# Informe de Laboratorio 06

## Tema: Estruturas de dados para grafos: Listas de adyacencia

| Nota |
|---|
| |

| Estudiante | Escuela | Asignatura |
|---|---|---|
| Reyser Julio Zapata Butron<br>rzapata@unsa.edu.pe | Escuela Profesional de Ingeniería de Sistemas | Análisis Y Diseño de Algoritmos<br>Semestre: IV<br>Código: 1702231 |

| Laboratorio | Tema | Duración |
|---|---|---|
| 06 | Estruturas de dados para grafos: Listas de adyacencia | 02 horas |

| Semestre académico | Fecha de inicio | Fecha de entrega |
|---|---|---|
| 2024 - B | 19 noviembre 2024 | 25 noviembre 2024 |

## 1. Código base

El siguiente código presentado, es la base para la realización de los ejerción propuestos, añadiendo los métodos requeridos.

```cpp
#include <iostream>
using namespace std;

typedef int vertex;
struct node;

struct node
{
  vertex w;
  node *next;
};

struct graph
{
  int V;      // Numero de vertices
  int A;      // Numero de arcos
  node **adj; // Vector de listas de adyacencia

  graph(int V) : V(V), A(0)
  {
    adj = new node *[V];
    for (int v = 0; v < V; ++v)
```

```
23      {
24        adj[v] = nullptr;
25      }
26    }
27
28    ~graph()
29    {
30      for (int v = 0; v < V; ++v)
31      {
32        node *a = adj[v];
33        while (a != nullptr)
34        {
35          node *temp = a;
36          a = a->next;
37          delete temp;
38        }
39      }
40      delete[] adj;
41    }
42  };
43
44  node *NEWnode(vertex w, node *next)
45  {
46    node *a = new node;
47    a->w = w;
48    a->next = next;
49    return a;
50  }
51
52  // Inserta un arco dirigido v-w
53  void GRAPHinsertArc(graph *G, vertex v, vertex w)
54  {
55    for (node *a = G->adj[v]; a != nullptr; a = a->next)
56    {
57      if (a->w == w)
58        return;
59    }
60    G->adj[v] = NEWnode(w, G->adj[v]);
61    G->A++;
62  }
63
64  int main()
65  {
66    graph *G = new graph(5);
67
68    GRAPHinsertArc(G, 0, 1);
69    GRAPHinsertArc(G, 0, 2);
70    GRAPHinsertArc(G, 1, 0);
71    GRAPHinsertArc(G, 2, 3);
72    GRAPHinsertArc(G, 3, 2);
73
74    delete G;
75    return 0;
76  }
```

Listing 1: base.cpp

# 2. Ejercicios Propuestos

## 2.1. Escriba una función GRAPHindeg() que calcule el grado de entrada de un vértice v de un grafo G. Escribe una función () que calcule el grado de salida de v.

```
int GRAPHindeg(graph *G, vertex v)
{
  int indeg = 0;
  for (vertex u = 0; u < G->V; ++u)
  {
    for (node *a = G->adj[u]; a != nullptr; a = a->next)
    {
      if (a->w == v)
      {
        ++indeg;
      }
    }
  }
  return indeg;
}

int GRAPHoutdeg(graph *G, vertex v)
{
  int outdeg = 0;
  for (node *a = G->adj[v]; a != nullptr; a = a->next)
  {
    ++outdeg;
  }
  return outdeg;
}
```

Listing 2: exercise1.cpp

**Ejecución del código**

```
> .\grafoDirigido.exe

[+] Grados de entrada y salida de cada vertice:
Vertice 0: Grado de entrada = 1, Grado de salida = 3
Vertice 1: Grado de entrada = 1, Grado de salida = 1
Vertice 2: Grado de entrada = 2, Grado de salida = 1
Vertice 3: Grado de entrada = 2, Grado de salida = 1
Vertice 4: Grado de entrada = 0, Grado de salida = 0
```

**`GRAPHindeg`: Grado de entrada**

Esta función calcula el número de arcos que llegan a un vértice dado $v$. El procedimiento es el siguiente:

1. Inicializa un contador: `int indeg = 0`.
2. Itera sobre todos los vértices del grafo ($u$).
3. Para cada vértice $u$, recorre su lista de adyacencia (`G->adj[u]`).
4. Comprueba si $v$ es el destino (`a->w == v`) de algún arco saliente desde $u$.
5. Si lo es, incrementa el contador `indeg`.

6. Devuelve el valor acumulado del contador, que representa el grado de entrada de $v$.

**Complejidad:** $O(V + A)$, donde $V$ es el número de vértices y $A$ es el número de arcos, porque se recorren todas las listas de adyacencia.

**`GRAPHoutdeg`: Grado de salida**

Esta función calcula el número de arcos que salen de un vértice dado $v$. El procedimiento es:

1. Inicializa un contador: `int outdeg = 0`.
2. Recorre directamente la lista de adyacencia de $v$ (`G->adj[v]`).
3. Incrementa el contador por cada arco encontrado.
4. Devuelve el valor acumulado del contador, que representa el grado de salida de $v$.

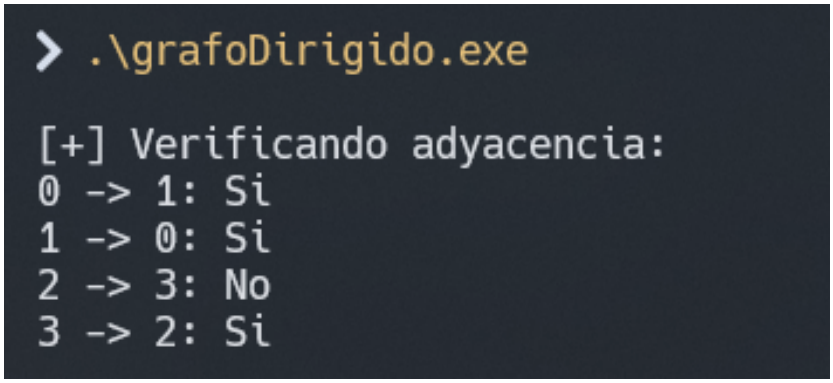
**Complejidad:** $O(\deg^+(v))$, donde $\deg^+(v)$ es el grado de salida de $v$, porque solo recorre la lista de adyacencia de $v$.

## 2.2. Consideremos el problema de decidir si dos vértices son adyacentes en un grafo G. ¿Cuánto tiempo se tarda en resolver problema? Da tu respuesta en función del número de vértices y arcos del grafo.

```cpp
bool GRAPHareAdjacent(graph *G, vertex v, vertex w)
{
  for (node *a = G->adj[v]; a != nullptr; a = a->next)
  {
    if (a->w == w)
    {
      return true;
    }
  }
  return false;
}
```

Listing 3: exercise2.cpp

**Ejecución del código**



### Descripción de la función

1. La función recibe como parámetros:
   - Un puntero al grafo $G$ (representado mediante listas de adyacencia).
   - Dos vértices $v$ y $w$.
2. Recorre la lista de adyacencia del vértice $v$ (`G->adj[v]`).
3. Para cada nodo en la lista, comprueba si el destino del arco (`a->w`) coincide con $w$.
4. Si encuentra una coincidencia, devuelve `true` (indicando que $v \to w$ es un arco existente).
5. Si recorre toda la lista sin encontrar $w$, devuelve `false`.

### Complejidad de la función

La complejidad depende del número de arcos salientes del vértice $v$ (es decir, su grado de salida, $\deg^+(v)$):

$$O(\deg^+(v))$$

En el peor caso, si $v$ tiene muchos arcos, la función deberá recorrer toda su lista de adyacencia.

## 2.3. Escribe una función GRAPHdestroy() que destruya la representación de un grafo G, liberando el espacio que la representación ocupa en memoria.

```
1  void GRAPHdestroy(graph *G)
2  {
3    for (int v = 0; v < G->V; ++v)
4    {
5      node *a = G->adj[v];
6      while (a != nullptr)
7      {
8        node *temp = a;
9        a = a->next;
10       delete temp;
11     }
12   }
13   delete[] G->adj;
14   delete G;
15 }
```

Listing 4: exercise3.cpp

### Descripción de la función

El objetivo de la función es destruir el grafo $G$ y liberar toda la memoria asignada. El procedimiento es el siguiente:

1. Recorre todos los vértices del grafo ($v$ de 0 a $G->V-1$).
2. Para cada vértice $v$:
   - Obtiene el puntero a la cabeza de su lista de adyacencia (`node *a = G->adj[v]`).
   - Recorre la lista de adyacencia y, para cada nodo:
     - Almacena un puntero temporal (`temp`) al nodo actual.

- Avanza al siguiente nodo de la lista (`a = a->next`).
- Libera la memoria del nodo actual utilizando `delete temp`.

3. Libera el vector de listas de adyacencia (`delete[] G->adj`).
4. Finalmente, libera la memoria asignada al propio grafo (`delete G`).

### Complejidad de la función

La complejidad de la función depende del número total de nodos en las listas de adyacencia. Si $A$ es el número de arcos del grafo:

$$O(V + A)$$

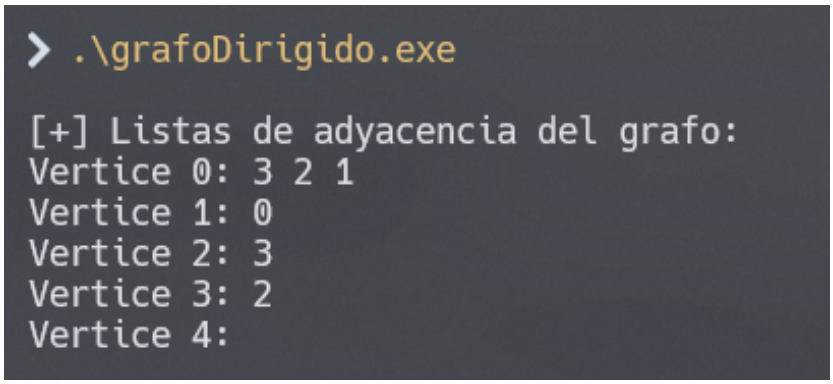
- $O(V)$: para iterar sobre todos los vértices.
- $O(A)$: para recorrer y liberar todos los nodos en las listas de adyacencia.

## 2.4. Escribe una función GRAPHshow() que imprima todos los vértices adyacentes a v en una línea para cada vértice v del grafo G.

```
void GRAPHshow(graph *G)
{
  for (int v = 0; v < G->V; ++v)
  {
    cout << "Vertice " << v << ":";
    for (node *a = G->adj[v]; a != nullptr; a = a->next)
    {
      cout << " " << a->w;
    }
    cout << endl;
  }
}
```

Listing 5: exercise4.cpp

### Ejecución del código




### Descripción de la función

El objetivo de la función es imprimir en consola las listas de adyacencia del grafo $G$. El procedimiento es el siguiente:

1. Recorre todos los vértices del grafo ($v$ de 0 a $G \to V - 1$).

2. Para cada vértice $v$:
   - Imprime el número del vértice ($cout << "Vertice" << v << " : "$).
   - Recorre su lista de adyacencia (`G->adj[v]`).
   - Para cada nodo en la lista, imprime el destino del arco almacenado en el nodo (`a->w`).
3. Después de imprimir todos los nodos adyacentes a $v$, pasa a la siguiente línea.

**Complejidad de la función**

La complejidad depende del número total de nodos en las listas de adyacencia. Si $A$ es el número de arcos del grafo:

$$O(V + A)$$

- $O(V)$: para iterar sobre todos los vértices.
- $O(A)$: para recorrer todas las listas de adyacencia.

## 2.5. Eliminación de arcos. Escriba una función GRAPHremoveArc() que tome dos vértices v y w de un grafo G representado por listas de adyacencia y elimine el arco v-w de G.

```
// Elimina el arco v-w del grafo G
void GRAPHremoveArc(graph *G, vertex v, vertex w)
{
  node *prev = nullptr;
  node *current = G->adj[v];

  while (current != nullptr)
  {
    if (current->w == w)
    {
      if (prev == nullptr)
      {
        G->adj[v] = current->next;
      }
      else
      {
        prev->next = current->next;
      }
      delete current;
      G->A--;
      return;
    }
    prev = current;
    current = current->next;
  }
}
```

Listing 6: exercise5.cpp

**Ejecución del código**

```
> .\grafoDirigido.exe

[+] Listas de adyacencia del grafo:
Vertice 0: 3 2 1
Vertice 1: 0
Vertice 2: 3
Vertice 3: 2
Vertice 4:

[+] Despues de eliminar el arco 0 -> 1:
Vertice 0: 3 2
Vertice 1: 0
Vertice 2: 3
Vertice 3: 2
Vertice 4:
```

### Descripción de la función

La función `GRAPHremoveArc` elimina un arco dirigido $v \to w$ del grafo $G$, representado mediante listas de adyacencia. El procedimiento es el siguiente:

1. Se inicializa un puntero `current` que recorre la lista de adyacencia de $v$ (`G->adj[v]`), y un puntero `prev` para mantener el nodo previo durante la iteración.
2. Se itera por los nodos de la lista:
   - Si se encuentra un nodo que contiene el destino $w$ (`current->w == w`), se actualizan los punteros para desvincular dicho nodo:
     - Si el nodo a eliminar es el primero en la lista, se ajusta la cabeza de la lista (`G->adj[v]`).
     - En caso contrario, el puntero `prev->next` se enlaza al nodo siguiente.
   - Se elimina el nodo encontrado utilizando `delete`.
   - Se decrementa el contador de arcos del grafo (`G->A`) y se termina la función con `return`.
3. Si se recorre toda la lista sin encontrar $w$, no se realiza ninguna modificación.

### Complejidad de la función

La complejidad de la función depende de la longitud de la lista de adyacencia del vértice $v$, lo cual equivale al grado de salida de $v$, denotado como $\deg^+(v)$:

$$O(\deg^+(v))$$

En el peor caso:

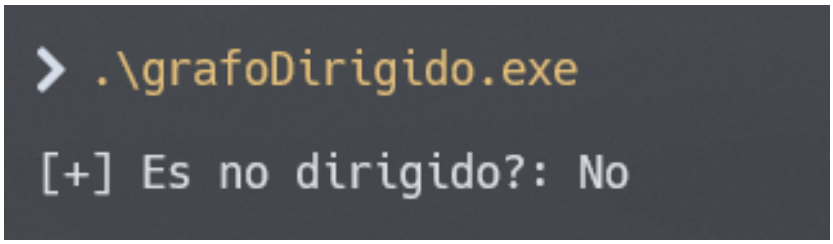
- Si $w$ no se encuentra en la lista de adyacencia, se recorrerá toda la lista.
- Si $v$ tiene muchos arcos salientes ($\deg^+(v)$ es grande), el tiempo será proporcional a la cantidad de arcos salientes.

## 2.6. ¿No dirigido? Escriba una función GRAPHundir() que decida si un grafo dado es no dirigido.

```cpp
// Verifica si el grafo G es no dirigido
bool GRAPHundir(graph *G)
{
  for (int v = 0; v < G->V; ++v)
  {
    for (node *a = G->adj[v]; a != nullptr; a = a->next)
    {
      vertex w = a->w;
      bool found = false;
      for (node *b = G->adj[w]; b != nullptr; b = b->next)
      {
        if (b->w == v)
        {
          found = true;
          break;
        }
      }
      if (!found)
      {
        return false;
      }
    }
  }
  return true;
}
```

Listing 7: exercise6.cpp

**Ejecución del código**




**Descripción de la función**

La función `GRAPHundir` verifica si un grafo dirigido $G$, representado mediante listas de adyacencia, es no dirigido. Un grafo es considerado no dirigido si, para cada arco $v \to w$, también existe el arco $w \to v$.

El procedimiento es el siguiente:

1. Se recorre cada vértice $v$ del grafo (`for (int v = 0; v <G->V; ++v)`).
2. Para cada vértice $v$, se itera sobre los nodos en su lista de adyacencia (`G->adj[v]`), obteniendo cada destino $w$ (`a->w`).
3. Para cada arco $v \to w$, se recorre la lista de adyacencia de $w$ para buscar un arco $w \to v$.
4. Si no se encuentra $w \to v$, la función devuelve `false`, indicando que el grafo no es no dirigido.
5. Si se verifican todos los vértices y arcos sin encontrar inconsistencias, la función devuelve `true`.

### Complejidad de la función

La complejidad depende del número de vértices $V$ y arcos $A$. Para cada arco $v \to w$, se busca su correspondiente arco $w \to v$ recorriendo la lista de adyacencia de $w$. Esto lleva a una complejidad:

$$O(A \cdot \Delta)$$

donde $\Delta$ es el grado máximo del grafo (la longitud máxima de una lista de adyacencia).

## 2.7. Inserción de aristas. Escribe una función UGRAPHinsertEdge() que inserte una arista v-w en un grafo no dirigido G.

```cpp
// Inserta una arista no dirigida v-w
void UGRAPHinsertEdge(graph *G, vertex v, vertex w)
{
  // Inserta el arco v -> w
  bool exists = false;
  for (node *a = G->adj[v]; a != nullptr; a = a->next)
  {
    if (a->w == w)
    {
      exists = true;
      break;
    }
  }
  if (!exists)
  {
    G->adj[v] = NEWnode(w, G->adj[v]);
    G->A++;
  }

  // Inserta el arco w -> v
  exists = false;
  for (node *a = G->adj[w]; a != nullptr; a = a->next)
  {
    if (a->w == v)
    {
      exists = true;
      break;
    }
  }
  if (!exists)
  {
    G->adj[w] = NEWnode(v, G->adj[w]);
    G->A++;
  }
}
```

Listing 8: exercise7.cpp

### Ejecución del código

```
> .\grafoNoDirigido.exe

[+] Listas de adyacencia del grafo no dirigido:
Vertice 0: 1
Vertice 1: 2 0
Vertice 2: 3 1
Vertice 3: 4 2
Vertice 4: 3
```

### Descripción de la función

La función `UGRAPHinsertEdge` inserta una arista no dirigida $v - w$ en un grafo representado mediante listas de adyacencia. Para lograrlo:

1. Verifica si el arco $v \to w$ ya existe en la lista de adyacencia de $v$. Si no, lo agrega y actualiza el contador de arcos.
2. Verifica si el arco $w \to v$ ya existe en la lista de adyacencia de $w$. Si no, lo agrega y también incrementa el contador.

De esta forma, asegura que ambos extremos de la arista estén representados.

### Complejidad de la función

La complejidad es proporcional a las longitudes de las listas de adyacencia de $v$ y $w$:

$$O(\deg^+(v) + \deg^+(w))$$

Esto la hace eficiente para grafos dispersos, pero más costosa en grafos densos con listas de adyacencia largas.

## 2.8. Eliminación de aristas. Escribe una función UGRAPHremoveEdge() que elimine una arista dada v-w de un grafo no dirigido G.

```
// Elimina una arista no dirigida v-w
void UGRAPHremoveEdge(graph *G, vertex v, vertex w)
{
  // Eliminar el arco v -> w
  node *prev = nullptr;
  node *current = G->adj[v];
  while (current != nullptr)
  {
    if (current->w == w)
    {
      if (prev == nullptr)
      {
        G->adj[v] = current->next;
      }
      else
      {
        prev->next = current->next;
      }
      delete current;
      G->A--;
      break;
    }
    prev = current;
    current = current->next;
  }

  // Eliminar el arco w -> v
  prev = nullptr;
  current = G->adj[w];
  while (current != nullptr)
  {
    if (current->w == v)
    {
      if (prev == nullptr)
      {
        G->adj[w] = current->next;
```

```
37        }
38        else
39        {
40          prev->next = current->next;
41        }
42        delete current;
43        G->A--;
44        break;
45      }
46      prev = current;
47      current = current->next;
48    }
49  }
```

Listing 9: exercise8.cpp

**Ejecución del código**

```
> .\grafoNoDirigido.exe

[+] Listas de adyacencia del grafo no dirigido:
Vertice 0: 1
Vertice 1: 2 0
Vertice 2: 3 1
Vertice 3: 4 2
Vertice 4: 3

[+] Es no dirigido?: Si

[+] Despues de eliminar la arista 1-2:
Vertice 0: 1
Vertice 1: 0
Vertice 2: 3
Vertice 3: 4 2
Vertice 4: 3
```

### Descripción de la función

La función `UGRAPHremoveEdge` elimina una arista no dirigida $v - w$ de un grafo representado mediante listas de adyacencia. Esto implica:

1. Recorrer la lista de adyacencia de $v$ para eliminar el arco $v \to w$.
2. Recorrer la lista de adyacencia de $w$ para eliminar el arco $w \to v$.
3. En ambos casos, actualiza los punteros de la lista para desvincular el nodo correspondiente y libera su memoria.
4. Decrementa el contador de arcos (`G->A`) por cada arco eliminado.

De esta forma, se garantiza que la arista no dirigida $v - w$ sea completamente eliminada.

### Complejidad de la función

La complejidad es proporcional a las longitudes de las listas de adyacencia de $v$ y $w$, ya que estas deben recorrerse para encontrar y eliminar los nodos correspondientes:

$$O(\deg^+(v) + \deg^+(w))$$

Esto la hace eficiente para grafos dispersos, pero menos eficiente en grafos densos donde las listas de adyacencia son largas.

# 3. Código main final

## 3.1. Código main final del grafo dirigido

```
int main()
{
  graph *G = new graph(5);

  GRAPHinsertArc(G, 0, 1);
  GRAPHinsertArc(G, 0, 2);
  GRAPHinsertArc(G, 0, 3);
  GRAPHinsertArc(G, 2, 3);
  GRAPHinsertArc(G, 1, 0);
  GRAPHinsertArc(G, 2, 3);
  GRAPHinsertArc(G, 3, 2);

  // Imprimir grados de entrada y salida de cada vertice
  printf("\n[+] Grados de entrada y salida de cada vertice:\n");
  for (vertex v = 0; v < G->V; ++v)
  {
    printf("Vertice %d: Grado de entrada = %d, Grado de salida = %d\n",
           v, GRAPHindeg(G, v), GRAPHoutdeg(G, v));
  }

  // Verificar adyacencia entre vertices
  printf("\n[+] Verificando adyacencia:\n");
  printf("0 -> 1: %s\n", GRAPHareAdjacent(G, 0, 1) ? "Si" : "No");
  printf("1 -> 0: %s\n", GRAPHareAdjacent(G, 1, 0) ? "Si" : "No");
  printf("2 -> 3: %s\n", GRAPHareAdjacent(G, 1, 3) ? "Si" : "No");
  printf("3 -> 2: %s\n", GRAPHareAdjacent(G, 3, 2) ? "Si" : "No");

  cout << "\n[+] Listas de adyacencia del grafo:\n";
  GRAPHshow(G);

  GRAPHremoveArc(G, 0, 1);
  cout << "\n[+] Despues de eliminar el arco 0 -> 1:\n";
  GRAPHshow(G);

  cout << "\n[+] Es no dirigido?: " << (GRAPHundir(G) ? "Si" : "No") << endl;
  cout << endl;

  GRAPHdestroy(G);

  return 0;
}
```

Listing 10: mainDirigido.cpp

## 3.2. Código main final del grafo NO dirigido

```
int main()
{
  graph *G = new graph(5);

  // Insertar aristas no dirigidas
  UGRAPHinsertEdge(G, 0, 1);
  UGRAPHinsertEdge(G, 1, 2);
```

```
  UGRAPHinsertEdge(G, 2, 3);
  UGRAPHinsertEdge(G, 3, 4);

  cout << "\n[+] Listas de adyacencia del grafo no dirigido:\n";
  GRAPHshow(G);

  cout << "\n[+] Es no dirigido?: " << (GRAPHundir(G) ? "Si" : "No") << endl;

  // Eliminar una arista
  UGRAPHremoveEdge(G, 1, 2);
  cout << "\n[+] Despues de eliminar la arista 1-2:\n";
  GRAPHshow(G);
  cout << endl;

  GRAPHdestroy(G);
  return 0;
}
```

Listing 11: mainNoDirigido.cpp

## 4. Repositorio de Github

- Repositorio de Github donde se encuentra el actual laboratorio
  `https://github.com/ReyserLynnn/ada-lab-b-24b/tree/main/laboratorio06/src`
- Repositorio de Github donde se encuentran los laboratorios del curso
  `https://github.com/ReyserLynnn/ada-lab-b-24b.git`

## 5. Conclusión

para finalizar el informe, entendí cómo manipular grafos usando listas de adyacencia, desde agregar y eliminar conexiones hasta verificar y mostrar su estructura. Analizar su complejidad y su utilidad práctica en diferentes tipos de grafos es de ayuda para comprender al 100 %.