Grafos: representación

AED II

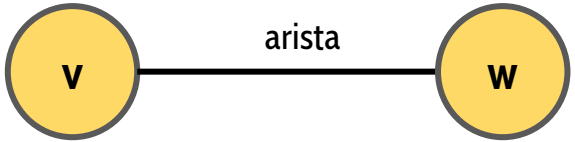
2020

UBP UNIVERSIDAD BLAS PASCAL

grafos: definición

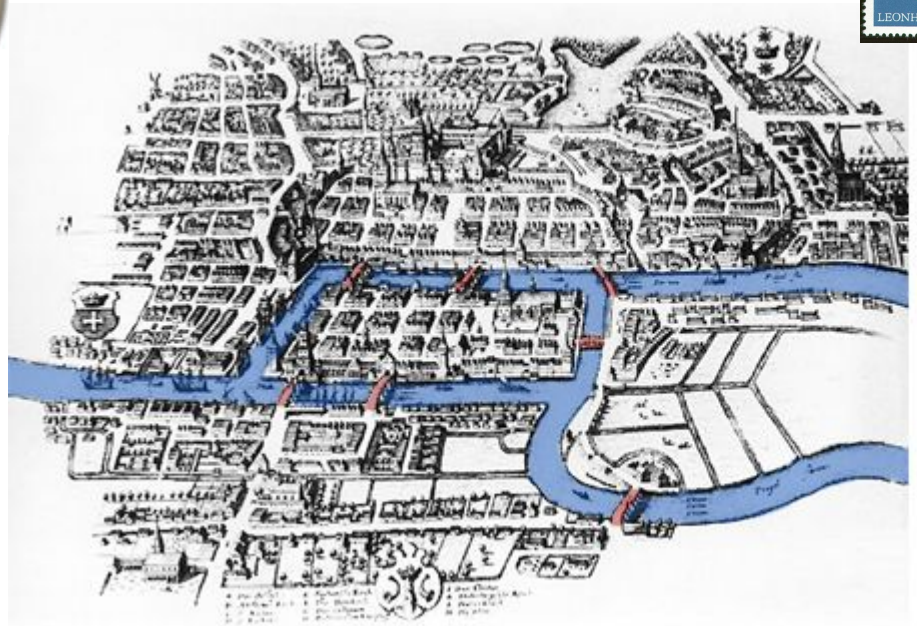
Un grafo **G** es una tupla **G= {V, A}**, donde **V** es un conjunto no vacío de vértices o nodos y **A** es un conjunto de aristas o arcos.

Cada arista es un par **(v, w)**, donde v, w ∈ V.



grafos: historia

Leonhard Euler
(1707-1783)



tipos de grafo

Grafo no dirigido: las aristas no están ordenadas

(v, w) = (w, v)

Grafos dirigidos (o digrafo): las aristas son pares ordenados

<v, w> ≠ <w, v>

<v, w> => w = cabeza de la arista, v = cola.

nodos adyacentes

Nodos **adyacentes** a un nodo v: son todos los nodos unidos a v mediante una arista.

En grafos dirigidos:

- Nodos adyacentes a **v**: todos los **w** con <v, w> ∈ A.
- Nodos adyacentes de **v**: todos los **u** con <u, v> ∈ A.

grafo ponderado

- Un grafo está etiquetado si cada arista tiene asociada una etiqueta o valor de cierto tipo.
- Grafo ponderado: grafo etiquetado cuyos valores pertenecen a un conjunto ordenado.
- Grafo etiquetado: **G**= (V, A, W), con W: A → TipoEtiqueta

camino

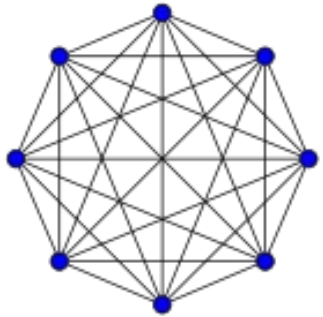
- **Camino** de un vértice w1 a wq: es una secuencia w1, w2, ..., wq ∈ V, tal que todas las aristas (w1, w2), (w2, w3), ..., (wq-1, wq) ∈ A.
- **Longitud de un camino**: número de aristas del camino = nº de nodos -1.
- **Camino simple**: aquel en el que todos los vértices son distintos (excepto el primero y el último que pueden ser iguales).
- **Ciclo**: es un camino en el cual el primer y el último vértice son iguales. En grafos no dirigidos las aristas deben ser diferentes.
- Se llama ciclo simple si el camino es simple.

grafo conexo

- Un **subgrafo** de G=[V, A] es un grafo G'=(V', A') tal que V' ⊆ V y A' ⊆ A.
- Dados dos vértices v, w, se dice que están conectados si existe un camino de v a w.
- Un grafo es **conexo** (o conectado) si hay un camino entre cualquier par de vértices.
- Si es un grafo dirigido, se llama **fuertemente conexo**.
- Un grafo dirigido es llamado **fuertemente conexo** si para cada par de vértices u y v existe un camino de u hacia v y un camino de v hacia u
- Una componente (fuertemente) conexa de un grafo G es un **subgrafo maximal** (fuertemente) conexo.

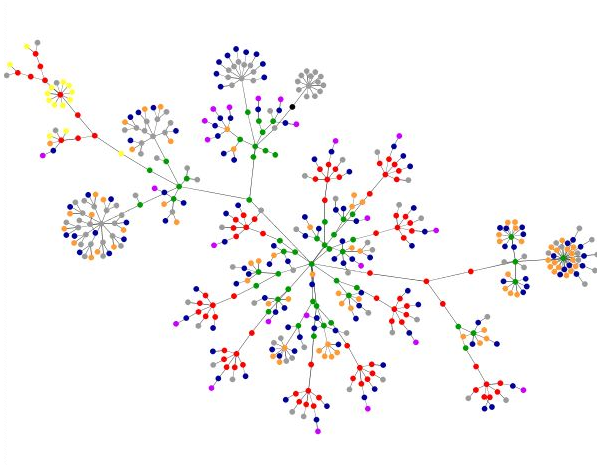
grafo completo

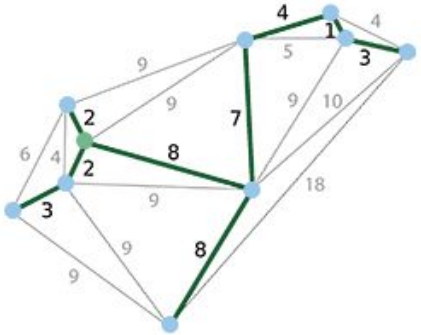
- Un grafo es **completo** si existe una arista entre cualquier par de vértices.
- Para n nodos, un grafo completo tendrá n(n-1)/2 aristas



¿y un árbol?

Un árbol es un grafo **conexo** y **acíclico**





grado

- **Grado** de un vértice v: número de arcos que inciden en él.
- Para grafos dirigidos:
  - **Grado de entrada** de v: nº de aristas con <x, v>
  - **Grado de salida** de v: nº de aristas con <v, x>

representación de grafos

- **Matriz de Adyacencia**
- **Lista de Adyacencia**
- Matriz de incidencia
- Lista de incidencia
- Otras

representación: matriz de adyacencia

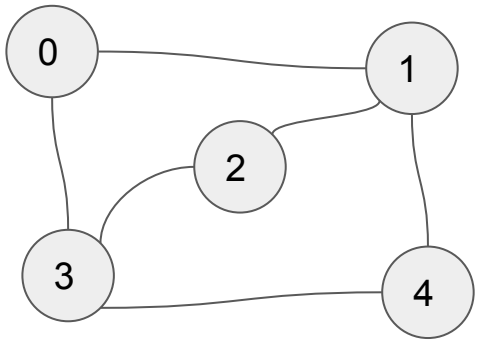
Grafos no etiquetados:

```
tipo GrafoNoEtiquetado = array [0..n-1, 0..n-1] de 0..1
```

Sea un grafo no etiquetado G=(V,A)

A[v,w] = 1 ⇔ (v,w) ∈ A

si no A[v,w] = 0



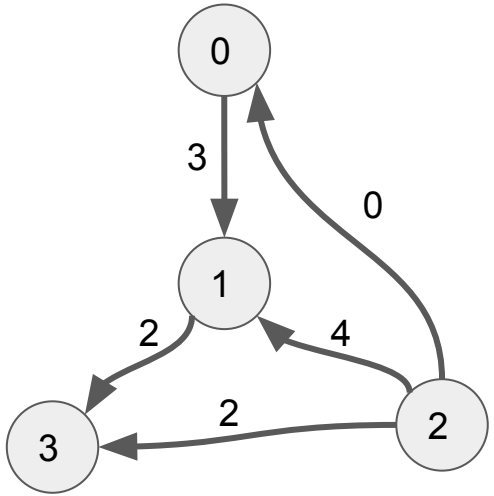
	0	1	2	3	4
0	0	1	0	1	0
1	1	0	1	0	1
2	0	1	0	1	0
3	1	0	1	0	1
4	0	1	0	1	0

representación: matriz de adyacencia

Grafos etiquetados:

```
tipo GrafoEtiquetado[E]= array [0..n-1, 0..n-1] de E
```

El tipo E tiene un valor infinito, para el caso de no existir arista.



	0	1	2	3
0	∞	3	∞	∞
1	∞	∞	∞	2
2	0	4	∞	2
3	∞	∞	∞	∞

UBP UNIVERSIDAD BLAS PASCAL

representación: matriz de adyacencia

Uso de memoria:

- Memoria usada: $O(v^2)$

Ventajas:

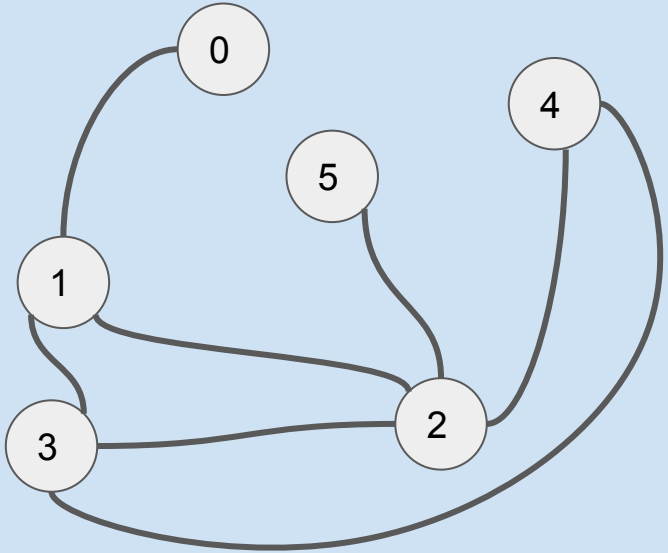
- Representación y operaciones muy sencillas.
- Remover una arista tiene costo O(1)
- Preguntas como si un nodo está conectado con otro se resuelven en O(1).

Desventajas:

- Agregar un vértice tiene un costo de $O(v^2)$.
- Si hay muchos nodos y pocas aristas ($a<<v^2$) se desperdicia mucha memoria (matriz escasa).

Ejercicio 1.01.

Representá el siguiente grafo en una matriz de adyacencia



Ejercicio 1.02.

Graficá el grafo cuya matriz de adyacencia es:

	0	1	2	3	4	5	6
0	0	1	0	1	0	1	0
1	1	0	1	0	1	0	1
2	0	1	0	0	0	1	0
3	1	0	0	0	1	0	0
4	0	1	0	1	0	0	0
5	1	0	1	0	0	0	0
6	0	1	0	0	0	0	0

Repaso
arrays en c++

arrays

En C y C++ **un array es un puntero** que apunta al primer elemento del array

```
const int N = 5;
int numeros[N] = {11, 22, 44, 21, 41};

cout << &numeros[0] << endl;
// direccion del primer elemento (0x22fef8)

cout << numeros << endl;
// lo mismo (0x22fef8)
```

aritmética de punteros

Vimos que `numeros` es un puntero al primer elemento del array, `(numeros + 1)` apunta al siguiente entero y no a la dirección contigua.

Asi:

```
int numeros[] = {11, 22, 33};
int * iPtr = numeros;
cout << iPtr << endl;          // 0x22cd30
cout << iPtr + 1 << endl;      // 0x22cd34
cout << *iPtr << endl;         // 11
cout << *(iPtr + 1) << endl; // 22
cout << *(iPtr + 2) << endl; // 33
cout << *iPtr + 1 << endl;    // 12
```

tamaño de un array

La operación `sizeof(array)` nos da el tamaño total del array. Si la dividimos por el tamaño de un elemento, podemos obtener la cantidad de elementos.

```
int numeros[100];
cout << sizeof(numeros) << endl;
cout << sizeof(numeros[0]) << endl;
cout << "Tiene " << sizeof(numeros) / sizeof(numeros[0]) <<
" elementos" << endl;
```

pasar un array a una función

Los arrays **se pasan** a una función **como un puntero** al primer elemento del array

```
int max(int numeros[], int nElementos);
int max(int *numeros, int nElementos);
int max(int numeros[50], int nElementos); //el tamaño se ignora
```

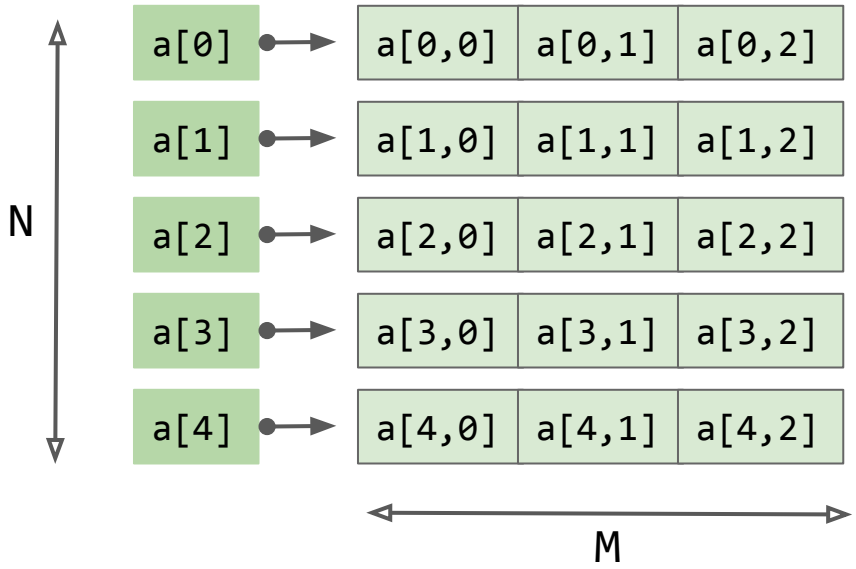
El tamaño del array no es parte del parámetro array. El compilador no tiene la capacidad de deducir el tamaño del array del puntero y no realiza verificación de límites

arrays creación dinámica: ejemplo

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main() {
   const int N = 5;
   int * pArray;
   pArray = new int[N];  // Asignación usando el operador new[]
   for (int i = 0; i < N; ++i) {
      *(pArray + i) = rand() % 100;
   }

   // Imprimir el array
   for (int i = 0; i < N; ++i) {
      cout << *(pArray + i) << " ";
   }
   cout << endl;
   delete[] pArray;  // Dealocar usando el operador delete[]
   return 0;
}
```

arrays bidimensionales en c++

Un array bidimensional en C++ es básicamente un array unidimensional de punteros. Cada puntero de este array apunta a un array unidimensional que es el que contiene los datos



array bidimensional: ejemplo

```
#include <iostream>

using namespace std;

int main () {
    const int N = 5;  // Filas
    const int M = 3;  // Columnas

    int a[N][M] = {{0,0,1}, {1,2,5}, {2,4,6}, {3,6,9},{4,8,7}};

    // Imprimir el array
    for ( int i = 0; i < N; i++ ){
      for ( int j = 0; j < M; j++ ) {
           cout << "a[" << i << "," << j << "]: " << a[i][j] << '\t';
      }
      cout<<endl;
    }
    return 0;
}
```

arrays bidimensionales dinámicos

```
const int N = 5;
const int M = 4;

int** a = new int*[N];
for (int i = 0; i < N; i++){
    a[i] = new int[M];
}
```

array bidimensional dinámico

```
const int N = 5;
const int M = 4;

// Crear
int** a = new int*[N];
for (int i = 0; i < N; i++)
   a[i] = new int[M];


// Asignar
for (int i = 0; i < N; i++)
   for (int j = 0; j < M; j++)
        a[i][j] = 1;
```

```
// Imprimir
for (int i = 0; i < N; i++){
   for (int j = 0; j < M; j++)
        cout << a[i][j] << "\t";
   cout << endl;
}


// Eliminar
for(int i = 0; i < N; i++)
   delete[] a[i];
delete[] a;
```

arrays bidimensionales como argumento

```
int suma(int** a, int n, int m){
    int s = 0;
    for (int i = 0; i < n; i++)
       for (int j = 0; j < m; j++)
          s+=a[i][j];

    return s;
}

// Pasar como argumento
cout << "La suma de sus elementos es: " << suma(a, N, M);
```

Ejercicio 1.03.

Creá un programa que:

- Permita ingresar la cantidad de filas de un array
- Permita ingresar la cantidad de columnas de un array
- Permita ingresar los elementos del array
- Imprima el array
- Imprima la suma de los elementos del array (este cálculo debe ser realizado por una función)
- Libere el espacio asignado dinámicamente antes de finalizar

Ejercicio 1.03.

Para el programa anterior calculá cuántos bytes son necesarios para almacenar el array,

Nota: Teniendo en cuenta que en las arquitecturas de 32 bits se utilizan 4 bytes para un puntero y en las de 64 bits se utilizan 8 bytes

Para saber la longitud de un int en tu entorno, podés hacer:

```
cout << "Tamaño de un int: " << sizeof(int) << " bytes" << endl;
```

Fin del Repaso

Ejercicio 1.04.

Crear la clase Grafo que implemente un grafo mediante matriz de adyacencia

Grafo
- v: int - a: int[][]
+ Grafo(n: int) + ~Grafo() + AgregarArista(origen: int, destino: int) + Mostrar()

Grafo: matriz de adyacencia

```
class Grafo {
    private:
       int v;
       int **a;
    public:
       Grafo(int);
       ~Grafo();
       void AgregarArista(int,int);
       void Mostrar();
};
```

```
Grafo::Grafo(int v){
    this->v = v;
    this->a = new int* [v];
    for (int i = 0; i < v; i++){
       a[i] = new int[n];
       for (int j = 0; j < v; j++){
            a[i][j] = 0;
       }
    }
}

Grafo::~Grafo(){
    for (int i = 0; i < v; i++){
       delete[] a[i];
    }
    delete[] a;
}
```

Grafo: matriz de adyacencia

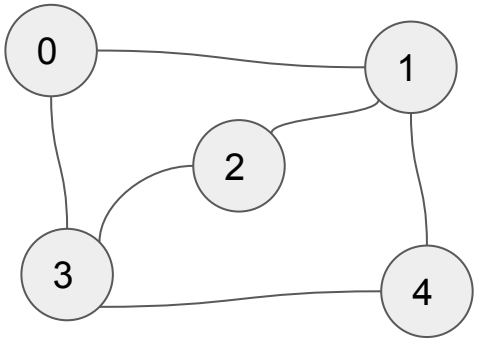
```
void Grafo::AgregarArista(int origen, int destino){
     if (origen >= 0 && destino >= 0
         && origen < v && destino < v){
           a[origen][destino] = 1;
     }
}

void Grafo::Mostrar(){
    for (int i = 0; i < v; i++){
      for (int j = 0; j < v; j++){
           cout<<a[i][j]<<" ";
      }
      cout<<endl;
    }
}
```

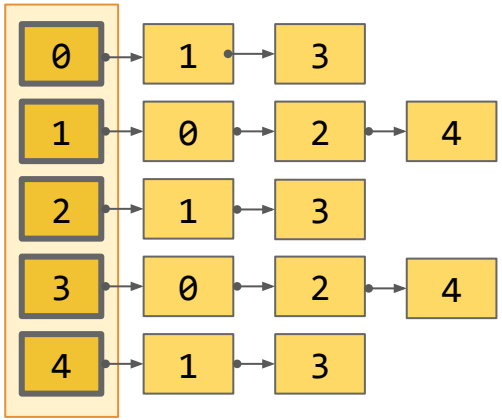
```
int main()
{
    Grafo m(4);
    m.AgregarArista(0,2);
    m.AgregarArista(0,3);
    m.AgregarArista(1,2);
    m.AgregarArista(2,3);

    m.Mostrar();

    return 0;
}
```

representación: lista de adyacencia

```
tipo Nodo= entero (1..n)

tipo GrafoNoEtiquetado = array [1..n] de Lista[Nodo]
```

Sea R de tipo GrafoNoEtiquetado, G= (V, A).
La lista R[v] contiene los w tal que (v, w) A.



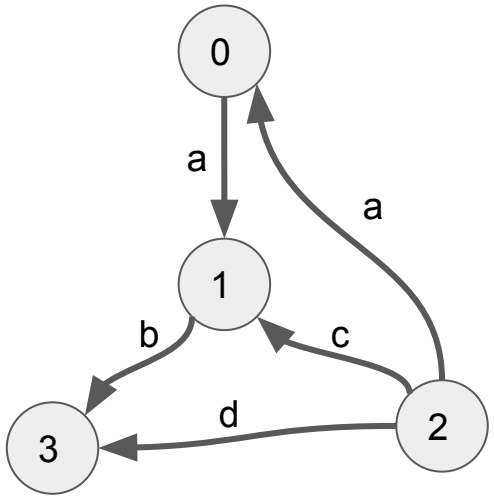


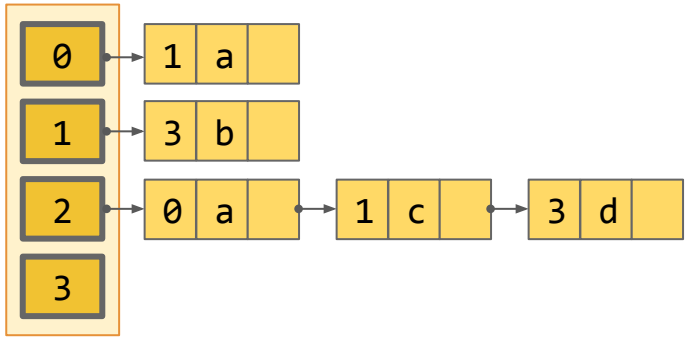
UBP UNIVERSIDAD BLAS PASCAL

representación: lista de adyacencia

Grafos etiquetados:

```
tipo  GrafoEtiquetado[E]= array [1..n] de Lista[Nodo,E]
```





UBP UNIVERSIDAD BLAS PASCAL

representación: lista de adyacencia

Uso de memoria:

- Memoria usada: O (v + a)

Ventajas:

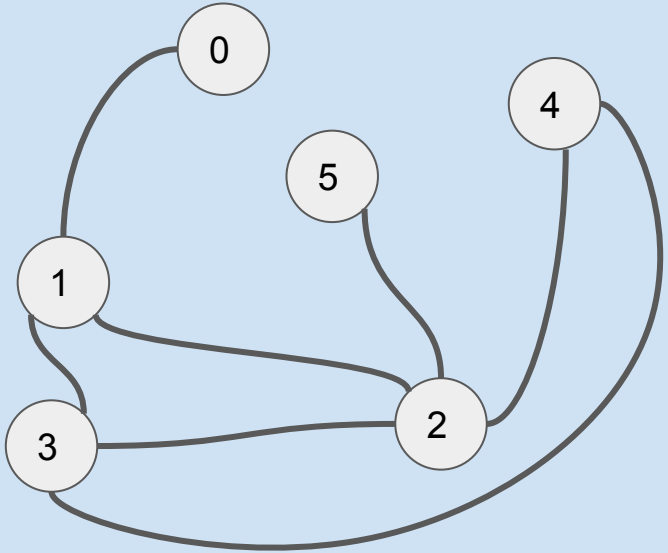
- Más adecuada cuando $a << v^2$
- Es fácil agregar un vértice

Desventajas:

- Representación más compleja
- Preguntas como si existe una arista de un vértice a otro tienen una eficiencia O(v)

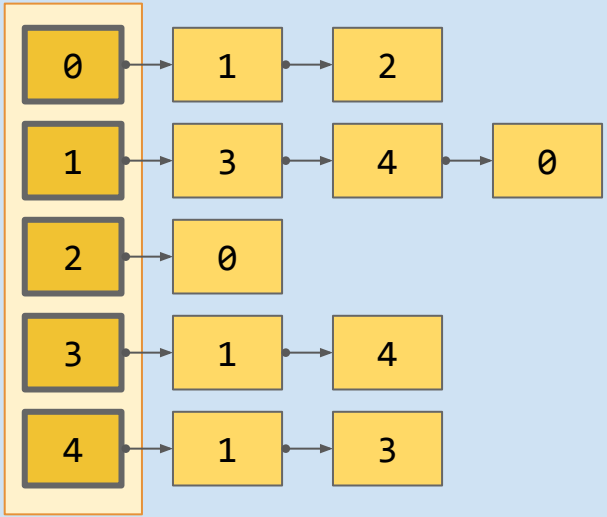
Ejercicio 1.05.

Representá el siguiente grafo mediante listas de adyacencia



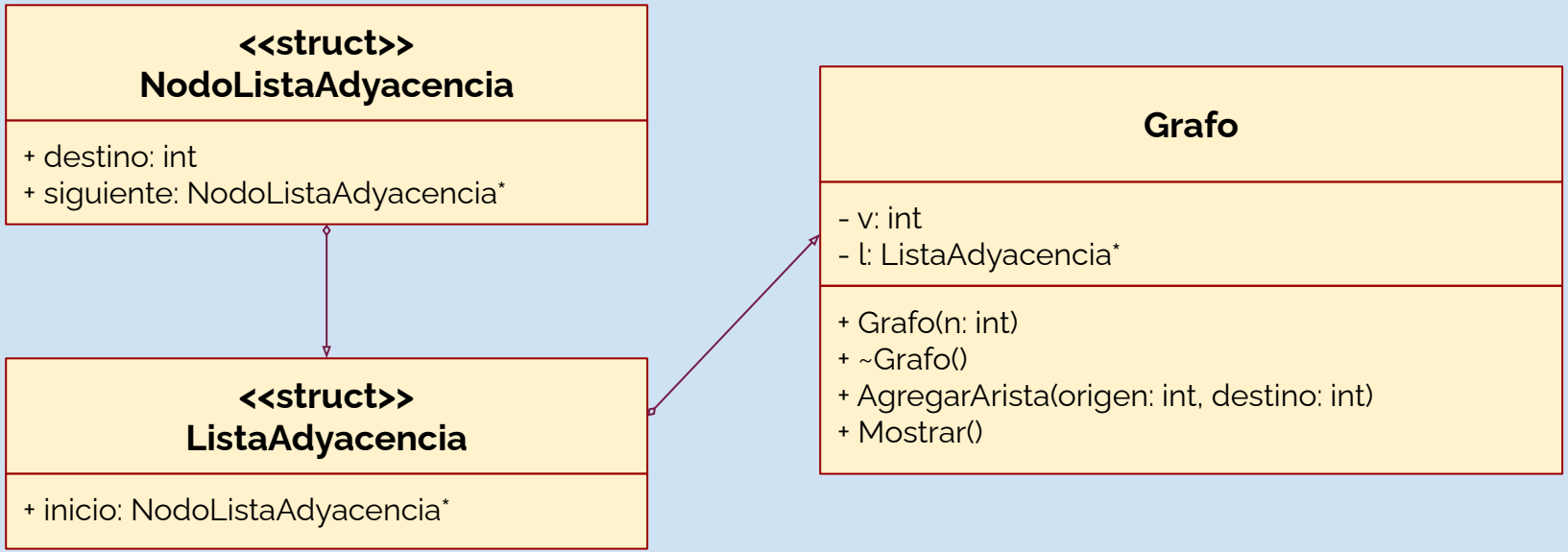
Ejercicio 1.06.

Graficá el grafo cuya representación por lista de adyacencia es:



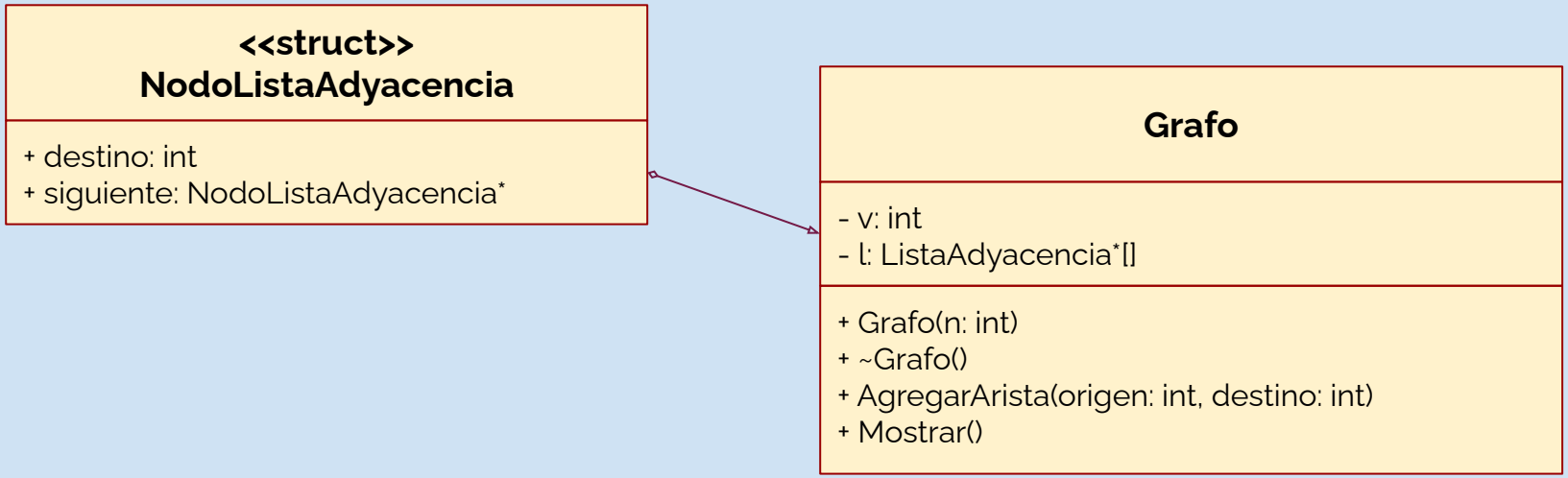
Ejercicio 1.07.

Creá la clase Grafo que implemente un grafo mediante lista de adyacencia



Ejercicio 1.07.

Creá la clase Grafo que implemente un grafo mediante lista de adyacencia



Grafo: lista de adyacencia

```
struct NodoListaAdyacencia{
    int destino;
    NodoListaAdyacencia* siguiente;
};

struct ListaAdyacencia{
    NodoListaAdyacencia* inicio;
};

class Grafo {
    private:
       int v;
       ListaAdyacencia* l;
    public:
       Grafo(int);
       ~Grafo();
       void AgregarArista(int,int);
       void Mostrar();
};
```

```
Grafo::Grafo(int v){
      this->v = v;
      l = new ListaAdyacencia[v];
      for (int i = 0; i < v; i++){
            l[i].inicio = NULL;
      }
}
```

Grafo: lista de adyacencia

```
void Grafo::AgregarArista(int origen, int destino){
    NodoListaAdyacencia* nuevoNodo = new NodoListaAdyacencia;
    nuevoNodo->destino = destino;
    nuevoNodo->siguiente = l[origen].inicio;
    l[origen].inicio = nuevoNodo;

    // si es no dirigido
    nuevoNodo = new NodoListaAdyacencia;
    nuevoNodo->destino = origen;
    nuevoNodo->siguiente = l[destino].inicio;
    l[destino].inicio = nuevoNodo;
}
```

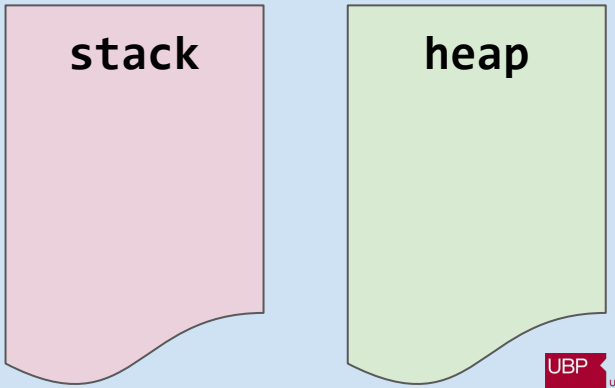
Grafo: lista de adyacencia

```
void Grafo::Mostrar(){
     for (int i = 0; i < v; i++){
          cout << "Lista de adyacencia para el vértice " << i << ":" << endl;
          NodoListaAdyacencia* nodoActual = l[i].inicio;
          while(nodoActual){
               cout << "[" << nodoActual->destino << "]-->";
               nodoActual = nodoActual->siguiente;
          }
          cout << endl;
     }
}
int main(){
     Grafo g(5);
     g.AgregarArista(1,3);
     g.AgregarArista(1,4);
     g.AgregarArista(2,3);
     g.AgregarArista(3,4);
     g.Mostrar();

     return 0;
}
```

Ejercicio 1.08.

Usando lápiz y papel, dibujá un esquema graficando la distribución en la memoria de un grafo representado por una lista de adyacencia.

Cuando estés seguro de comprender como se ha asignado la memoria, implementá el destructor



Grafo: lista de adyacencia

```
Grafo::~Grafo(){
      NodoListaAdyacencia* nodoActual;
      for (int i = 0; i < v; i++){
            nodoActual = l[i].inicio;
            while(nodoActual) {
             NodoListaAdyacencia *nodoABorrar = nodoActual;
             nodoActual = nodoActual->siguiente;
             delete nodoABorrar;
           }
      }

      delete[] l;
}
```