```c
/*
 * list.h
 *
 *  Created on: 22 αιεπ 2016
 *      Author: Shirel_local
 */

#ifndef LIST_H_
#define LIST_H_

typedef struct List_* PList;
typedef void* PElem;
typedef enum{FAIL, SUCCESS} Result;

/*User functions*/
typedef PElem (*CLONE_FUNC)(PElem);
typedef void (*DESTROY_FUNC)(PElem);

/*Interface functions*/
PList ListCreate(CLONE_FUNC, DESTROY_FUNC);
void ListDestroy(PList);
Result ListAdd(PList, PElem);
PElem ListGetFirst(PList);
PElem ListGetNext(PList);

Result ListRemove(PList);
int ListGetSize(PList);

#endif
```

```c
/*
* list.c
*/

#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include "list.h"

typedef struct Node_t { // linked list nodes
    PElem current_elem;
    struct Node_t* next;

} NODE, *PNode;

typedef struct List_  {// linked list struct
    PNode Iterator;
    PNode head;
    int LSize; // num of nodes
    CLONE_FUNC cloneFunc;
    DESTROY_FUNC destroyFunc;

} List;


//**************************************************************************************************
**********
//* function name:  ListCreate
//* Description  :  Given a clone and destroy functions from the user, the fucntion creates a
general ADT list
//* Parameters   :  cloneFunc - function given by user to copy
//*                 destroyFunc - function given by user to destroy
//* Return Value :  PList - a pointer to the list
//**************************************************************************************************
**********
PList ListCreate(CLONE_FUNC cloneFunc, DESTROY_FUNC destroyFunc) {
    PList pList = NULL;
    if ((pList = (PList)malloc(sizeof(List)))) {
        pList->Iterator = NULL;
        pList->head = NULL;
        pList->LSize = 0;
        pList->cloneFunc = cloneFunc;
        pList->destroyFunc = destroyFunc;
    }
    return pList;
}

//**************************************************************************************************
**********
//* function name:  ListDestroy
//* Description  :  Destroys and free alloced memory to a given list
//* Parameters   :  pList - a pointer to a list
//* Return Value :  None
//**************************************************************************************************
**********
void ListDestroy(PList pList) {
    if ((pList == NULL)) return; // no list or empty list
    while (pList->head) {
        pList->destroyFunc(pList->head->current_elem);
        pList->Iterator = pList->head;
        pList->head = pList->head->next;
        free(pList->Iterator);
        (pList->LSize)--;
    }
```

```
    free(pList);
}

//************************************************************************************************
**********
//* function name:  ListAdd
//* Description  :  Gets an element to copy in the given list
//* Parameters   :  pList - a pointer to the list
//*                 pElem - a poiinter to the element that needs to be COPIED to the list
//* Return Value :  SUCCESS or FAIL - adding the element to the list ( FAIL can be caused by
invalid parameters
//*                 or failing to allocate memory)
//************************************************************************************************
**********
Result ListAdd(PList pList, PElem pElem) {
    if ((pList == NULL) || (pElem == NULL)) return FAIL;
    PElem new_elem = pList->cloneFunc(pElem);
    if (new_elem == NULL) return FAIL; // failed to clone the elem
    PNode new_node;
    if ((new_node = (PNode)malloc(sizeof(NODE))) == NULL) {
        free(new_elem);
        return FAIL; // failed to alloc a Node;
    }
    new_node->current_elem = new_elem;
    new_node->next = pList->head;
    pList->head = new_node;
    (pList->LSize)++;
    pList->Iterator = NULL;
    return SUCCESS;
}

//************************************************************************************************
**********
//* function name:  ListGetFirst
//* Description  :  points iterator to the head of the list and returns a pointer to it
//* Parameters   :  pList - a pointer to the list
//* Return Value :  PElem - pointer to the first element in the list
//************************************************************************************************
**********
PElem ListGetFirst(PList pList) {
    if ((pList == NULL) || (pList->LSize == 0))
        return NULL; // no list or empty list
    pList->Iterator = pList->head;
    return pList->Iterator->current_elem;

}

//************************************************************************************************
**********
//* function name:  ListGetNext
//* Description  :  returns a pointer to the next elem in the list.
//* Parameters   :  pList - a pointer to the list
//* Return Value :  PElem - pointer to the next element in the list(NULL if fails / pointing at
NULL / list ended)
//************************************************************************************************
**********
PElem ListGetNext(PList pList) {
    if ((pList == NULL) || (pList->Iterator == NULL))
        return NULL; // no list or iterator is pointing at NULL
    pList->Iterator = pList->Iterator->next;
    if ((pList->Iterator == NULL))
        return NULL;
    return pList->Iterator->current_elem;
}
```

```
//************************************************************************************************
**********
//* function name:  ListRemove
//* Description  :  Deletes the node the iterator currently pointing at
//* Parameters   :  pList - a pointer to the list
//* Return Value :  SUCCESS or FAIL - SUCCESS if node is deleted or FAIL if not(iterator at
NULL or deleting failed
//************************************************************************************************
**********
Result ListRemove(PList pList) {
    if ((pList == NULL) || (pList->Iterator == NULL) || (pList->head == NULL))
        return FAIL; // no list/empty list or illegal iterator
    if (pList->head == pList->Iterator) { // case it's the head of the list
        pList->destroyFunc(pList->Iterator->current_elem);
        pList->head = pList->Iterator->next;
        free(pList->Iterator);
        pList->Iterator = NULL;
        (pList->LSize)--;
        return SUCCESS;
    }
    PNode tmp = pList->head;
    while (tmp->next != pList->Iterator) { // advance to the one before the requested node
        tmp = tmp->next;
    }
    pList->destroyFunc(pList->Iterator->current_elem);
    tmp->next = pList->Iterator->next;
    free(pList->Iterator);
    pList->Iterator = NULL;
    (pList->LSize)--;
    return SUCCESS;
}

//************************************************************************************************
**********
//* function name:  ListGetSize
//* Description  :  return the size of the list(in int)
//* Parameters   :  pList - a pointer to the list
//* Return Value :  List Size - (int)
//************************************************************************************************
**********
int ListGetSize(PList pList) {
    if (pList == NULL)
        return 0; // no list or empty list
    return pList->LSize;
}
```

```c
/*
 * graph.h
 *
 *  Created on: 22 αιεπ 2016
 *      Author: Shirel_local
 */

#ifndef GRAPH_H_
#define GRAPH_H_

typedef struct _Graph* PGraph;

typedef struct _Vertex
{
    int serialNumber;
} Vertex;

typedef struct _Edge
{
    struct _Vertex* nodeA;
    struct _Vertex* nodeB;
    int weight;
} Edge;

typedef struct _Vertex* PVertex;
typedef struct _Edge* PEdge;

PGraph GraphCreate();
void GraphDestroy(PGraph);
Bool GraphAddVertex(PGraph, int);
Bool GraphAddEdge(PGraph pGraph, int vertex1, int vertex2, int weight);
PSet GraphNeighborVertices(PGraph, int);
Bool GraphFindShortestPath(PGraph pGraph, int source, int* dist, int* prev);

int GraphGetNumberOfEdges(PGraph);
int GraphGetNumberOfVertices(PGraph);

PSet GraphVerticesStatus(PGraph);
PSet GraphEdgesStatus(PGraph);

#endif /* GRAPH_H_ */
```

```c
/*
* graph.c
*/

#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <limits.h>
#include "list.h"
#include "set.h"
#include "graph.h"

typedef struct _Graph {
    PSet Vertices;
    PSet Edges;
} Graph;

//**************************************************************************************************
**********
//*                                        Vertices Functions (compare, clone, destroy)
//**************************************************************************************************
**********



//**************************************************************************************************
**********
//* function name:  Compare_Vertices
//* Description  :  Given two pointers to vertices, cast them to "PVertex" and check wheter
they are equal or not
//* Parameters   :  pVertex1 - a pointer to the first vertix
//*                 pVertex2 - a pointer to the second vertix
//* Return Value :  TRUE or FALSE (TRUE if equal or FALSE if aren't or illegal parameters)
//**************************************************************************************************
**********
Bool Compare_Vertices(PElem pVertex1, PElem pVertex2) {
    if ((pVertex1 == NULL) || (pVertex2 == NULL))
        return FALSE;
    PVertex p1 = (PVertex)pVertex1;
    PVertex p2 = (PVertex)pVertex2;
    if (p1->serialNumber == p2->serialNumber)
        return TRUE;
    return FALSE;
}

//**************************************************************************************************
**********
//* function name:  cloneVertex
//* Description  :  Given a pointer to a vertex, clones it and return a pointer to the new one.
//* Parameters   :  pVertex - a pointer to the vertex to be cloned
//* Return Value :  PElem - a pointer to the cloned vertex
//**************************************************************************************************
**********
PElem cloneVertex(PElem pVertex) {
    if ((pVertex == NULL))
        return NULL;
    PVertex newVertex;
    if ((newVertex = (PVertex)malloc(sizeof(Vertex)))) {
        newVertex->serialNumber = ((PVertex)pVertex)->serialNumber;
        return (PElem)newVertex;
    }

    return NULL; // failed to allocate
}
```

```
//************************************************************************************************
**********
//* function name:  destroyVertex
//* Description  :  Given a pointer to a vertex and destroying it(free the memory)
//* Parameters   :  pVertex - a pointer to the vertex to be destroyed
//* Return Value :  None
//************************************************************************************************
**********
void destroyVertex(PElem pVertex) {
    if ((pVertex == NULL))
        return;
    pVertex = (PVertex)pVertex;
    free(pVertex);
}

//************************************************************************************************
**********
//*                                      Edges Functions (compare, clone, destroy)
//************************************************************************************************
**********

//************************************************************************************************
**********
//* function name:  Compare_Edges
//* Description  :  Given two pointers to edges, cast them to "PEdge" and check wheter they are
equal or not
//*                 by the serial numbers of the connected Vertices. (note: ij edge is
identical to ji edge)
//* Parameters   :  pEdge1 - a pointer to the first edge
//*                 pEdge2 - a pointer to the second edge
//* Return Value :  TRUE or FALSE (TRUE if equal or FALSE if aren't or illegal parameters)
//************************************************************************************************
**********
Bool Compare_Edges(PElem pEdge1, PElem pEdge2) {
    if ((pEdge1 == NULL) || (pEdge2 == NULL))
        return FALSE;
    PEdge p1 = (PEdge)pEdge1;
    PEdge p2 = (PEdge)pEdge2;
    if (((p1->nodeA->serialNumber == p2->nodeA->serialNumber) && (p1->nodeB->serialNumber ==
    p2->nodeB->serialNumber)) ||
        ((p1->nodeA->serialNumber == p2->nodeB->serialNumber) && (p1->nodeB->serialNumber ==
        p2->nodeA->serialNumber)))
        return TRUE;
    return FALSE;
}

//************************************************************************************************
**********
//* function name:  cloneEdge
//* Description  :  Given a pointer to a edge, clones it and return a pointer to the new one.
//* Parameters   :  pEdge - a pointer to the vertex to be cloned
//* Return Value :  PElem - a pointer to the cloned edge
//************************************************************************************************
**********
PElem cloneEdge(PElem pEdge) {
    if ((pEdge == NULL))
        return NULL;
    PEdge newEdge;
    if ((newEdge = (PEdge)malloc(sizeof(Edge)))) {
        PVertex new_nodeA;
        if ((new_nodeA = (PVertex)malloc(sizeof(Vertex))) == NULL) { //failed to copy nodeA
            free(newEdge);
            return NULL;
        }
```

```
        new_nodeA->serialNumber = ((PEdge)pEdge)->nodeA->serialNumber;
        PVertex new_nodeB;
        if ((new_nodeB = (PVertex)malloc(sizeof(Vertex))) == NULL) { //failed to copy nodeB
            destroyVertex((PElem)new_nodeA);
            free(newEdge);
            return NULL;
        }
        new_nodeB->serialNumber = ((PEdge)pEdge)->nodeB->serialNumber;
        newEdge->nodeA = new_nodeA;
        newEdge->nodeB = new_nodeB;
        newEdge->weight = ((PEdge)pEdge)->weight;
        return (PElem)newEdge;
    }
    return NULL; // failed to allocate
}

//**************************************************************************************************
**********
//* function name:  destroyEdge
//* Description  :  Given a pointer to an Edge and destroying it(free the memory)
//* Parameters   :  pEdge - a pointer to the edge to be destroyed
//* Return Value :  None
//**************************************************************************************************
**********
void destroyEdge(PElem pEdge) {
    if ((pEdge == NULL))
        return;
    PEdge target = (PEdge)pEdge;
    destroyVertex((PElem)(target->nodeA));
    destroyVertex((PElem)(target->nodeB));
    free(target);
}


//**************************************************************************************************
**********
//*                                      Graph Functions
//**************************************************************************************************
**********

//**************************************************************************************************
**********
//* function name:  GraphCreate
//* Description  :  Creates a new empty graph and returns a pointer to it
//* Parameters   :  None
//* Return Value :  pGraph - a pointer to a new graph
//**************************************************************************************************
**********
PGraph GraphCreate() {
    PGraph pGraph = NULL;
    if((pGraph = (PGraph)malloc(sizeof(Graph)))) {
        if ((pGraph->Vertices = SetCreate(Compare_Vertices, cloneVertex, destroyVertex)) ==
        NULL) {
            free(pGraph);
            return NULL; // failed to alloc vertices
        }
        if ((pGraph->Edges = SetCreate(Compare_Edges, cloneEdge, destroyEdge)) == NULL) {
            SetDestroy(pGraph->Vertices);
            free(pGraph);
            return NULL; // failed to alloc edges
        }
    }
    return pGraph;
}
```

```
//************************************************************************************************
**********
//* function name:  GraphDestroy
//* Description  :  Destroys the graph and all items in it (freeing the data
//* Parameters   :  pGraph - a pointer to a graph
//* Return Value :  None
//************************************************************************************************
**********
void GraphDestroy(PGraph pGraph) {
    if ((pGraph == NULL)) return; // no graph given
    SetDestroy(pGraph->Vertices);
    SetDestroy(pGraph->Edges);
    free(pGraph);
}

//************************************************************************************************
**********
//* function name:  GraphAddVertex
//* Description  :  Given a pointer to a graph and serial number the function adds a vertex.
Can only insert
//*                 n+1 vertex id when the number or vertexes is n. Returns TRUE if succeeded
or FALSE if not
//* Parameters   :  pGraph - a pointer to a graph
//*                 serialNumber - Vertex serialNumber
//* Return Value :  TRUE or FALSE
//************************************************************************************************
**********
Bool GraphAddVertex(PGraph pGraph, int serialNumber) {
    if ((pGraph == NULL) || (serialNumber != SetGetSize(pGraph->Vertices)))
        return FALSE; // no graph or illegal Vertex serial number according to the ruleset
        given on HW3
    PVertex newVertex;
    if ((newVertex = (PVertex)malloc(sizeof(Vertex)))) {
        newVertex->serialNumber = serialNumber;
        if ((SetAdd(pGraph->Vertices, (PElem)newVertex))) {
            free(newVertex);
            return TRUE;
        }
        free(newVertex);
        return FALSE; // failed to add to Vertices set
    }
    return FALSE; // failed to alloc Vertex
}

//************************************************************************************************
**********
//* function name:  GraphAddEdge
//* Description  :  Connects the two given vertexes in the graph with an edge of <weight>.
Return TRUE if
//*                 succeeded or FALSE if not
//* Parameters   :  pGraph - a pointer to a graph
//*                 vertex1 - first vertex
//*                 vertex2 - second vertex
//*                 weight - the weight of the vertex
//* Return Value :  TRUE or FALSE (FALSE can be caused by giving non-existing vertexes or the
given Edge
//*                 doesn't follow the ruleset given on HW3
//************************************************************************************************
**********
Bool GraphAddEdge(PGraph pGraph, int vertex1, int vertex2, int weight) {
    if ((pGraph == NULL) || (weight < 0) || (weight > 10) || (vertex1 == vertex2))
        return FALSE; //illegal parameters
    PEdge newEdge;
```

```
    PVertex pVertex1, pVertex2;
    if ((newEdge = (PEdge)malloc(sizeof(Edge)))) {
        if ((pVertex1 = (PVertex)malloc(sizeof(Vertex))) == NULL) {
            destroyEdge((PElem)newEdge);
            return FALSE; // failed to allocate Vertex1;
        }
        pVertex1->serialNumber = vertex1;
        if ((pVertex2 = (PVertex)malloc(sizeof(Vertex))) == NULL) {
            destroyEdge((PElem)newEdge);
            return FALSE; // failed to allocate Vertex2;
        }
        pVertex2->serialNumber = vertex2;
        newEdge->nodeA = pVertex1;
        newEdge->nodeB = pVertex2;
        newEdge->weight = weight; // for Ido: here it saves the weight
        if ((SetFindElement(pGraph->Vertices, (PElem)(pVertex1))) == NULL ||
            (SetFindElement(pGraph->Vertices, (PElem)(pVertex2))) == NULL ||
            (SetFindElement(pGraph->Edges, (PElem)(newEdge))) != NULL){
            destroyEdge((PElem)newEdge);
            return FALSE; // Vertices not in graph or the edge already exists
        }
        if ((SetAdd(pGraph->Edges, (PElem)newEdge))) { //for Ido: here it saves garbage
            destroyEdge((PElem)newEdge);
            return TRUE; // successs
        }

        destroyEdge((PElem)newEdge);
    }
    return FALSE; // failed to allocate an Edge
}

//**************************************************************************************************
**********
//* function name:  GraphNeighborVertices
//* Description  :  Finds all neighbouring vertices of a given vertex in a graph. Return NULL
if fails and
//*                 can return an EMPTY set if there are no neighbours
//* Parameters   :  pGraph - a pointer to a graph
//*                 serialNumber - Vertex serialNumber
//* Return Value :  PSet of all neighbouring Vertices of vertex with serialNumber
//**************************************************************************************************
**********
PSet GraphNeighborVertices(PGraph pGraph, int serialNumber) {
    if ((pGraph == NULL))
        return NULL; // no graph
    PVertex newVertex = NULL;
    if ((newVertex = (PVertex)malloc(sizeof(Vertex)))) {
        newVertex->serialNumber = serialNumber;
        if ((SetFindElement(pGraph->Vertices, (PElem)newVertex)) == NULL) {
            destroyVertex((PElem)newVertex);
            return NULL; // no such vertex in graph
        }
        destroyVertex((PElem)newVertex);
        PSet setVertices;
        if ((setVertices = SetCreate(Compare_Vertices, cloneVertex, destroyVertex)) == NULL)
            return NULL; // set creation failed
        PEdge Edge_iterator = (PEdge)(SetGetFirst(pGraph->Edges));
        do {
            if (Edge_iterator) {
                if (Edge_iterator->nodeA->serialNumber == serialNumber) {
                    SetAdd(setVertices, (PElem)Edge_iterator->nodeB);
                }
                else if (Edge_iterator->nodeB->serialNumber == serialNumber) {
                    SetAdd(setVertices, (PElem)Edge_iterator->nodeA);
```

```
                }
            }
        } while ((Edge_iterator = (PEdge)(SetGetNext(pGraph->Edges))));
        return setVertices;
    }
    return NULL; // fail to allocate a vertex
}

//*************************************************************************************************
**********
//* function name:  getMinDistVertex
//* Description  :  The function finds the vertex with the minimum distance stored in dist
array from the set Q
//* Parameters   :  Q - a set of Vertices
//*                 *dist - a pointer to an array with distances to the source vertex
//* Return Value :  min_vertex - the current vertex with minimum distance
//*************************************************************************************************
**********
PVertex getMinDistVertex(PSet Q, int* dist) {
    if ((dist == NULL) || (Q == NULL)) return NULL; // illegal parameters
    PVertex min_vertex = NULL;
    int min = INT_MAX;
    PVertex tmp = (PVertex)SetGetFirst(Q);
    while (tmp) {
        if (dist[tmp->serialNumber] < min){
            min = dist[tmp->serialNumber];
            min_vertex = tmp;
        }
        tmp = (PVertex)SetGetNext(Q);
    }
    return min_vertex;
}

//*************************************************************************************************
**********
//* function name:  getLength
//* Description  :  The function finds the distance between two vertices on a given set Q(a
Neigbouring Set)
//* Parameters   :  pGraph - a pointer to a graph
//*                 u - a Vertex serial number
//*                 v - a Vertex serial number
//* Return Value :  length - int of the length between two neigbouring vertices(weight)
//*************************************************************************************************
**********
int getLength(PGraph pGraph, int u, int v) {
    if ((pGraph == NULL) || (u < 0 ) || (v < 0))
        return -1; // illegal parameters
    PEdge tmp = (PEdge)SetGetFirst(pGraph->Edges);
    while (tmp) {
        if (((tmp->nodeA->serialNumber == u) && (tmp->nodeB->serialNumber == v)) ||
            ((tmp->nodeA->serialNumber == v) && (tmp->nodeB->serialNumber == u)))
            return tmp->weight;
        tmp = (PEdge)SetGetNext(pGraph->Edges);
    }
    return -1; // vertices are not connected
}

//*************************************************************************************************
**********
//* function name:  GraphFindShortestPath
//* Description  :  The function finds the shortest path from a the source vertex to other
vertices.
//*                 Saves the distances in dist array and the previous vertex serial number in
prev array
```

```
//* Parameters   :  pGraph - a pointer to a graph
//*                 source - serial number of the source vetrex
//*                 *dist - a pointer to disty array
//*                 *prev - a pointer to prev array
//* Return Value :  TRUE or FALSE - if the function succeeded or failed
//**************************************************************************************************
**********
Bool GraphFindShortestPath(PGraph pGraph, int source, int* dist, int* prev) {
    if ((pGraph == NULL) || (dist == NULL) || (prev == NULL) || (source < 0)) return FALSE; //
    illegal parameters
    if (GraphGetNumberOfVertices(pGraph) <= source ) return FALSE; // No such vertex in graph

    //****************************************
    //* Dijkstra Algorithm as described in HW3 **
    //****************************************
    PSet Q = SetCreate(Compare_Vertices, cloneVertex, destroyVertex);

    PVertex v = (PVertex)SetGetFirst(pGraph->Vertices);
    while (v){
        if ((SetAdd(Q, (PElem)(v))) == FALSE) {
            SetDestroy(Q);
            return FALSE;// Failed to add to set
        }
        dist[v->serialNumber] = INT_MAX;
        prev[v->serialNumber] = -1; // -1 is illegal thus UNDEFINED
        v = (PVertex)SetGetNext(pGraph->Vertices);
    }
    dist[source] = 0;
    prev[source] = source;
    PVertex u = NULL;
    while (SetGetSize(Q) != 0) {
        if ((u = getMinDistVertex(Q, dist)) == NULL) {
            SetDestroy(Q);
            return TRUE; // case of a Not Connected Graph
        }
        int u_serial = u->serialNumber;
        PSet u_Neighbors = GraphNeighborVertices(pGraph, u_serial);
        SetRemoveElement(Q, (PElem)(u));

        v = (PVertex)SetGetFirst(u_Neighbors);
        while (v) {
            int len = getLength(pGraph, u_serial, v->serialNumber);
            int alt = ((len != -1) && (dist[u_serial] != INT_MAX)) ? dist[u_serial] + len :
            INT_MAX;
            if ((dist[v->serialNumber] == INT_MAX) || // case the Edge length is infinity
                ( alt < dist[v->serialNumber])) {
                dist[v->serialNumber] = alt;
                prev[v->serialNumber] = u_serial;
            }
            v=(PVertex)SetGetNext(u_Neighbors);
        }
        SetDestroy(u_Neighbors);
    }
    SetDestroy(Q);
    return TRUE;
}

//**************************************************************************************************
**********
//* function name:  GraphGetNumberOfEdges
//* Description  :  Returns the number or edges on a given graph
//* Parameters   :  pGraph - a pointer to a graph
//* Return Value :  INT - number of vertices in the graph (0 if failed by illegal parameters)
//**************************************************************************************************
```

```
**********
int GraphGetNumberOfEdges(PGraph pGraph) {
    if ((pGraph == NULL) || (pGraph->Edges == NULL))
        return 0; // no graph given or no Edges
    return SetGetSize(pGraph->Edges);
}

//*****************************************************************************************
**********
//* function name:  GraphGetNumberOfVertices
//* Description  :  Returns the number or vertices on a given graph
//* Parameters   :  pGraph - a pointer to a graph
//* Return Value :  INT - number of vertices in the graph (0 if failed by illegal parameters)
//*****************************************************************************************
**********
int GraphGetNumberOfVertices(PGraph pGraph) {
    if ((pGraph == NULL)|| (pGraph->Vertices == NULL))
        return 0; // no graph given or no Vertices
    return SetGetSize(pGraph->Vertices);
}

//*****************************************************************************************
**********
//* function name:  GraphVerticesStatus
//* Description  :  Given a pointer to a graph the function returns a pointer to Vertices Set
//* Parameters   :  pGraph - a pointer to a graph
//* Return Value :  Vertices - a pointer to the Vertices set
//*****************************************************************************************
**********
PSet GraphVerticesStatus(PGraph pGraph) {
    if ((pGraph == NULL))
        return NULL;
    return pGraph->Vertices;
}

//*****************************************************************************************
**********
//* function name:  GraphEdgesStatus
//* Description  :  Given a pointer to a graph the function returns a pointer to Edges Set
//* Parameters   :  pGraph - a pointer to a graph
//* Return Value :  Edges - a pointer to the Edges set
//*****************************************************************************************
**********
PSet GraphEdgesStatus(PGraph pGraph) {
    if ((pGraph == NULL))
        return NULL;
    return pGraph->Edges;
}
```

```
#!/bin/bash

input=$(<"$1")
convfile=$(echo "$input" | tr ,.: " ")
maxlen=0
for token in $convfile; do
  len=${#token}
  if (( len > maxlen ));then
    maxlen=${#token}
    longest=$token
  fi
done
printf 'The longest word is %s and its length is %d.\n' "$longest" "$maxlen"
exit 0
```