```c
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <assert.h>

// A vertex is a 2D point
typedef struct Vertex {
        int x; // x-coordinate
        int y; // y-coordinate
} Vertex;

// each edge is a pair of vertices (end-points)
typedef struct Edge {
        Vertex *p1; // first end point
        Vertex *p2; // second end point
        int weight ;
} Edge;

//node in adjacency list
typedef struct ANode{
        Vertex *v;
        struct ANode *next_edge;
        struct VertexNode *positon;
}ANode;

//vertex in graph
typedef struct VertexNode {
        Vertex *v;
        int visit;     //1->visit;0->explored;-1->unexplored;
        struct ANode *first_edge;
        struct VertexNode *next;
} VertexNode;

typedef struct GraphRep { // graph header
        int nV; // #vertices
        int nE; // #edges
        VertexNode *vertices; //an linked list of vertices
} GraphRep;

typedef struct GraphRep *Graph;

typedef struct QNode{
        Vertex *v;
        struct QNode *next;
}QNode;

typedef struct QRep{
        int length;
        QNode *head;
        QNode *tail;
}QRep;

typedef struct QRep *Queue;

// time complexity:O(1)
Graph CreateEmptyGraph() {
        Graph g = malloc(sizeof(GraphRep));
        assert(g!=NULL);
        g->nV = 0;
        g->nE = 0;
        g->vertices = NULL;
   return g;
}

//create empty queue
Queue newQueue(){
        Queue Q = malloc(sizeof(QRep));
        Q->head =NULL;
        Q->tail=NULL;
```

```
        Q->length =0;
        return Q;
}

//insert a vertex at the end if the queue
void EnQueue(Queue Q,Vertex *value){
        QNode *new = malloc(sizeof(QNode));
        assert(new!=NULL);
        new->v =value;
        new->next = NULL;
        if(Q->tail!=NULL){     //insert new node to the tail of the queue
                Q->tail->next = new;
                Q->tail=new;
        }
        else{                           //if the queue is empty
                Q->head=new;
                Q->tail=new;
        }
        Q->length++;
}

//remove vertex from front of queue
Vertex *DeQueue(Queue Q){
        assert(Q->length>0);
        QNode *temp = Q->head;
        Q->head=Q->head->next;
        if(Q->head==NULL){   //empty queue
                Q->tail=NULL;
        }
        Q->length--;
        Vertex *value=temp->v;
        free(temp);
        return value;
}

//create a new vertex node
VertexNode *NewVNode(Vertex* value){
        VertexNode *new = malloc(sizeof(VertexNode));
        assert(new!=NULL);
        new->v=value;
        new->visit=-1;
        new->first_edge=NULL;
        return new;
}

//create a new node in adjacency list
ANode *NewANode(Vertex* value){
        ANode *new = malloc(sizeof(ANode));
        assert(new!=NULL);
        new->v=value;
        new->next_edge =NULL;
        new->positon=NULL;
        return new;
}

//check whether the edge is exist
//O(m)
int inLL(ANode *L,Vertex *n){
        if (L==NULL){
                return 0;
        }
        if(L->v->x==n->x && L->v->y==n->y){
                return 1;
        }
        return inLL(L->next_edge,n);
}

// insert node into the adjacency list
//O(m)
```

```
ANode *insertLL(ANode *L,Vertex *n){
        if(inLL(L,n)==1)  return L;
        ANode *new =NewANode(n);
        new->next_edge = L;
        return new;
}

//traverse the linked list to find a vertex node.
//O(n)
VertexNode *search(VertexNode *node,Vertex *value){
        while(node!=NULL){
                if(node->v->x==value->x && node->v->y==value->y){
                        return node;
                }
                node=node->next;
        }
        return NULL;
}

//time complexity: need to call search() for two end points of new edge,which is O(n)
//suppose m adjacent vertices for end points of edge, inLL() is O(m),
//Hence,it's O(m+n).
int InsertEdge(Graph g, Edge *e)
{
        VertexNode *new,*new2;
        ANode *temp,*temp2;
        new = search(g->vertices,e->p1);     //finding the vertex in array of vertices
        new2 = search(g->vertices,e->p2);
        if(new!=NULL){                                          //check whether the vertex is
exist
                if(!inLL(new->first_edge,e->p2)){     //check whether the edge is exist
                        temp = insertLL(new->first_edge,e->p2);
                }
                else return 0;                  // if the edge is already exist,return 0
                if(new2==NULL){                                 //when the end point of the edge
is no longer exist
                        new2 = NewVNode(e->p2);
                        new2->next=g->vertices;
                        g->vertices =new2;
                        g->nV++;
                }
                temp->positon = new2;
                new->first_edge = temp;
                temp2 = insertLL(new2->first_edge,e->p1);
                temp2->positon=new;
                new2->first_edge=temp2;
        }
        else{
                if(new2==NULL){
                        new2 = NewVNode(e->p2);
                        new2->next=g->vertices;
                        g->vertices =new2;
                        g->nV++;
                }
                else{
                        if(inLL(new2->first_edge,e->p1)) return 0;
                }
                new = NewVNode(e->p1);
                new->next=g->vertices;
                g->vertices =new;
                g->nV++;
                temp2 = insertLL(new2->first_edge,e->p1);
                temp2->positon =new;
                new2->first_edge=temp2;
                temp = insertLL(new->first_edge,e->p2);
                temp->positon = new2;
                new->first_edge=temp;
        }
```

```
        e->weight=sqrt(pow((e->p1->x-e->p2->x),2)+pow((e->p1->y-e->p2->y),2));
        g->nE+=1;
        return 1;
}

//delete node in adjacency list.
//O(m)
ANode *DeleteLL(ANode *L,Vertex* n){
        if(L==NULL)     return L;
        if(L->v->x==n->x && L->v->y==n->y) {
                L->positon=NULL;
                return L->next_edge;
        }
        L->next_edge = DeleteLL(L->next_edge,n);
        return L;
}

//time complexity: need to call search() for two end points of edge.
//suppose m adjacent vertices for end points of edge, inLL() is O(m),and recursively call
DeleteLL() m times.
//Hence,it's O(n+m).
void DeleteEdge(Graph g, Edge *e)
{
        VertexNode *new,*new2;
        new = search(g->vertices,e->p1);
        new2 = search(g->vertices,e->p2);
        if (new==NULL||new2==NULL){
                return;
        }
        if(inLL(new->first_edge,e->p2)==1 && inLL(new2->first_edge,e->p1)==1 ){
                new->first_edge=DeleteLL(new->first_edge,e->p2);
                new2->first_edge=DeleteLL(new2->first_edge,e->p1);
                g->nE--;
        }
}

//print queue in non-decreasing order.
//O(n)
void PrintQueue(Queue Q){
        QNode *head,*first,*t,*p,*q;
        first = Q->head->next->next;
        head= Q->head->next;
        head->next = NULL;
        while(first!=NULL){
                        for(t=first,q=head;(q!=NULL)&&(t->v->x>q->v->x||(t->v->x==q->v->x && t->v-
>y >= q->v->y));p=q,q=q->next);
                        first = first->next;
                        if(q==head) head=t;
                        else p->next = t;
                        t->next =q;
                }
                printf("\nreachable vertices:\n");
                while(head->next!=NULL){
                        printf("(%d,%d),",head->v->x,head->v->y);
                        head=head->next;
                }
                printf("(%d,%d)",head->v->x,head->v->y);
}

//time complexity:Suppose there are n vertices and m edges in the graph.
//Every vertex can enqueue and dequeue only once,so the maximum length of queue is n.
//for every vertex, the number of loops of inner loop equals to the degree d of this vertex --
>O(n+d).
//recall the sum of degrees equals to 2m.
//Hence,O(n(2m)+n)=O(mn)
void ReachableVertices(Graph g, Vertex *v)
{
        assert(g!=NULL);
        Vertex *pop;
```

```
        ANode *p;
        QNode *current;
        Queue q= newQueue();
        VertexNode *temp,*new,*new2,*temp2;
        temp = search(g->vertices,v);
        temp->visit=0;
        EnQueue(q,temp->v);
        if(q->length==1) current=q->head;
        while(current !=NULL){  //queue is not empty
                pop = current->v;
                new = search(g->vertices,pop);
                p = new->first_edge;
                while(p!=NULL){
                        new2 = p->positon;
                        if(new2->visit==-1){
                                new2->visit=0;
                                EnQueue(q,new2->v);
                        }
                        p=p->next_edge;
                }
                current=current->next;
        }
        PrintQueue(q);

        temp2 = g->vertices;
        while(temp2!=NULL){    //set the visit flag of each vertex to -1 after traversal.
          temp2->visit=-1;
          temp2=temp2->next;
        }

}


// Add the time complexity analysis of ShortestPath() here
void ShortestPath(Graph g, Vertex *u, Vertex *v)
{

}

//free the adjacency list.
void freeLL(ANode *L){
        if(L!=NULL){
                freeLL(L->next_edge);
                free(L);
        }

}
//time complexity:Suppose there are n vertices and m edges in the graph.
//O(m+n)
void FreeGraph(Graph g)
{
        assert(g!=NULL);
        VertexNode *node;
        node=g->vertices;
        while(node!=NULL){
                freeLL(node->first_edge);
                node = node->next;
        }
        free(g);
}

//time complexity:Suppose there are n vertices and m edges in the graph.
//similar to ReachableVertices()
//Hence,O(n(2m)+n)=O(mn)
void ShowGraph(Graph g)
{
  Vertex *pop;
  ANode *p;
  assert(g!=NULL);
```

```
   printf("Show graph:\n");
   Queue q= newQueue();
   VertexNode *temp,*new,*new2,*temp2;
   temp = g->vertices;
   while(temp!=NULL){
          if( temp->visit==-1){
                  temp->visit=0;
                  EnQueue(q,temp->v);
                  while(q->length !=0){  //queue is not empty
                         pop = DeQueue(q);
                         new = search(g->vertices,pop);
                         new->visit=1;   //explored vertex
                         p = new->first_edge;
                         while(p!=NULL){
                                 new2 = p->positon;
                                 if(new2->visit==-1){
                                         new2->visit=0;
                                         printf("(%d,%d),(%d,%d) ",new->v->x,new->v->y,new2->v-
>x,new2->v->y);

                                         EnQueue(q,new2->v);
                                 }
                                 else if(new2->visit==0){
                                         printf("(%d,%d),(%d,%d) ",new->v->x,new->v->y,new2->v-
>x,new2->v->y);
                                 }
                                 p=p->next_edge;
                         }
                  }
          }
          temp= temp->next;
   }
   temp2 = g->vertices;
   while(temp2!=NULL){    //set the visit flag of each vertex to -1 after traversal.
           temp2->visit=-1;
           temp2=temp2->next;
   }
   printf("\n");
 }

 int main() //sample main for testing
 {
         Graph g1;
          Edge *e_ptr;
          Vertex *v1, *v2;

          // Create an empty graph g1;
          g1=CreateEmptyGraph();

          // Create first connected component
          // Insert edge (0,0)-(0,10)
          e_ptr = (Edge*) malloc(sizeof(Edge));
          assert(e_ptr != NULL);
          v1=(Vertex*) malloc(sizeof(Vertex));
          assert(v1 != NULL);
          v2=(Vertex *) malloc(sizeof(Vertex));
          assert(v2 != NULL);
          v1->x=0;
          v1->y=0;
          v2->x=0;
          v2->y=10;
          e_ptr->p1=v1;
          e_ptr->p2=v2;
          if (InsertEdge(g1, e_ptr)==0) printf("edge exists\n");

          // Insert edge (0,0)-(5,6)
          e_ptr = (Edge*) malloc(sizeof(Edge));
          assert(e_ptr != NULL);
          v1=(Vertex*) malloc(sizeof(Vertex));
          assert(v1 != NULL);
```

```
        v2=(Vertex *) malloc(sizeof(Vertex));
        assert(v2 != NULL);
        v1->x=0;
        v1->y=0;
        v2->x=5;
        v2->y=6;
        e_ptr->p1=v1;
        e_ptr->p2=v2;
        if (InsertEdge(g1, e_ptr)==0) printf("edge exists\n");

        // Insert edge (0, 10)-(10, 10)
        e_ptr = (Edge*) malloc(sizeof(Edge));
        assert(e_ptr != NULL);
        v1=(Vertex*) malloc(sizeof(Vertex));
        assert(v1 != NULL);
        v2=(Vertex *) malloc(sizeof(Vertex));
        assert(v2 != NULL);
        v1->x=0;
        v1->y=10;
        v2->x=10;
        v2->y=10;
        e_ptr->p1=v1;
        e_ptr->p2=v2;
        if (InsertEdge(g1, e_ptr)==0) printf("edge exists\n");

        // Insert edge (0,10)-(5,6)
        e_ptr = (Edge*) malloc(sizeof(Edge));
        assert(e_ptr != NULL);
        v1=(Vertex*) malloc(sizeof(Vertex));
        assert(v1 != NULL);
        v2=(Vertex *) malloc(sizeof(Vertex));
        assert(v2 != NULL);
        v1->x=0;
        v1->y=10;
        v2->x=5;
        v2->y=6;
        e_ptr->p1=v1;
        e_ptr->p2=v2;
        if (InsertEdge(g1, e_ptr)==0) printf("edge exists\n");

        // Insert edge (0,0)-(5,4)
        e_ptr = (Edge*) malloc(sizeof(Edge));
        assert(e_ptr != NULL);
        v1=(Vertex*) malloc(sizeof(Vertex));
        assert(v1 != NULL);
        v2=(Vertex *) malloc(sizeof(Vertex));
        assert(v2 != NULL);
        v1->x=0;
        v1->y=0;
        v2->x=5;
        v2->y=4;
        e_ptr->p1=v1;
        e_ptr->p2=v2;
        if (InsertEdge(g1, e_ptr)==0) printf("edge exists\n");

        // Insert edge (5, 4)-(10, 4)
        e_ptr = (Edge*) malloc(sizeof(Edge));
        assert(e_ptr != NULL);
        v1=(Vertex*) malloc(sizeof(Vertex));
        assert(v1 != NULL);
        v2=(Vertex *) malloc(sizeof(Vertex));
        assert(v2 != NULL);
        v1->x=5;
        v1->y=4;
        v2->x=10;
        v2->y=4;
        e_ptr->p1=v1;
        e_ptr->p2=v2;
        if (InsertEdge(g1, e_ptr)==0) printf("edge exists\n");
```

```
    // Insert edge (5,6)-(10,6)
    e_ptr = (Edge*) malloc(sizeof(Edge));
    assert(e_ptr != NULL);
    v1=(Vertex*) malloc(sizeof(Vertex));
    assert(v1 != NULL);
    v2=(Vertex *) malloc(sizeof(Vertex));
    assert(v2 != NULL);
    v1->x=5;
    v1->y=6;
    v2->x=10;
    v2->y=6;
    e_ptr->p1=v1;
    e_ptr->p2=v2;
    if (InsertEdge(g1, e_ptr)==0) printf("edge exists\n");

    // Insert edge (10,10)-(10,6)
    e_ptr = (Edge*) malloc(sizeof(Edge));
    assert(e_ptr != NULL);
    v1=(Vertex*) malloc(sizeof(Vertex));
    assert(v1 != NULL);
    v2=(Vertex *) malloc(sizeof(Vertex));
    assert(v2 != NULL);
    v1->x=10;
    v1->y=10;
    v2->x=10;
    v2->y=6;
    e_ptr->p1=v1;
    e_ptr->p2=v2;
    if (InsertEdge(g1, e_ptr)==0) printf("edge exists\n");

    // Insert edge (10, 6)-(10, 4)
    e_ptr = (Edge*) malloc(sizeof(Edge));
    assert(e_ptr != NULL);
    v1=(Vertex*) malloc(sizeof(Vertex));
    assert(v1 != NULL);
    v2=(Vertex *) malloc(sizeof(Vertex));
    assert(v2 != NULL);
    v1->x=10;
    v1->y=6;
    v2->x=10;
    v2->y=4;
    e_ptr->p1=v1;
    e_ptr->p2=v2;
    if (InsertEdge(g1, e_ptr)==0) printf("edge exists\n");

    // Create second connected component
    // Insert edge (20,4)-(20,10)
    e_ptr = (Edge*) malloc(sizeof(Edge));
    assert(e_ptr != NULL);
    v1=(Vertex*) malloc(sizeof(Vertex));
    assert(v1 != NULL);
    v2=(Vertex *) malloc(sizeof(Vertex));
    assert(v2 != NULL);
    v1->x=20;
    v1->y=4;
    v2->x=20;
    v2->y=10;
    e_ptr->p1=v1;
    e_ptr->p2=v2;
    if (InsertEdge(g1, e_ptr)==0) printf("edge exists\n");

    // Insert edge (20,10)-(30,10)
    e_ptr = (Edge*) malloc(sizeof(Edge));
    assert(e_ptr != NULL);
    v1=(Vertex*) malloc(sizeof(Vertex));
    assert(v1 != NULL);
    v2=(Vertex *) malloc(sizeof(Vertex));
    assert(v2 != NULL);
```

```
	v1->x=20;
	v1->y=10;
	v2->x=30;
	v2->y=10;
	e_ptr->p1=v1;
	e_ptr->p2=v2;
	if (InsertEdge(g1, e_ptr)==0) printf("edge exists\n");

	// Insert edge (25,5)-(30,10)
	e_ptr = (Edge*) malloc(sizeof(Edge));
	assert(e_ptr != NULL);
	v1=(Vertex*) malloc(sizeof(Vertex));
	assert(v1 != NULL);
	v2=(Vertex *) malloc(sizeof(Vertex));
	assert(v2 != NULL);
	v1->x=25;
	v1->y=5;
	v2->x=30;
	v2->y=10;
	e_ptr->p1=v1;
	e_ptr->p2=v2;
	if (InsertEdge(g1, e_ptr)==0) printf("edge exists\n");

	//Display graph g1
	ShowGraph(g1);

	// Find the shortest path between (0,0) and (10,6)
	v1=(Vertex*) malloc(sizeof(Vertex));
	assert(v1 != NULL);
	v2=(Vertex *) malloc(sizeof(Vertex));
	assert(v2 != NULL);
	v1->x=0;
	v1->y=0;
	v2->x=10;
	v2->y=6;
	ShortestPath(g1, v1, v2);
	free(v1);
	free(v2);

	// Delete edge (0,0)-(5, 6)
	e_ptr = (Edge*) malloc(sizeof(Edge));
	assert(e_ptr != NULL);
	v1=(Vertex*) malloc(sizeof(Vertex));
	assert(v1 != NULL);
	v2=(Vertex *) malloc(sizeof(Vertex));
	assert(v2 != NULL);
	v1->x=0;
	v1->y=0;
	v2->x=5;
	v2->y=6;
	e_ptr->p1=v1;
	e_ptr->p2=v2;
	DeleteEdge(g1, e_ptr);
	free(e_ptr);
	free(v1);
	free(v2);

	// Display graph g1
	ShowGraph(g1);

	// Find the shortest path between (0,0) and (10,6)
	v1=(Vertex*) malloc(sizeof(Vertex));
	assert(v1 != NULL);
	v2=(Vertex *) malloc(sizeof(Vertex));
	assert(v2 != NULL);
	v1->x=0;
	v1->y=0;
	v2->x=10;
	v2->y=6;
```

```
        ShortestPath(g1, v1, v2);
        free(v1);
        free(v2);

        // Find the shortest path between (0,0) and (25,5)
        v1=(Vertex*) malloc(sizeof(Vertex));
        assert(v1 != NULL);
        v2=(Vertex *) malloc(sizeof(Vertex));
        assert(v2 != NULL);
        v1->x=0;
        v1->y=0;
        v2->x=25;
        v2->y=5;
        ShortestPath(g1, v1, v2);
        free(v1);
        free(v2);

        // Find reachable vertices of (0,0)
        v1=(Vertex*) malloc(sizeof(Vertex));
        assert(v1 != NULL);
        v1->x=0;
        v1->y=0;
        ReachableVertices(g1, v1);
        free(v1);

        // Find reachable vertices of (20,4)
        v1=(Vertex*) malloc(sizeof(Vertex));
        assert(v1 != NULL);
        v1->x=20;
        v1->y=4;
        ReachableVertices(g1, v1);
        free(v1);

        // Free graph g1
        FreeGraph(g1);

        return 0;
}
```