CODE:

prog1_4.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "graph.h"

int main() {
    node *head=NULL;
    int **adj_matrix=NULL, n, *vertex_arr, chx, u, v;
    bool flg = true;

    adj_matrix = input_adj_matrix(&n);
    vertex_arr = input_vertex_arr(n);

    head = create_adj_list(adj_matrix, vertex_arr, n);
    print_adj_list(head);



    while(flg) {
        printf("\nSelect Option\n");
        printf("1. Add vertex\n");
        printf("2. Add edge\n");
        printf("3. Delete edge\n");
        printf("4. Delete vertex\n");
        printf("0. Exit\n");
```

```
        printf("enter choice: ");
        scanf("%d", &chx);

        switch (chx) {
            case 0:
                flg = false;
                break;
            case 1:
                printf("Enter vertex name: ");
                scanf("%d", &v);
                add_vertex(head, v);
                print_adj_list(head);
                break;
            case 2:
                printf("Enter vertices of edge (comma
seperated u,v): ");
                scanf("%d,%d", &u, &v);
                add_edge(head, u, v);
                print_adj_list(head);
                break;
            case 3:
                printf("Enter vertices of edge (comma
seperated u,v): ");
                scanf("%d,%d", &u, &v);
                del_edge(head, u, v);
                print_adj_list(head);
                break;
            case 4:
```

```
                printf("Enter vertex name: ");
                scanf("%d", &v);
                head = del_vertex(head, v);
                print_adj_list(head);
                break;
            default:
                printf("Invalid Input.\n");
        }
    }

    return 0;
}
```

prog5_8&10_11.c

```
#include <stdio.h>
#include <stdlib.h>
#include "graph.h"

int main() {
    node *head=NULL;
    int **adj_matrix=NULL, n, *vertex_arr, chx, start_vertex;
    bool flg = true, found=false;

    adj_matrix = input_adj_matrix(&n);
    vertex_arr = input_vertex_arr(n);

    head = create_adj_list(adj_matrix, vertex_arr, n);
```

```c
print_adj_list(head);

printf("Enter start vertex (BFS, DFS, Dijkstra): ");
scanf("%d", &start_vertex);
for (int i=0; i<n; i++) {
    if(vertex_arr[i] == start_vertex) {
        found = true;
        break;
    }
}
if(!found) {
    printf("Invalid start vertex.\n");
    exit(-1);
}

while(flg) {
    printf("\nSelect Option\n");
    printf("1. BFS\n");
    printf("2. DFS\n");
    printf("3. Dijkstra's Algorithm\n");
    printf("4. Floyd's Algorithm\n");
    printf("5. Kruskal's Algorithm\n");
    printf("6. Prim's Algorithm\n");
    printf("0. Exit\n");

    printf("enter choice: ");
    scanf("%d", &chx);
```

```c
    switch (chx) {
        case 0:
            flg = false;
            break;
        case 1:
            bfs(head, n, start_vertex);
            break;
        case 2:
            dfs(head, n, start_vertex);
            break;
        case 3:
            dijkstra(head, n, start_vertex);
            break;
        case 4:
            floyd_shortest_path(adj_matrix, n);
            break;
        case 5:
            kruskal_mst(head, n);
            break;
        case 6:
            prim_mst(head, n);
            break;
        default:
            printf("Invalid Input.\n");
    }
}

// free mat and list
```

```c
    return 0;
}
```

prog9.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "graph.h"

int main() {
    node *head=NULL;
    int **adj_matrix=NULL, n, *vertex_arr, chx, start_vertex;
    bool flg = true, found=false;

    printf("Enter Directed Acyclic Graph:\n");
    adj_matrix = input_adj_matrix(&n);
    vertex_arr = input_vertex_arr(n);

    head = create_adj_list(adj_matrix, vertex_arr, n);
    print_adj_list(head);

    topo_sort(head, n);

    // free mat and list
    return 0;
}
```

graph.h

```c
#ifndef GRAPH_H
#define GRAPH_H

#include <stdio.h>
#include <stdlib.h>

struct node {
   int val;
   int weight;
   struct node *ptr1;
   struct node *ptr2;
};
typedef struct node node;

enum bool {
   false = 0,
   true = 1
};
typedef enum bool bool;

typedef struct prim_node {
   int vertex;
   int pred;
   int length;
   bool perm_status;
} prim_node;

typedef struct kruskal_edge {
```

```
    int u;
    int v;
    int weight;
} kruskal_edge;



void fill_indeg_arr(node *adj_list, int *indeg_arr, int n);
int find_zero_indeg(int *indeg_arr, int n);
bool check_all_vertex_removed(int *indeg_arr, int n);
void topo_sort(node *adj_list, int n);
void floyd_shortest_path(int **adj_matrix, int n);
void dijkstra(node *adj_list, int n, int start_vertex);
void swap(kruskal_edge *a, kruskal_edge *b);
int partition(kruskal_edge *arr,int low,int high);
void quick_sort(kruskal_edge *arr,int low,int high);
kruskal_edge *create_edge_arr(node *adj_list, int n, int
*no_edges);
void union_set(int *disjoint_set, int len, int u, int v);
int find_set(int *disjoint_set, int len, int u);
void kruskal_mst(node *adj_list, int n);
int find_least_len(prim_node *arr, int len);
bool check_all_reached(prim_node *arr, int len);
void prim_mst(node *adj_list, int n);
void bfs(node *adj_list, int n, int start_vertex);
void dfs(node *adj_list, int n, int start_vertex);
node *create_node(int val, int weight);
```

```c
node *create_adj_list(int **adj_matrix, int *vertex_list,
int n);
void print_adj_list(node *adj_list_head);
int **input_adj_matrix(int *n);
int *input_vertex_arr(int n);
void add_vertex(node *head, int n);
void add_edge(node *head, int u, int v);
void del_edge(node *head, int u, int v);
node *del_vertex(node *head, int n);
void clr_buffr();

#endif
```

graph.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "graph.h"

void fill_indeg_arr(node *adj_list, int *indeg_arr, int n) {
   // indeg_arr[i] = indegree of vertex i
   // indeg_arr[i] = -1 if vertex i does not exists
   node *down_ptr=NULL, *side_ptr=NULL;

   // change arr elem to 0 except those which are -1
   for (int i=0; i<n; i++) {
       if (indeg_arr[i] != -1) {
```

```
            indeg_arr[i] = 0;
        }
    }

    down_ptr = adj_list;
    while(down_ptr) {
        if (indeg_arr[down_ptr->val] == -1) {
            down_ptr = down_ptr->ptr2;
            continue; // vertex is removed.
        }
        side_ptr = down_ptr->ptr1;
        while(side_ptr) {
            if (indeg_arr[side_ptr->val] != -1) {
                indeg_arr[side_ptr->val] += 1;
            }
            side_ptr = side_ptr->ptr1;
        }
        down_ptr = down_ptr->ptr2;
    }
}

int find_zero_indeg(int *indeg_arr, int n) {
    int indx = -1;
    for (int i=0; i<n; i++) {
        if (indeg_arr[i] == 0) {
            indx = i;
            break;
        }
```

```
    }
    return indx;
}

bool check_all_vertex_removed(int *indeg_arr, int n) {
    bool all_removed = true;

    for (int i=0; i<n; i++) {
        if(indeg_arr[i] != -1) {
            all_removed = false;
            break;
        }
    }
    return all_removed;
}

void topo_sort(node *adj_list, int n) {
    int *indeg_arr = NULL, indx_zero;

    indeg_arr = calloc(n,sizeof(int)); // init to 0. so that
no -1
    if (indeg_arr == NULL) {
        printf("MEMORY ALLOCATION ERROR.\n");
        exit(-1);
    }

    printf("TOPOLOGICAL SORT.\n");
    while(check_all_vertex_removed(indeg_arr, n) != true) {
```

```
        fill_indeg_arr(adj_list, indeg_arr, n);

        indx_zero = find_zero_indeg(indeg_arr, n);
        if (indx_zero == -1) {
            printf("ERROR. Cycle detected.\n");
        }

        while(indx_zero != -1) {
            printf("%d, ", indx_zero);
            indeg_arr[indx_zero] = -1;
            indx_zero = find_zero_indeg(indeg_arr, n);
        }
    }
    printf("\n");
    free(indeg_arr);
}


void floyd_shortest_path(int **adj_matrix, int n) {
    int **dist=NULL, **pred=NULL; // distance and predecessor
matrix;
    int INF = 1000;

    dist = malloc(n*sizeof(int *));
    pred = malloc(n*sizeof(int *));
    if (dist == NULL || pred == NULL) {
        printf("MEMORY ALLOCATION ERROR.\n");
        exit(-1);
```

```
    }

    for (int i=0; i<n; i++) {
        dist[i] = malloc(n*sizeof(int));
        pred[i] = malloc(n*sizeof(int));
        if (dist[i] == NULL || pred[i] == NULL) {
            printf("MEMORY ALLOCATION ERROR.\n");
            exit(-1);
        }
    }

    // D_(k) is path between edges using 0, 1, ..k as
intermediary nodes
    // find D_(-1) ie direct path dist
    // ie just adj_matrix but 0 replaced by INF
    for (int i=0; i<n; i++) {
        for (int j=0; j<n; j++) {
            if (adj_matrix[i][j] == 0) {
                dist[i][j] = INF;
                pred[i][j] = -1; // no direct predecesor
            } else {
                dist[i][j] = adj_matrix[i][j];
                pred[i][j] = i; // predecesor of j is i
(direct path)
            }
        }
    }
```

```
    // compute D_(0), D_(1), ..., D(n-1)
    // D_(k)[i][j] = min(D_(k-1)[i][j], D_(k-1)[i][k] +
D_(k-1)[k][j])
    // ie path i-j direct or through intermediary k
    for (int k=0; k<n; k++) {
        for(int i=0; i<n; i++) {
            for(int j=0; j<n; j++) {
                if (dist[i][j] > (dist[i][k] + dist[k][j])) {
                    dist[i][j] = dist[i][k] + dist[k][j];
                    pred[i][j] = pred[k][j];
                } else {
                    // ie dist[i][j] <= dist[i][k] +
dist[k][j]
                    // no update
                    continue;
                }
            }
        }
    }

    printf("FLOYD'S ALGORITHM\n");
    printf("Predecessor Matrix\n");
    for (int i=0; i<n; i++) {
        for (int j=0; j<n; j++) {
            printf("%d\t", pred[i][j]);
        }
        printf("\n");
    }
```

```c
    printf("\nDistance Matrix\n");
    for (int i=0; i<n; i++) {
        for (int j=0; j<n; j++) {
            if (dist[i][j] >= INF) {
                printf("INF\t");
            } else {
                printf("%d\t", dist[i][j]);
            }
        }
        printf("\n");
    }

    for(int i=0; i<n; i++) {
        free(dist[i]);
        free(pred[i]);
    }
    free(dist);
    free(pred);
}

void dijkstra(node *adj_list, int n, int start_vertex) {
    int curr, neighbour;
    node *down_ptr=NULL, *side_ptr=NULL;
    // init all node
    prim_node *arr = malloc(n*sizeof(prim_node));
    if (arr == NULL) {
        printf("MEMORY ALLOCATION ERROR.\n");
        exit(-1);
```

```
    }

    for (int i=0; i<n; i++) {
        arr[i].vertex = i;
        arr[i].length = 10000;
        arr[i].pred = -1;
        arr[i].perm_status = false;
    }
    // start at 0;
    arr[start_vertex].length = 0;

    while(!check_all_reached(arr, n)) {
        curr = find_least_len(arr, n);
        arr[curr].perm_status = true;

        // find adjacent
        down_ptr = adj_list;
        while(down_ptr) {
            if (down_ptr->val == curr) {
                side_ptr = down_ptr->ptr1;
                while(side_ptr) {
                    neighbour = side_ptr->val;
                    if (arr[curr].length + side_ptr->weight <
arr[neighbour].length) {
                        arr[neighbour].length =
arr[curr].length + side_ptr->weight;
                        arr[neighbour].pred = curr;
                    }
```

```
                    side_ptr = side_ptr->ptr1;
                }
                down_ptr = NULL;
            } else {
                down_ptr = down_ptr->ptr2;
            }
        }
    }

    printf("DIJKSTRA'S ALGO.\n");
    printf("Vertex\tLength\tPred\n");
    for (int i=0; i<n; i++) {
        printf("%d\t%d\t%d\n", arr[i].vertex, arr[i].length,
arr[i].pred);
    }
    printf("\n");
    free(arr);
}

// Quicksort
void swap(kruskal_edge *a, kruskal_edge *b) {
    kruskal_edge tmp;
    tmp = *a;
    *a = *b;
    *b = tmp;
}

int partition(kruskal_edge *arr,int low,int high) {
```

```
 int pivot=arr[high].weight;
 //Index of smaller element and Indicate
 //the right position of pivot found so far
 int i=(low-1);

 for(int j=low;j<=high;j++) {
   if(arr[j].weight < pivot) {
     //Increment index of smaller element
     i++;
     swap(&arr[i],&arr[j]);
   }
 }
 swap(&arr[i+1],&arr[high]);
 return (i+1);
}

void quick_sort(kruskal_edge *arr,int low,int high) {
 if(low<high) {
   int pi = partition(arr,low,high);
   quick_sort(arr,low,pi-1);
   quick_sort(arr,pi+1,high);
 }
}

kruskal_edge *create_edge_arr(node *adj_list, int n, int
*no_edges) {
   kruskal_edge *arr = NULL;
   int len = 0;
```

```
    bool **visited_matrix = NULL;
    node *down_ptr = NULL, *side_ptr = NULL;

    // to keep track of edges already added in array
    visited_matrix = calloc(n, sizeof(bool *));
    if (visited_matrix == NULL) {
        printf("MEMORY ALLOCATION ERROR.\n");
        exit(-1);
    }
    for (int i=0; i<n; i++) {
        visited_matrix[i] = calloc(n, sizeof(bool));
        if (visited_matrix[i] == NULL) {
            printf("MEMORY ALLOCATION ERROR.\n");
            exit(-1);
        }
    }

    down_ptr = adj_list;
    while(down_ptr) {
        side_ptr = down_ptr->ptr1;
        while(side_ptr) {
            // check if not visited
            if
(!visited_matrix[down_ptr->val][side_ptr->val]) {
                len++;
                arr = realloc(arr, len*sizeof(kruskal_edge));
                if (arr == NULL) {
                    printf("MEMORY ALLOCATION ERROR.\n");
```

```
                exit(-1);
            }
            arr[len-1].u = down_ptr->val;
            arr[len-1].v = side_ptr->val;
            arr[len-1].weight = side_ptr->weight;

            visited_matrix[arr[len-1].u][arr[len-1].v] =
true;
            visited_matrix[arr[len-1].v][arr[len-1].u] =
true;
        }

        side_ptr = side_ptr->ptr1;
    }
    down_ptr = down_ptr->ptr2;
}

for(int i=0; i<n; i++) {
    free(visited_matrix[i]);
}
free(visited_matrix);
*no_edges = len;
return arr;
}

void union_set(int *disjoint_set, int len, int u, int v) {
    int paren_u, paren_v;
```

```
    paren_u = find_set(disjoint_set, len, u);
    paren_v = find_set(disjoint_set, len, v);

    disjoint_set[paren_v] = paren_u; // parent of v = u;
}

int find_set(int *disjoint_set, int len, int u) {
    int indx = disjoint_set[u];
    int old_indx = u;
    while (indx != -1) {
        old_indx = indx;
        indx = disjoint_set[indx];
    }
    return old_indx;
}

void kruskal_mst(node *adj_list, int n) {
    int *disjoint_set = NULL, no_edges, i, paren_u, paren_v;
    kruskal_edge *edge_arr = NULL, curr_edge;

    disjoint_set = malloc(n*sizeof(int));
    if (disjoint_set == NULL) {
        printf("MEMORY ALLOCATION ERROR.\n");
        exit(-1);
    }
    for(int i=0; i<n; i++) {
        disjoint_set[i] = -1;
    }
```

```
    edge_arr = create_edge_arr(adj_list, n, &no_edges);
    quick_sort(edge_arr, 0, no_edges-1);

    printf("KRUSKAL'S ALGORITHM\n");
    printf("u\tv\tWeight\n");
    i = 0;
    while(i < no_edges) {
        curr_edge = edge_arr[i];

        // see if not cycle
        paren_u = find_set(disjoint_set, n, curr_edge.u);
        paren_v = find_set(disjoint_set, n, curr_edge.v);

        if (paren_u != paren_v) {
            union_set(disjoint_set, n, curr_edge.u,
curr_edge.v);
            printf("%d\t%d\t%d\n", curr_edge.u, curr_edge.v,
curr_edge.weight);
        }
        i++;
    }
    free(disjoint_set);
    free(edge_arr);
}

int find_least_len(prim_node *arr, int len) {
    int curr_min = 100000, curr_indx = -1;
```

```
   for (int i=0; i<len; i++) {
       if (arr[i].perm_status == false && arr[i].length <
curr_min) {
           curr_min = arr[i].length;
           curr_indx = i;
       }
   }
   return curr_indx;
}

bool check_all_reached(prim_node *arr, int len) {
   bool checked_all = true;
   for (int i=0; i<len; i++) {
       if (arr[i].perm_status != true) {
           checked_all = false;
           break;
       }
   }
   return checked_all;
}

void prim_mst(node *adj_list, int n) {
   int curr, neighbour;
   node *down_ptr=NULL, *side_ptr=NULL;
   // init all node
   prim_node *arr = malloc(n*sizeof(prim_node));
   if (arr == NULL) {
       printf("MEMORY ALLOCATION ERROR.\n");
```

```
        exit(-1);
    }

    for (int i=0; i<n; i++) {
        // arr[i] = malloc(sizeof(prim_node));
        arr[i].vertex = i;
        arr[i].length = 10000;
        arr[i].pred = -1;
        arr[i].perm_status = false;
    }

    // start at 0;
    arr[0].length = 0;

    while(!check_all_reached(arr, n)) {
        curr = find_least_len(arr, n);
        arr[curr].perm_status = true;

        // find adjacent
        down_ptr = adj_list;
        while(down_ptr) {
            if (down_ptr->val == curr) {
                side_ptr = down_ptr->ptr1;
                while(side_ptr) {
                    neighbour = side_ptr->val;
                    if (side_ptr->weight <
arr[neighbour].length) {
```

```
                        arr[neighbour].length =
side_ptr->weight;
                        arr[neighbour].pred = curr;
                    }
                    side_ptr = side_ptr->ptr1;
                }
                down_ptr = NULL;
            } else {
                down_ptr = down_ptr->ptr2;
            }
        }
    }

    printf("PRIMS ALGO.\n");
    printf("Vertex\tLength\tPred\n");
    for (int i=0; i<n; i++) {
        printf("%d\t%d\t%d\n", arr[i].vertex, arr[i].length,
arr[i].pred);
    }
    printf("\n");

    free(arr);
}

void bfs(node *adj_list, int n, int start_vertex) {
    int *queue = NULL, curr_vertex, neighbour;
    int  queue_end = -1;
    node *side_ptr, *down_ptr;
```

```c
    bool *visited = NULL;

    queue = malloc(n*sizeof(int));
    visited = calloc(n,sizeof(bool)); // calloc init to 0
(false)
    if (queue == NULL || visited == NULL) {
        printf("MEMORY ALLOCATION ERROR.\n");
        exit(-1);
    }
    queue_end = 0;
    // queue_start = 0;

    queue[queue_end] = start_vertex;
    visited[start_vertex] = true;

    printf("BFS:\n");
    while(queue_end >= 0) {
        curr_vertex = queue[0];
        for (int i=0; i<queue_end; i++) {
            queue[i] = queue[i+1];
        }
        queue_end--;

        printf("%d, ", curr_vertex);

        // see neighbour of currvertex
        down_ptr = adj_list;
        while(down_ptr) {
```

```
            if (down_ptr->val == curr_vertex) {
                // search neighbours of this vertex
                side_ptr = down_ptr->ptr1;
                while(side_ptr) {
                    neighbour = side_ptr->val;
                    // add to stack
                    if (!visited[neighbour]) {
                        queue_end++;
                        queue[queue_end] = neighbour;
                        visited[neighbour] = true;
                    }
                    side_ptr = side_ptr->ptr1;
                }
                down_ptr = NULL; // to stop the loop
            } else {
                down_ptr = down_ptr->ptr2;
            }
        }
    }
    printf("\n");
    free(visited);
    free(queue);
}

void dfs(node *adj_list, int n, int start_vertex) {
    int *stack = NULL, stack_top = -1, curr_vertex,
neighbour;
    node *side_ptr, *down_ptr;
```

```
    bool *visited = NULL;

    stack = malloc(n*sizeof(int));
    visited = calloc(n,sizeof(bool)); // calloc init to 0
(false)
    if (stack == NULL || visited == NULL) {
        printf("MEMORY ALLOCATION ERROR.\n");
        exit(-1);
    }
    stack_top = 0;

    stack[stack_top] = start_vertex;
    visited[start_vertex] = true;

    printf("DFS:\n");
    while(stack_top >= 0) {
        curr_vertex = stack[stack_top--];
        printf("%d, ", curr_vertex);

        // see neighbour of currvertex
        down_ptr = adj_list;
        while(down_ptr) {
            if (down_ptr->val == curr_vertex) {
                // search neighbours of this vertex
                side_ptr = down_ptr->ptr1;
                while(side_ptr) {
                    neighbour = side_ptr->val;
                    // add to stack
```

```
                if (!visited[neighbour]) {
                    stack_top++;
                    stack[stack_top] = neighbour;
                    visited[neighbour] = true;
                }
                side_ptr = side_ptr->ptr1;
            }
            down_ptr = NULL; // to stop the loop
        } else {
            down_ptr = down_ptr->ptr2;
        }
    }
  }
  printf("\n");
  free(visited);
  free(stack);
}


node *create_node(int val, int weight) {
  node *tmp = malloc(sizeof(node));
  if (tmp == NULL) {
      printf("MEMORY ALLOCATION ERROR.\n");
      exit(-1);
  }
  tmp->val = val;
  tmp->weight = weight;
  tmp->ptr1 = NULL;
```

```
    tmp->ptr2 = NULL;
    return tmp;
}

node *create_adj_list(int **adj_matrix, int *vertex_list,
int n) {
    node *head = NULL, *down_ptr = NULL, *side_ptr = NULL,
*tmp = NULL, *tmp2 = NULL;

    for (int i=0; i<n; i++) {
        tmp = create_node(vertex_list[i], 0);
        // add edges
        side_ptr = tmp;
        for (int j=0; j<n; j++) {
            // edge => adj_matrix[i][j] == 1 or any number
(weight)
            // vertex_name = vertex_list[j]
            if (adj_matrix[i][j] != 0) {
                // add the vertex in adj_list
                tmp2 = create_node(vertex_list[j],
adj_matrix[i][j]);
                side_ptr->ptr1 = tmp2;
                side_ptr = side_ptr->ptr1;
            }
        }

        if (head == NULL) {
            head = tmp;
```

```c
            down_ptr = head;
        } else {
            down_ptr->ptr2 = tmp;
            down_ptr = down_ptr->ptr2;
        }
    }
    return head;
}

void print_adj_list(node *adj_list_head) {
    node *down_ptr = NULL, *side_ptr = NULL;

    printf("ADJACENCY LIST:\n");
    down_ptr = adj_list_head;
    while(down_ptr != NULL) {
        side_ptr = down_ptr;

        while(side_ptr != NULL) {
            printf("%d ->", side_ptr->val);
            side_ptr = side_ptr->ptr1;
        }
        printf("NULL\n");
        printf("|\nV\n");

        down_ptr = down_ptr->ptr2;
    }
    printf("NULL\n");
}
```

```c
int **input_adj_matrix(int *n) {
   int **arr = NULL, val;

   printf("Enter number of vertices: ");
   scanf("%d", n);

   arr = malloc((*n)*sizeof(int *));
   if (arr == NULL) {
       printf("MEMORY ALLOCATION ERROR.\n");
   }

   printf("Enter Adjacency Matrix:\n");
   for (int i=0; i<(*n); i++) {
       arr[i] = malloc((*n)*sizeof(int));
       if (arr[i] == NULL) {
           printf("MEMORY ALLOCATION ERROR.\n");
       }

       for (int j=0; j<(*n); j++) {
           scanf("%d,", &val);
           arr[i][j] = val;
       }
   }
   clr_buffr();

   return arr;
}
```

```c
int *input_vertex_arr(int n) {
   int val;
   int *vertex_arr = malloc(n*sizeof(char));
   if (vertex_arr == NULL) {
       printf("MEMORY ALLOCATION ERROR.\n");
   }

   printf("Enter vertex names: ");
   for (int i=0; i<n; i++) {
       scanf("%d,", &val);
       vertex_arr[i] = val;
   }
   clr_buffr();

   return vertex_arr;
}

void add_vertex(node *head, int n) {
   node *down_ptr = NULL, *side_ptr = NULL;
   bool vertex_exists = false;

   if (head == NULL) {
       printf("Graph is empty.\n");
       return;
   }

   down_ptr = head;
```

```
    do {
        if (down_ptr->val == n) {
            vertex_exists = true;
            break;
        }
        down_ptr = down_ptr->ptr2;
    } while(down_ptr->ptr2 != NULL);

    // down_ptr at last postion.
    // check last position
    if (vertex_exists) {
        printf("Vetex %d already exists.\n", n);
        return;
    }

    // add vertex
    if (!vertex_exists) {
        down_ptr->ptr2 = create_node(n, 0);
    }

    return;
}


void add_edge(node *head, int u, int v) {
    node *down_ptr = NULL, *side_ptr = NULL;
```

```
    bool found_u = false, found_v = false, edge_exists =
false;

    if (head == NULL) {
        printf("Graph is empty.\n");
        return;
    }

    down_ptr = head;

    // check if u and v exists
    while (down_ptr != NULL) {
        if(down_ptr->val == u) {
            found_u = true;
        }
        if(down_ptr->val == v) {
            found_v = true;
        }

        if (found_u && found_v) {
            break;
        }

        down_ptr = down_ptr->ptr2;
    }

    if (!found_u || !found_v) {
        printf("Vertices %d or %d does not exist.\n", u, v);
```

```
        return;
    }

    // add edge
    found_u = false; found_v = false; down_ptr = head;
    while (down_ptr != NULL) {
        if (down_ptr->val == u) {
            side_ptr = down_ptr->ptr1;
            if (side_ptr == NULL) {
                // u had no adjacent edges
                down_ptr->ptr1 = create_node(v, 1);
            } else {
                while(side_ptr->ptr1 != NULL) {
                    if (side_ptr->val == v) {
                        printf("Edge %d-%d already
exists.\n", u, v);
                        edge_exists = true;
                        break;
                    }
                    side_ptr = side_ptr->ptr1;
                } // reached last node
                if (!edge_exists && side_ptr->val == v) {
                    printf("Edge %d-%d already exists.\n", u,
v);
                    edge_exists = true;
                }
                // add vertex name
                if (edge_exists) {
```

```
                    break; // outer loop
                }
                side_ptr->ptr1 = create_node(v, 1);
            }
            found_u = true;
        }

        if (down_ptr->val == v) {
            side_ptr = down_ptr->ptr1;
            if (side_ptr == NULL) {
                down_ptr->ptr1 = create_node(u, 1);
            } else {
                while(side_ptr->ptr1 != NULL) {
                    if (side_ptr->val == u) {
                        printf("Edge %d-%d already
exists.\n", u, v);
                        edge_exists = true;
                        break;
                    }
                    side_ptr = side_ptr->ptr1;
                }
                // since above loop stopped at side_ptr->ptr1
== NULL
                // last node was not checked
                if (!edge_exists && side_ptr->val == u) {
                    printf("Edge %d-%d already exists.\n", u,
v);
                    edge_exists = true;
```

```
                }
                if (edge_exists) {
                    break;
                }
                side_ptr->ptr1 = create_node(u, 1);
            }
            found_v = true;
        }

        if (found_u && found_v) {
            break;
        }
        down_ptr = down_ptr->ptr2;
    }

    return;
}


void del_edge(node *head, int u, int v) {
    node *down_ptr = NULL, *side_ptr = NULL, *tmp = NULL,
*prev = NULL;
    bool deleted_from_u = false, deleted_from_v = false;

    if (head == NULL) {
        printf("Graph is empty.\n");
        return;
    }
```

```
    down_ptr = head;
    while (down_ptr != NULL) {
        if (down_ptr->val == u) {
            // search for v
            side_ptr = down_ptr->ptr1;
            prev = down_ptr; // one step behind side_ptr
            deleted_from_u = false;
            while(side_ptr != NULL) {
                if(side_ptr->val == v) {
                    // delete
                    prev->ptr1 = side_ptr->ptr1;
                    free(side_ptr);
                    deleted_from_u = true;
                    break;
                } else {
                    prev = side_ptr;
                    side_ptr = side_ptr->ptr1;
                }
            }
            if (!deleted_from_u) {
                printf("Vertex %d or Edge %d-%dnot found.\n",
v, u, v);
                break;
            }
        }

        if (down_ptr->val == v) {
```

```
            side_ptr = down_ptr->ptr1;
            prev = down_ptr;
            deleted_from_v = false;
            while(side_ptr != NULL) {
                if(side_ptr->val == u) {
                    prev->ptr1 = side_ptr->ptr1;
                    free(side_ptr);
                    deleted_from_v = true;
                    break;
                } else {
                    prev = side_ptr;
                    side_ptr = side_ptr->ptr1;
                }
            }
            if (!deleted_from_v) {
                printf("Vertex %d or Edge %d-%d not
found.\n", u, u, v);
                break;
            }
        }

        if (deleted_from_u && deleted_from_v) {
            break;
        }

        down_ptr = down_ptr->ptr2;
    }
    return;
```

```
}

node *del_vertex(node *head, int n) {
   node *down_ptr = NULL, *side_ptr = NULL, *tmp = NULL,
*prev = NULL;
   node  *side_ptr2 = NULL, *tmp2 = NULL, *prev2 = NULL;
   int val;

   if (head == NULL) {
       printf("Graph is empty.\n");
       return head;
   }

   down_ptr = head;

   while(down_ptr != NULL) {
       if (down_ptr->val == n) {
           tmp = down_ptr;
           if(prev == NULL) {
               head = down_ptr->ptr2;
           } else {
               prev->ptr2 = down_ptr->ptr2;
           }
           break;
       }
       prev = down_ptr;
       down_ptr = down_ptr->ptr2;
   }
```

```
if (tmp == NULL) {
    printf("Vertex %d not found.\n", n);
    return head;
}

side_ptr = tmp->ptr1;
prev = tmp;
while(side_ptr != NULL) {
    prev->ptr1 = side_ptr->ptr1;
    tmp2 = side_ptr;
    val = side_ptr->val;
    side_ptr = side_ptr->ptr1;
    free(tmp2);

    down_ptr = head;
    while(down_ptr != NULL) {
        if (down_ptr->val == val) {
            // find n in list list
            side_ptr2 = down_ptr->ptr1;
            prev2 = down_ptr; // one step behind side_ptr
            while(side_ptr2 != NULL) {
                if (side_ptr2->val == n) {
                    prev2->ptr1 = side_ptr2->ptr1;
                    free(side_ptr2);
                    break;
                }
                prev2 = side_ptr2;
```

```
                    side_ptr2 = side_ptr2->ptr1;
                }
                break;
            }
            down_ptr = down_ptr->ptr2;
        }
    }
    free(tmp);

    return head;
}

void clr_buffr() {
    char x;
    do {
        x = getchar();
    } while(x != '\n');
}
```

OUTPUTS:

Prog1_4.c

```
Enter number of vertices: 4
Enter Adjacency Matrix:
0,1,1,0
1,0,1,0
```

```
1,1,0,1
0,0,1,0
Enter vertex names: 0,1,2,3
ADJACENCY LIST:
0 ->1 ->2 ->NULL
|
V
1 ->0 ->2 ->NULL
|
V
2 ->0 ->1 ->3 ->NULL
|
V
3 ->2 ->NULL
|
V
NULL

Select Option
1. Add vertex
2. Add edge
3. Delete edge
4. Delete vertex
0. Exit
enter choice: 1
Enter vertex name: 4
ADJACENCY LIST:
0 ->1 ->2 ->NULL
|
V
1 ->0 ->2 ->NULL
|
V
```

```
2 ->0 ->1 ->3 ->NULL
|
V
3 ->2 ->NULL
|
V
4 ->NULL
|
V
NULL

Select Option
1. Add vertex
2. Add edge
3. Delete edge
4. Delete vertex
0. Exit
enter choice: 2
Enter vertices of edge (comma seperated u,v): 3,4
ADJACENCY LIST:
0 ->1 ->2 ->NULL
|
V
1 ->0 ->2 ->NULL
|
V
2 ->0 ->1 ->3 ->NULL
|
V
3 ->2 ->4 ->NULL
|
V
4 ->3 ->NULL
```

```
|
V
NULL

Select Option
1. Add vertex
2. Add edge
3. Delete edge
4. Delete vertex
0. Exit
enter choice: 3
Enter vertices of edge (comma seperated u,v): 4,3
ADJACENCY LIST:
0 ->1 ->2 ->NULL
|
V
1 ->0 ->2 ->NULL
|
V
2 ->0 ->1 ->3 ->NULL
|
V
3 ->2 ->NULL
|
V
4 ->NULL
|
V
NULL

Select Option
1. Add vertex
2. Add edge
```

```
3. Delete edge
4. Delete vertex
0. Exit
enter choice: 4
Enter vertex name: 4
ADJACENCY LIST:
0 ->1 ->2 ->NULL
|
V
1 ->0 ->2 ->NULL
|
V
2 ->0 ->1 ->3 ->NULL
|
V
3 ->2 ->NULL
|
V
NULL
```

prog5_8&10_11.c

```
Enter number of vertices: 6
Enter Adjacency Matrix:
0,6,2,3,10,0
6,0,0,11,0,9
2,0,0,14,8,0
3,11,14,0,7,5
10,0,8,7,0,4
0,9,0,5,4,0
Enter vertex names: 0,1,2,3,4,5
ADJACENCY LIST:
0 ->1 ->2 ->3 ->4 ->NULL
|
V
```

```
1 ->0 ->3 ->5 ->NULL
|
V
2 ->0 ->3 ->4 ->NULL
|
V
3 ->0 ->1 ->2 ->4 ->5 ->NULL
|
V
4 ->0 ->2 ->3 ->5 ->NULL
|
V
5 ->1 ->3 ->4 ->NULL
|
V
NULL
Enter start vertex (BFS, DFS, Dijkstra): 0

Select Option
1. BFS
2. DFS
3. Dijkstra's Algorithm
4. Floyd's Algorithm
5. Kruskal's Algorithm
6. Prim's Algorithm
0. Exit
enter choice: 1
BFS:
0, 1, 2, 3, 4, 5,

Select Option
1. BFS
2. DFS
3. Dijkstra's Algorithm
4. Floyd's Algorithm
5. Kruskal's Algorithm
6. Prim's Algorithm
0. Exit
```

```
enter choice: 2
DFS:
0, 4, 5, 3, 2, 1,

Select Option
1. BFS
2. DFS
3. Dijkstra's Algorithm
4. Floyd's Algorithm
5. Kruskal's Algorithm
6. Prim's Algorithm
0. Exit
enter choice: 3
DIJKSTRA'S ALGO.
Vertex  Length  Pred
0       0       -1
1       6       0
2       2       0
3       3       0
4       10      0
5       8       3


Select Option
1. BFS
2. DFS
3. Dijkstra's Algorithm
4. Floyd's Algorithm
5. Kruskal's Algorithm
6. Prim's Algorithm
0. Exit
enter choice: 4
FLOYD'S ALGORITHM
Predecessor Matrix
2       0       0       0       0       3
1       0       0       0       5       1
2       0       0       0       2       3
3       0       0       0       3       3
```

```
4       5       4       4       5       4
3       5       0       5       5       4

Distance Matrix
4       6       2       3       10      8
6       12      8       9       13      9
2       8       4       5       8       10
3       9       5       6       7       5
10      13      8       7       8       4
8       9       10      5       4       8

Select Option
1. BFS
2. DFS
3. Dijkstra's Algorithm
4. Floyd's Algorithm
5. Kruskal's Algorithm
6. Prim's Algorithm
0. Exit
enter choice: 5
KRUSKAL'S ALGORITHM
u       v       Weight
0       2       2
0       3       3
4       5       4
3       5       5
0       1       6

Select Option
1. BFS
2. DFS
3. Dijkstra's Algorithm
4. Floyd's Algorithm
5. Kruskal's Algorithm
6. Prim's Algorithm
0. Exit
enter choice: 6
PRIMS ALGO.
```

```
Vertex  Length  Pred
0       0       -1
1       6       0
2       2       0
3       3       0
4       4       5
5       4       4
```

Prog9.c

```
Enter Directed Acyclic Graph:
Enter number of vertices: 4
Enter Adjacency Matrix:
0,0,1,0
1,0,1,0
0,0,0,0
1,0,1,0
Enter vertex names: 0,1,2,3
ADJACENCY LIST:
0 ->2 ->NULL
|
V
1 ->0 ->2 ->NULL
|
V
2 ->NULL
|
V
3 ->0 ->2 ->NULL
|
V
NULL
TOPOLOGICAL SORT.
1, 3, 0, 2,
```