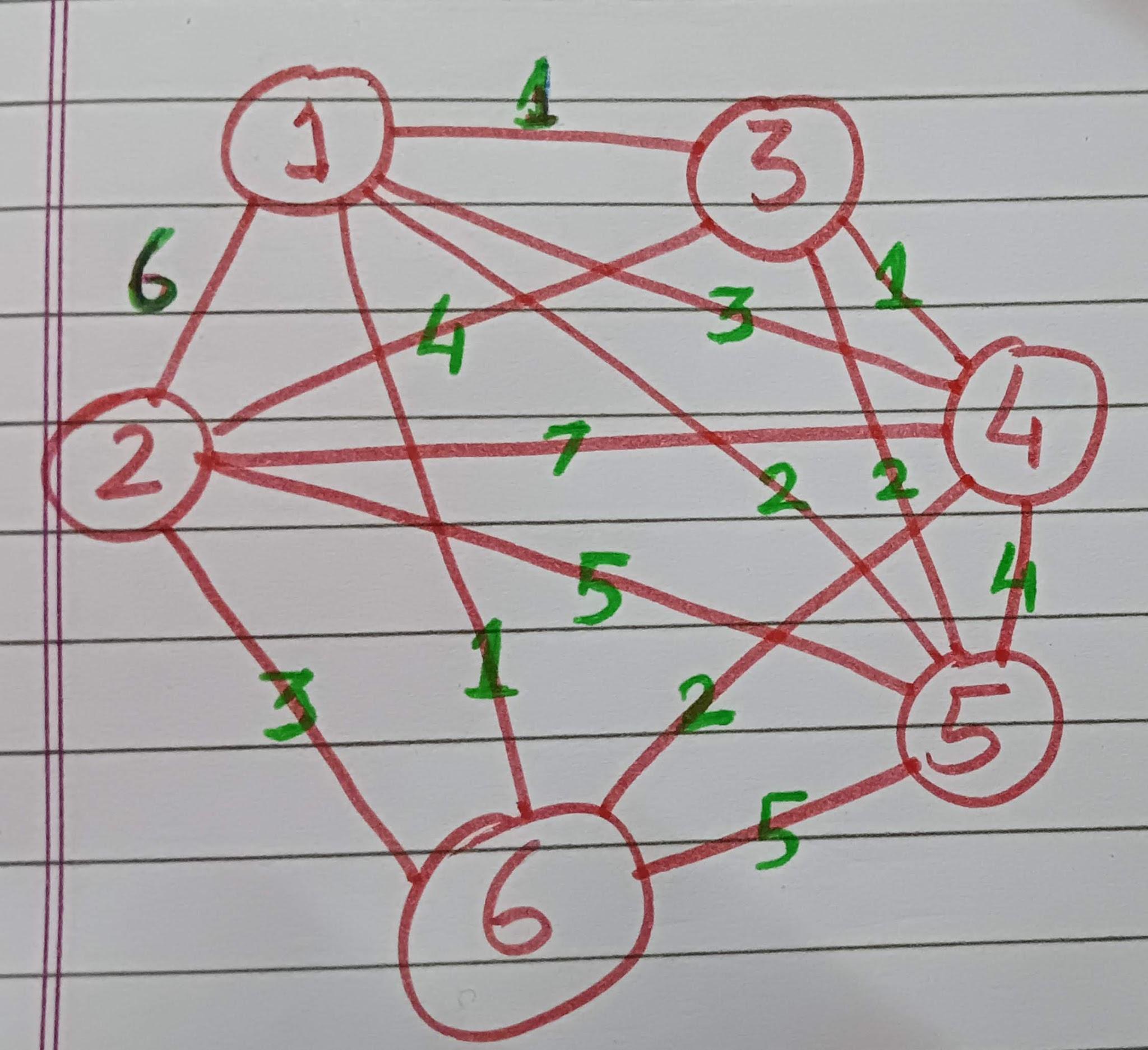
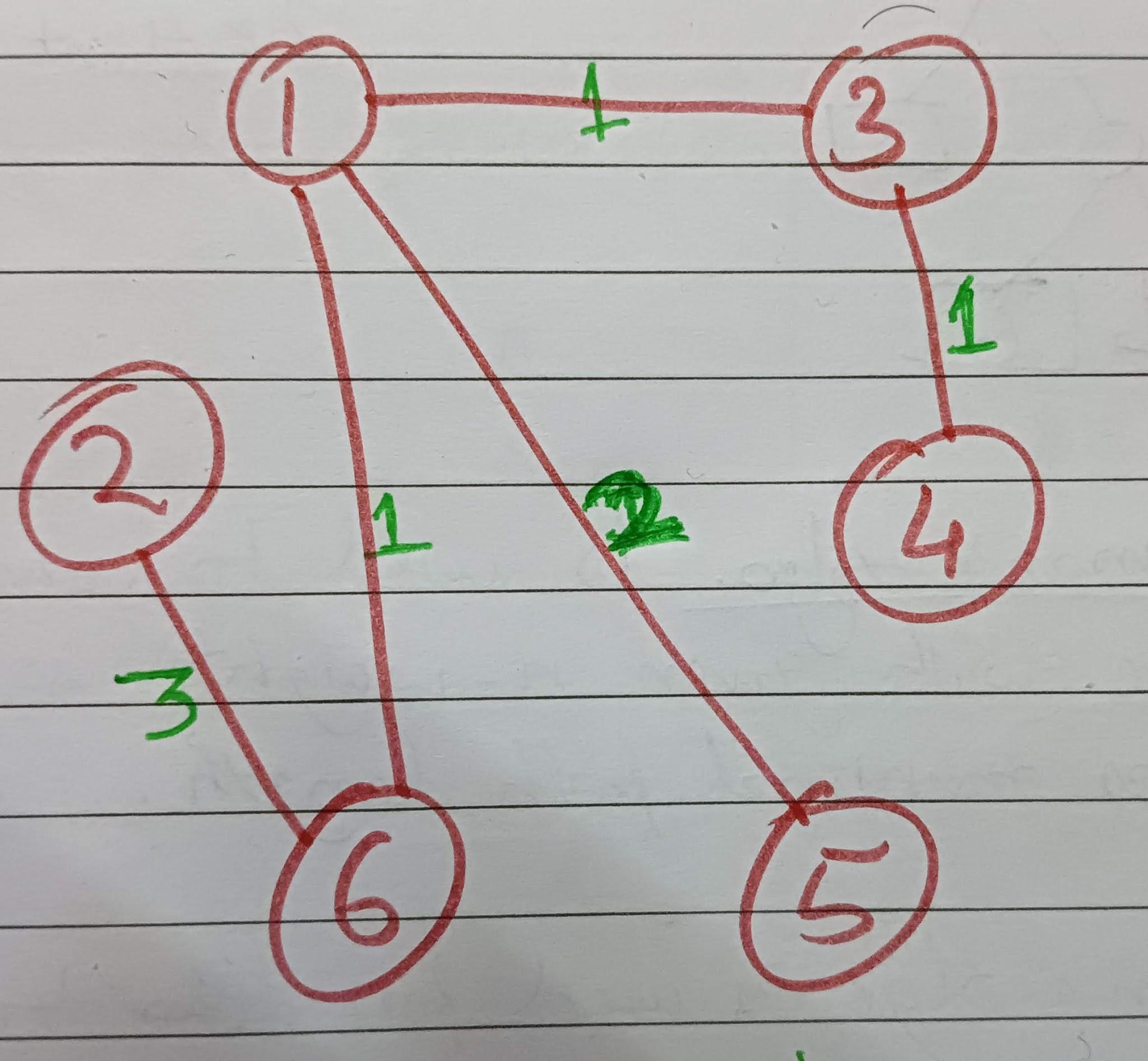
**Kruskal’s Algorithm**



This is inputted graph And this is outputted minimum spanning tree

Code on next page…

#include <stdio.h>

#include <stdlib.h>

#define max 100

typedef struct graph

{

int V;

int E;

struct graph \*adj;

} graph;

typedef struct EWtable

{

int E1, E2, W;

} EWtable;

void initialise\_graph(graph \*\*G)

{

for (int i = 0; i < max; i++)

{

G[i] = NULL;

}

}

void insert(graph \*\*G, int V, int node\_to\_join, int weight)

{

graph \*temp = G[V], \*p = (graph \*)malloc(sizeof(graph));

p->V = node\_to\_join;

p->E = weight;

p->adj = NULL;

if (temp == NULL)

{

G[V] = p;

}

else

{

while (temp->adj != NULL)

{

temp = temp->adj;

}

temp->adj = p;

}

}

void insert\_by\_array(graph \*\*G, int V, int nodes\_to\_join[][2], int no\_nodes)

{

for (int i = 0; i < no\_nodes; i++)

{

insert(G, V, nodes\_to\_join[i][0], nodes\_to\_join[i][1]);

}

}

void print\_adj\_list(graph \*\*G)

{

graph \*temp;

for (int i = 0; i < max; i++)

{

temp = G[i];

if (G[i] != NULL)

{

printf("%d:->", i);

while (temp != NULL)

{

printf(" (v:%d e:%d) ", temp->V, temp->E);

temp = temp->adj;

}

printf("\n");

}

}

}

void add\_value\_to\_graph(graph \*\*G, int number\_of\_vertex)

{

while (number\_of\_vertex--)

{

int V;

printf("Enter vertex: ");

scanf("%d", &V);

int adj\_edges;

printf("Enter number of linked edges to vertex: ");

scanf("%d", &adj\_edges);

while (adj\_edges--)

{

int node\_to\_add, weight;

printf("Enter node to join to vertex and their weight:\n");

scanf("%d%d", &node\_to\_add, &weight);

insert(G, V, node\_to\_add, weight);

}

}

}

int count\_edges(graph \*\*G)

{

int ct = 0;

for (int i = 0; i < max; i++)

{

graph \*temp = G[i];

if (G[i] != NULL)

{

while (temp != NULL)

{

ct++;

temp = temp->adj;

}

}

}

return ct;

}

void set\_EWtable(graph \*\*G, EWtable \*table)

{

graph \*temp;

int idx = 0;

for (int i = 0; i < max; i++)

{

temp = G[i];

if (G[i] != NULL)

{

while (temp != NULL)

{

table[idx].E1 = i;

table[idx].E2 = temp->V;

table[idx].W = temp->E;

temp = temp->adj;

idx++;

}

}

}

}

void print\_EWtable(EWtable \*table, int n)

{

printf("Edges -> Weights \n");

for (int i = 0; i < n; i++)

{

printf("%d-%d -> %d\n", table[i].E1, table[i].E2, table[i].W);

}

printf("\n");

}

void sort\_EWtable(EWtable \*table, int n)

{

for (int i = 0; i < n; i++)

{

for (int j = i + 1; j < n; j++)

{

if (table[i].W > table[j].W)

{

EWtable temp = table[i];

table[i] = table[j];

table[j] = temp;

}

}

}

}

int find\_op(int union\_array[], int count\_vertices, int X)

{

if (union\_array[X] == X)

{

return X;

}

else

{

return find\_op(union\_array, count\_vertices, union\_array[X]);

}

}

int union\_op(int union\_array[], int count\_vertices, int a, int b)

{

a = find\_op(union\_array, count\_vertices, a), b = find\_op(union\_array, count\_vertices, b);

if (a == b)

{

return -1;

}

union\_array[b] = a;

return 0;

}

void initialise\_Union\_array(int \*a, int size)

{

for (int i = 0; i <= size; i++)

{

a[i] = i;

}

}

int count\_vertices(graph \*\*G)

{

int ct = 0;

for (int i = 0; i < max; i++)

{

if (G[i] != NULL)

ct++;

}

return ct;

}

void swap(int \*a, int \*b)

{

int temp = (\*a);

(\*a) = (\*b);

(\*b) = temp;

}

int count\_total\_weight\_sum(graph \*\*ANS)

{

EWtable ans\_table[2 \* max];

set\_EWtable(ANS, ans\_table);

int size=count\_edges(ANS);

int cantor\_pair[10000], ans = 0;

for (int i = 0; i < 10000; i++)

cantor\_pair[i] = 0;

for (int i = 0; i < size; i++)

{

int a = ans\_table[i].E1, b = ans\_table[i].E2;

if (ans\_table[i].E1 < ans\_table[i].E2)

{

swap(&a, &b);

}

if (cantor\_pair[a \* a + a + b] == 0)

{

cantor\_pair[a \* a + a + b] = 1;

ans += ans\_table[i].W;

}

}

return ans;

}

int construct\_minimum\_spanning\_tree(graph \*\*G, graph \*\*ANS, EWtable \*table, int n)

{

initialise\_graph(ANS);

EWtable ans\_table[3 \* max];

int size = count\_vertices(G);

int union\_array[size + 1];

initialise\_Union\_array(union\_array, size);

int ct = 0, union\_ct = 0;

for (int i = 0; i < n; i++)

{

if (union\_op(union\_array, size, table[i].E1, table[i].E2) != -1 && union\_ct != count\_vertices(G) - 1)

{

union\_ct++;

ans\_table[ct].E1 = table[i].E1;

ans\_table[ct].E2 = table[i].E2;

ans\_table[ct].W = table[i].W;

for (int j = 0; j < n; j++)

{

if ((table[j].E1 == table[i].E2 && table[j].E2 == table[i].E1 && table[j].W == table[i].W) && j != i)

{

ct++;

ans\_table[ct].E1 = table[i].E2;

ans\_table[ct].E2 = table[i].E1;

ans\_table[ct].W = table[i].W;

break;

}

}

ct++;

}

}

printf("Final EWtable to be inseted in minimum spanning tree:\n");

print\_EWtable(ans\_table, ct);

int ans = 0;

for (int i = 0; i < ct; i++)

{

insert(ANS, ans\_table[i].E1, ans\_table[i].E2, ans\_table[i].W);

}

ans = count\_total\_weight\_sum(ANS);

return ans;

}

void kruskal(graph \*\*G, graph \*\*ANS)

{

EWtable table[3 \* max], ans\_table[3 \* max];

initialise\_graph(ANS);

printf("\nThe inputted graph is:\n");

print\_adj\_list(G);

set\_EWtable(G, table);

int n = count\_edges(G);

sort\_EWtable(table, n);

printf("\nThe edges and weights in sorted manner: \n");

print\_EWtable(table, n);

int min\_weight\_sum = construct\_minimum\_spanning\_tree(G, ANS, table, n);

printf("Minimum spanning tree with total weight sum: %d\n", min\_weight\_sum);

print\_adj\_list(ANS);

}

int main()

{

graph \*G[max], \*ANS[max];

initialise\_graph(G);

// Inserting values in graph...

int v\_1[5][2] = {{2, 6}, {3, 1}, {4, 3}, {5, 2}, {6, 1}};

int v\_2[5][2] = {{6, 3}, {1, 6}, {3, 4}, {5, 5}, {4, 7}};

int v\_3[4][2] = {{1, 1}, {2, 4}, {4, 1}, {5, 2}};

int v\_4[5][2] = {{2, 7}, {1, 3}, {3, 1}, {5, 4}, {6, 2}};

int v\_5[5][2] = {{3, 2}, {1, 2}, {2, 5}, {4, 4}, {6, 5}};

int v\_6[4][2] = {{4, 2}, {1, 1}, {5, 5}, {2, 3}};

insert\_by\_array(G, 1, v\_1, 5);

insert\_by\_array(G, 2, v\_2, 5);

insert\_by\_array(G, 3, v\_3, 4);

insert\_by\_array(G, 4, v\_4, 4);

insert\_by\_array(G, 5, v\_5, 5);

insert\_by\_array(G, 6, v\_6, 4);

// Using kruskal's algorithm...

kruskal(G, ANS);

return 0;

}

**OUTPUT:**

**The inputted graph is:**

**1:-> (v:2 e:6) (v:3 e:1) (v:4 e:3) (v:5 e:2) (v:6 e:1)**

**2:-> (v:6 e:3) (v:1 e:6) (v:3 e:4) (v:5 e:5) (v:4 e:7)**

**3:-> (v:1 e:1) (v:2 e:4) (v:4 e:1) (v:5 e:2)**

**4:-> (v:2 e:7) (v:1 e:3) (v:3 e:1) (v:5 e:4)**

**5:-> (v:3 e:2) (v:1 e:2) (v:2 e:5) (v:4 e:4) (v:6 e:5)**

**6:-> (v:4 e:2) (v:1 e:1) (v:5 e:5) (v:2 e:3)**

**The edges and weights in sorted manner:**

**Edges -> Weights**

**1-3 -> 1**

**1-6 -> 1**

**3-1 -> 1**

**3-4 -> 1**

**4-3 -> 1**

**6-1 -> 1**

**1-5 -> 2**

**5-3 -> 2**

**5-1 -> 2**

**6-4 -> 2**

**3-5 -> 2**

**4-1 -> 3**

**2-6 -> 3**

**1-4 -> 3**

**6-2 -> 3**

**5-4 -> 4**

**4-5 -> 4**

**2-3 -> 4**

**3-2 -> 4**

**5-2 -> 5**

**2-5 -> 5**

**6-5 -> 5**

**5-6 -> 5**

**1-2 -> 6**

**2-1 -> 6**

**2-4 -> 7**

**4-2 -> 7**

**Final EWtable to be inseted in minimum spanning tree:**

**Edges -> Weights**

**1-3 -> 1**

**3-1 -> 1**

**1-6 -> 1**

**6-1 -> 1**

**3-4 -> 1**

**4-3 -> 1**

**1-5 -> 2**

**5-1 -> 2**

**2-6 -> 3**

**6-2 -> 3**

**Minimum spanning tree with total weight sum: 8**

**1:-> (v:3 e:1) (v:6 e:1) (v:5 e:2)**

**2:-> (v:6 e:3)**

**3:-> (v:1 e:1) (v:4 e:1)**

**4:-> (v:3 e:1)**

**5:-> (v:1 e:2)**

**6:-> (v:1 e:1) (v:2 e:3)**