Ejercicio 1

Implementar la función crear grafo que dada una lista de vértices y una lista de aristas cree un grafo con la representación por Lista de Adyacencia.

def createGraph(List, List)

Descripción: Implementa la operación crear grafo

Entrada: LinkedList con la lista de vértices y **LinkedList** con la lista de aristas donde por cada par de elementos representa una conexión entre dos vértices.

Salida: retorna el nuevo grafo

```
def createGraph(ListA,ListV):
    SG=simpleGraph(ListA,ListV)
    DG=doubleGraph(ListA,ListV)
    return SG,DG

def simpleGraph(ListA,ListV):
    cantA=length(ListA)
    if cantA%2!=0:
        return None
    cantV=length(ListV)
    G=Array(cantV,LinkedList())
    NodeV=ListV.head
    for i in range (cantV):
        G[i]=LinkedList()
        NewNode=Node()
        NewNode.value_L=NodeV.value_L
        G[i].head=NewNode
        NodeV=NodeV.nextNode

    NodeA=ListA.head
    while NodeA!=None:
        if search_Graph(G,NodeA.value_L,NodeA.nextNode.value_L,False):
            value1=(NodeA.value_L)
            value2=NodeA.nextNode.value_L

            nodeAux1=Node()

            nodeAux1.value_L=value2
            nodeAux1.nextNode=G[value1-1].head.nextNode
            G[value1-1].head.nextNode=nodeAux1


        NodeA=NodeA.nextNode.nextNode
```

```
        return G

def doubleGraph(ListA,ListV):
        cantA=length(ListA)
        if cantA%2!=0:
                return None
        cantV=length(ListV)
        G=Array(cantV,LinkedList())
        NodeV=ListV.head
        for i in range (cantV):
                G[i]=LinkedList()
                NewNode=Node()
                NewNode.value_L=NodeV.value_L
                G[i].head=NewNode
                NodeV=NodeV.nextNode

        NodeA=ListA.head
        while NodeA!=None:
                if search_Graph(G,NodeA.value_L,NodeA.nextNode.value_L,False):
                        value1=(NodeA.value_L)
                        value2=NodeA.nextNode.value_L

                        nodeAux1=Node()

                        nodeAux1.value_L=value2
                        nodeAux1.nextNode=G[value1-1].head.nextNode
                        G[value1-1].head.nextNode=nodeAux1

                        if value1!=value2:
                                nodeAux2=Node()

                                nodeAux2.value_L=value1
                                nodeAux2.nextNode=G[value2-1].head.nextNode
                                G[value2-1].head.nextNode=nodeAux2


                NodeA=NodeA.nextNode.nextNode
        return G

def search_Graph(G,A1,A2,C):
        if A1<=len(G) and A2<=len(G) and A1<=0 and A2<=0:
                return False
        List=G[A1-1]
        LenList=length(List)-1
        cnode=List.head.nextNode
        while cnode!=None:
```

```
            if cnode.value_L==A2:
                  return False
            cnode=cnode.nextNode
      if C ==False:
            return search_Graph(G,A2,A1,True)
      return True
```

Ejercicio 2

Implementar la función que responde a la siguiente especificación.

```
def existPath(Grafo, v1, v2):
      Descripción: Implementa la operación existe camino que busca si
      existe un camino entre los vértices v1 y v2
      Entrada: Grafo con la representación de Lista de Adyacencia,
      v1 y v2 vértices en el grafo.
      Salida: retorna True si existe camino entre v1 y v2, False en
      caso contrario.

def existPath(G,V1,V2):
      found=existPathR(G,V1,V2)
      if found==True:
            return True
      else:
            found=existPathR(G,V2,V1)
      if found==True:
            return True
      else:
            return False

def existPathR(G,V1,V2):
      if V1==None or V2==None:
            return False
      cnode1=G[V1-1].head
      cnode2=G[V2-1].head
      while cnode2!=None:
            if search(G[V1-1],cnode2.value_L)!=None:
                  return True
            cnode2=cnode2.nextNode
      while cnode1!=None:
            if cnode1.nextNode!=None:
                  if     cnode1.value_L==G[V1-1].head.value_L     and
cnode1!=G[V1-1].head:
                        cnode1=cnode1.nextNode
                        continue
                  if              cnode1==G[V1-1].head             and
G[V1-1].head.value_L==G[V1-1].head.nextNode.value_L:
```

```
                        aux=cnode1.nextNode.nextNode
                        if aux==None:
                              return False
                        else:
                              cnode1=aux
                              found=existPathR(G,aux.value_L,V2)
                              if found==True:
                                    return True
                  else:

                        found=existPathR(G,cnode1.nextNode.value_L,V2)
                        if found==True:
                              return True
            cnode1=cnode1.nextNode
```

Ejercicio 3

Implementar la función que responde a la siguiente especificación.

def isConnected(Grafo):

Descripción: Implementa la operación es conexo
Entrada: Grafo con la representación de Lista de Adyacencia.
Salida: retorna **True** si existe camino entre todo par de vertices, **False** en caso contrario.

```
def isConnected(Graph):
    A=Array(len(Graph),False)
    for i in range(len(Graph)):
        A[i]=False
    A=isConnectedR(Graph,A,1)
    for i in range (len(A)):
        if A[i]==False:
            return False
    return True

def isConnectedR(Graph,A,c):
    cnode=Graph[c-1].head.nextNode
    A[c-1]=True
    while cnode!=None:
        newC=cnode.value_L
        if A[newC-1]==False:
            A=isConnectedR(Graph,A,newC)
        cnode=cnode.nextNode
    return A
```

Ejercicio 4

Implementar la función que responde a la siguiente especificación.

```
def isTree(Grafo):
      Descripción: Implementa la operación es árbol
      Entrada: Grafo con la representación de Lista de Adyacencia.
      Salida: retorna True si el grafo es un árbol.
def isTree(G):
      DG=simpleToDouble(G)
      if isConnected(DG)==False:
            return False
      vert=len(G)
      ari=0
      for i in range (len(G)):
            ari=ari+(length(G[i]))-1
      if vert-1==ari:
            return True
      else:
            return False
```

Ejercicio 5

Implementar la función que responde a la siguiente especificación.

```
def isComplete(Grafo):
      Descripción: Implementa la operación es completo
      Entrada: Grafo con la representación de Lista de Adyacencia.
      Salida: retorna True si el grafo es completo.
```

Nota: Tener en cuenta que un grafo es completo cuando existe una arista entre todo par de vértices.

```
def isComplete(G):
      for i in range (len(G)):
            if lengthSR(G[i])-1!=len(G)-1:
                  return False
      return True

def lengthSR(L):
      cnode=L.head
      if cnode==None:
            return 0
      c=1
      cnode=cnode.nextNode
      while cnode!=None:
            if cnode.value_L !=L.head.value_L:
                  c+=1
            cnode=cnode.nextNode
      return c
```

Ejercicio 6

Implementar una función que dada un grafo devuelva una lista de aristas que si se eliminan el grafo se convierte en un árbol. Respetar la siguiente especificación

```
def convertTree(Grafo)
      Descripción: Implementa la operación es convertir a árbol
      Entrada: Grafo con la representación de Lista de Adyacencia.
      Salida: LinkedList de las aristas que se pueden eliminar y el
      grafo resultante se convierte en un árbol.
```

```
def convertTree(Graph):

   delEdges = LinkedList()

   DG= simpleToDouble(Graph)

   if not isConnected(DG):
      return False

   if isTree(Graph):
      return delEdges

   cant_vertices = len(Graph)
   cant_aristas = 0

   for i in range(cant_vertices):
      cant_aristas = cant_aristas + length(Graph[i])-1

   cant_delEdges = abs(cant_aristas-cant_vertices)
   G=cloneGraph(Graph)
   for i in range (cant_vertices):
         if isTree(G)==False:
                  c=searchEdges(G,i+1,delEdges)
                  G=deleteEdges(G,delEdges,c,i+1)
   return delEdges

def searchEdges(G,V1,L):
         c=0
         for i in range (len(G)):
                  if V1-1!=i:
                           if search(G[i],V1)!=None:
                                    add(L,V1)
                                    add(L,G[i].head.value_L)
                                    c+=1
         return c
```

```
def deleteEdges(G,L,c,V1):
	Gaux=cloneGraph(G)
	printGraph(Gaux)
	printEdges(L)
	cnode=L.head
	A=Array(c,0)
	for i in range (c):
		val=cnode.value_L
		if cnode !=None:
			A[i]=cnode.value_L
			delete(Gaux[val-1],V1)
			if cnode.nextNode!=None:
				cnode=cnode.nextNode.nextNode
	cond=False
	for i in range (len(A)):
		for j in range (i,len(A)-1):
			if existPath(Gaux,A[i],A[j]):
				cond=True
				break
		if cond:
			break
	cnode=L.head
	con=0
	while cnode!=None:

		Gaux2=cloneGraph(G)
		delete(Gaux2[cnode.value_L-1],cnode.nextNode.value_L)
		Gauxx2=simpleToDouble(Gaux2)
		if isConnected(Gauxx2)==False:
			cnode=cnode.nextNode.nextNode
			con+=2
		else:
			G=Gaux2
			break
	print (con)
	for i in range ((c)*2):
		if i!=con and i!=con+1:
			delete_position(L,i)
	return G

def cloneGraph(G):
	retG=Array(len(G),LinkedList())
	for i in range (len(G)):
		retG[i]=LinkedList()
		cnode=G[i].head
		while cnode!=None:
```

```
                add(retG[i],cnode.value_L)
                cnode=cnode.nextNode
        retG[i]=invertirLista(retG[i])
    return retG
```

Parte 2

Ejercicio 7

Implementar la función que responde a la siguiente especificación.

```
def countConnections(Grafo):
    Descripción: Implementa la operación cantidad de componentes
    conexas
    Entrada: Grafo con la representación de Lista de Adyacencia.
    Salida: retorna el número de componentes conexas que componen
    el grafo.

def countConnections(G):
    DG=simpleToDouble(G)
    if isConnected(DG):
        return 1
    c=1
    discQueue=LinkedList()
    for i in range (1,len(G)):
        print (G[i].head.value_L)
        if existPath(G,1,G[i].head.value_L)==False:
            enqueue(discQueue,G[i].head.value_L)
    cantDesc=length(discQueue)
    while cantDesc>1:
        c=c+1
        V1=dequeue(discQueue)
        cantDesc=cantDesc-1
        auxQueue=LinkedList()
        while cantDesc>0:#
            V2=dequeue(discQueue)
            if existPath(G,V1,V2)==False:
                enqueue(auxQueue,V2)
            cantDesc=cantDesc-1
        cantDesc=length(auxQueue)
        discQueue=auxQueue
    return c
```

Ejercicio 8

Implementar la función que responde a la siguiente especificación.

```
def convertToBFSTree(Grafo, v):
      Descripción: Convierte un grafo en un árbol BFS
      Entrada: Grafo con la representación de Lista de Adyacencia, v
      vértice que representa la raíz del árbol
      Salida: Devuelve una Lista de Adyacencia con la representación
      BFS del grafo recibido usando v como raíz.
def convertToBFSTree(G,v):
      for i in range (len(G)):
            cnode=G[i].head
            while cnode!=None:
                  cnode.color="White"
                  cnode=cnode.nextNode
      if isConnected(G)==False:
            return

      check=LinkedList()
      cnode=G[v-1].head
      cnode.color="Grey"
      cnode.parent=None
      cnode.distance=0
      enqueue(check,cnode)
      while length(check)>0:
            parent=dequeue(check)
            parent.color="Black"
            cnode=parent.nextNode
            while cnode!=None:
                  node=G[cnode.value_L-1].head
                  if node.color=="White":
                        cnode.color="White"
                        cnode.edgeType="TreeEdge"

searchReturnNode(G[cnode.value_L-1],parent.value_L).edgeType="TreeEd
ge"
                        node.parent=parent
                        node.color="Grey"
                        node.distance=parent.distance+1
                        enqueue(check,node)
                  else:
                        if cnode.edgeType==None:
                              cnode.edgeType="BackWardsEdge"
                  cnode.color="Black"
                  cnode.parent=node.parent
                  cnode.distance=node.distance
                  cnode=cnode.nextNode
```

Ejercicio 9

Implementar la función que responde a la siguiente especificación.

```
def convertToDFSTree(Grafo, v):
      Descripción: Convierte un grafo en un árbol DFS
      Entrada: Grafo con la representación de Lista de Adyacencia, v
      vértice que representa la raíz del árbol
      Salida: Devuelve una Lista de Adyacencia con la representación
      DFS del grafo recibido usando v como raíz.
def convertToDFSTree(G,v):
      for i in range (len(G)):
            cnode=G[i].head
            while cnode!=None:
                  cnode.color="White"
                  cnode=cnode.nextNode
      if isConnected(G)==False:
            return
      cnode=G[v-1].head
      DFSTR(G,cnode,None)
def DFSTR(G,cnode,parent):
      cnode.color="Grey"
      cnode.parent=parent
      parent=cnode
      cnode=cnode.nextNode
      while cnode!=None:
            aux=G[cnode.value_L-1].head
            if aux.color=="White":
                  if cnode.edgeType==None:
                        cnode.edgeType="TreeEdge"

searchReturnNode(G[cnode.value_L-1],parent.value_L).edgeType="TreeEd
ge"
                  DFSTR(G,aux,parent)
            elif aux.color=="Grey":
                  if cnode.edgeType==None:
                        cnode.edgeType="BackwardsEdge"

searchReturnNode(G[cnode.value_L-1],parent.value_L).edgeType="Backwa
rdsEdge"
            elif aux.color=="Black":
                  if cnode.edgeType==None:
                        cnode.edgeType="FowardsEdge"

searchReturnNode(G[cnode.value_L-1],parent.value_L).edgeType="Foward
sEdge"
            cnode=cnode.nextNode
```

```
	parent.color="Black"
```

Ejercicio 10

Implementar la función que responde a la siguiente especificación.

def bestRoad(Grafo, v1, v2):

Descripción: Encuentra el mejor camino, en caso de existir, entre dos vértices.

Entrada: Grafo con la representación de Lista de Adyacencia, **v1** y **v2** vértices del grafo.

Salida: retorna la lista de vértices que representan el camino más corto entre **v1** y **v2.** La lista resultante contiene al inicio a **v1** y al final a **v2.** En caso que no exista camino se retorna la lista vacía.

```
def bestRoad(G,V1,V2):
	for i in range (len(G)):
		cnode=G[i].head
		while cnode!=None:
			cnode.color="White"
			cnode=cnode.nextNode

	check=LinkedList()
	cnode=G[V1-1].head
	cnode.color="Grey"
	cnode.parent=None
	enqueue(check,cnode)
	while length(check)>0:
		parent=dequeue(check)
		parent.color="Black"
		cnode=parent.nextNode
		while cnode!=None:
			node=G[cnode.value_L-1].head
			if node.color=="White":
				cnode.color="White"
				node.parent=parent
				node.color="Grey"
				enqueue(check,node)
			cnode.color="Black"
			cnode.parent=node.parent
			cnode=cnode.nextNode

	Way=LinkedList()
	vertex=G[V2-1].head
	while vertex!=None:
		if vertex.color=="White":
			return Way
		else:
```

```
                add(Way,vertex.value_L)
                vertex=vertex.parent
        return Way
```

Ejercicio 11

Implementar la función que responde a la siguiente especificación.

```
def isBipartite(Grafo):
        Descripción: Implementa la operación es bipartito
        Entrada: Grafo con la representación de Lista de Adyacencia.
        Salida: retorna True si el grafo es bipartito.
```

NOTA: Un grafo es **bipartito** si no tiene ciclos de longitud impar.

```
def isBipartite(G):
        if isTree(G):
                return True
        G=simpleToDouble(G)
        for i in range (len(G)):
                cnode=G[i].head
                cnode.color="Black"
                for limitLong in range (1,len(G),2):
                        if findBipartite(G,limitLong,cnode,0):
                                return False
                        for j in range (len(G)):
                                G[j].head.visited=None
                cnode.color=None
        return True

def findBipartite(G,limitLong,cnode,cont):
        node=cnode.nextNode
        cnode.visited=True
        if cont>=limitLong:
                return False
        while node!=None:
                newHead=G[node.value_L-1].head
                if newHead.color=="Black":
                        if cont+1==limitLong:
                                return True
                if newHead.visited!=True:
                        if findBipartite(G,limitLong,newHead,cont+1):
                                return True
                        newHead.visited=None
                node=node.nextNode
```

Ejercicio 12

Demuestre que si el grafo G es un árbol y se le agrega una arista nueva entre cualquier par de vértices se forma exactamente un ciclo y deja de ser un árbol.

Ejercicio 13

Demuestre que si la arista (u,v) no pertenece al árbol BFS, entonces los niveles de u y v difieren a lo sumo en 1.