Ejercicios tipo Examen Estructura de Datos No Lineales

Índice general

1. Ejercicios Árboles

En este apartado vas a encontrar todos los ejercicios tipo examen sobre árboles binarios, generales, etc.

Árbol Binario Reflejado

Enunciado: *A partir de un árbol binario, creamos otro intercambiando los subárboles del mismo.*

```
#include <iostream>
#include "abin.h"

template <typename T>
Abin<T> AbinReflejado(const Abin<T> &A){
  //Creamos el nuevo árbol binario
  Abin<T> Reflejado;
  if(!A.arbolVacio()){
    Reflejado.insertarRaiz(A.elemento(A.raiz()));
    return AbinReflejado_rec(A.raiz(),Reflejado.raiz(),A,Reflejado);
  }
  return Reflejado; //devolvemos el árbol
}

template <typename T>
void AbinReflejado_rec(typename Abin<T>::nodo a, typename Abin<T>::nodo
↪  b, const Abin<T> &A, Abin<T> &B){
  if(a != Abin<T>::NODO_NULO){
    //Vamos a insertar en el subárbol izquierdo del nodo b de B el
    ↪  subárbol derecho del nodo a de A y viceversa.
    if(A.hijoDrcho(a)!=Abin<T>::NODO_NULO)
      B.insertarHijoIzqdo(b,A.elemento(A.hijoDrcho(a)));
    //Llamamos a la función recursiva con el subárbol derecho de A y
    ↪  izquierdo de B.
    AbinReflejado(A.hijoDrcho(a),A.hijoIzqdo(b),A,B);
    if(A.hijoIzqdo(a) != Abin<T>::NODO_NULO)
      B.insertarHijoDrcho(b,A.elemento(A.hijoIzqdo(a)));
    //Llamamos a la función con el subárbol izquierdo A y derecho de B.
    AbinReflejado(A.hijoIzqdo(a),B.hijoDrcho(b),A,B);
  }
  //Devolvemos por referencia el árbol B (Reflejado),
}
```

Para llevar a cabo este ejercicio hemos hecho uso de un árbol binario (ya que no lo especificaba el enunciado) con la representación enlazada, siendo la parte privada del TAD Abin:

```
  private:
    struct celda{
      T elto;
      nodo padre, hizq, hder;
      celda(const T& e, nodo p = NODO_NULO):elto(e),padre(p),
        hizq(NODO_NULO),hder(NODO_NULO){}
    };
  nodo r; //raíz del árbol
};//fin del TAD
```

Además hemos hecho uso de los métodos públicos del TAD Abin:

- `Abin<T> Abin();`
  - *Post*: Crea y devuelve un Abin vacío.
- `bool arbolVacio()const;`
  - *Post*: Devuelve true si el árbol está vacío, si no false.
- `void insertarRaiz(const T& e);`
  - *Pre*: Árbol vacío
  - *Post*: Inserta el elemento en la raíz del árbol.
- `nodo raiz() const;`
  - *Post*: Devuelve el nodo que es raíz del árbol, si vacío devuelve `NODO_NULO`.
- `const T& elemento(nodo n)const;`
  - *Pre*: El nodo n existe en el árbol.
  - *Post*: Devuelve el elemento del nodo n.
- `nodo hijoIzqdo(nodo n)const;`
  - *Pre*: El nodo n existe en el árbol.
  - *Post*: Devuelve el hijo derecho del nodo n, si no existe devuelve `NODO_NULO`.
- `nodo hijoDrcho(nodo n)const;`
  - *Pre*: El nodo n existe en el árbol.
  - *Post*: Devuelve el hijo derecho del nodo n, si no existe devuelve `NODO_NULO`.
- `void insertarHijoIzqdo(nodo n, const T& e);`
  - *Pre*: El nodo n existe en el árbol y no tiene hijo izquierdo previo.
  - *Post*: Inserta el elemento en el hijo izquierdo de n.

- `void insertarHijoDrcho(nodo n, const T& e);`
  - *Pre*: El nodo n existe en el árbol y no tiene hijo derecho previo.
  - *Post*: Inserta el elemento en el hijo derecho de n.

Árbol General Reflejado

Enunciado: *A partir de un árbol general, creamos otro intercambiando los subárboles del mismo*

```
#include <iostream>
#include "agen.h"

template <typename T>
Agen<T> AgenReflejado(const Agen<T> &A){
  //Creamos el agen a devolver
  Agen<T> Reflejado;
  Reflejado.insertarRaiz(A.elemento(A.raiz()));
  //Si la raiz de A no tiene hijo izquierdo, es un árbol que solo contiene
  ↪  un nodo.
  if(A.hijoIzqdo(A.raiz()) != Agen<T>::NODO_NULO)
    return AgenReflejado_rec(A.raiz(),Reflejado.raiz(),A,Reflejado);
  return Reflejado; //devolvemos el árbol general Reflejado
}

template <typename T>
void AgenReflejado_rec(typename Agen<T>::nodo a, typename Agen<T>::nodo
↪  b, const Agen<T> &A, Agen<T> &B){
  if(a != Agen<T>::NODO_NULO){ //Comprobamos que a no sea un nodo nulo
    //Vamos a ir recorriendo los hijos de a (y sus nietos, ...) e
    ↪  insertando en el árbol B
    typename Agen<T>::nodo hijo = A.hijoIzqdo(a);
    while(hijo != Agen<T>::NODO_NULO){
      //Si insertamos un hijo izquierdo que ya existe, este se
      ↪  convertirá en hermano derecho automaticamente del nuevo hijo
      ↪  izquierdo que se inserta
      B.insertarHijoIzqdo(b,A.elemento(hijo));
      //Ahora acedemos al hijo de hijo
      AgenReflejado(hijo,b,A,B);
      hijo = A.hermDrcho(hijo); //ahora accedemos a los hermanos.
    }
  }
  //Devolvemos el árbol general Reflejado por referencia
}
```

Al igual que en el ejercicio con el árbol binario hemos hecho uso de la representación enlazada pero ahora del árbol general, ya que no tenemos un número de nodos predefinidos, siendo está la parte privada del TAD:

```
  private:
    struct celda{
      T elto;
      nodo padre, hizq, hder;
      celda(const T& e, nodo p = NODO_NULO): padre(p),
        hizq(NODO_NULO),hder(NODO_NULO){}
    };
  nodo r; //raíz del árbol
}; //fin del TAD Agen
```

También hemos hecho uso de los siguientes métodos públicos del TAD Agen:

- `Agen<T> Agen();`
  - *Post*: Crea y devuelve un Agen vacío.
- `void insertarRaiz(const T& e);`
  - *Post*: Inserta el elemento en la raíz del árbol.
- `const T& elemento (nodo n)const;`
  - *Pre*: El nodo n existe en el árbol.
  - *Post*: Devuelve el elemento del nodo n.
- `nodo raiz()const;`
  - *Post*: Devuelve el nodo raiz del árbol, si vacío devuelve `NODO_NULO`.
- `nodo hijoIzqdo(nodo n)const;`
  - *Pre*: El nodo n existe en el árbol.
  - *Post*: Devuelve el hijo izquierdo del árbol, si no existe devuelve `NODO_NULO`.
- `void insertarHijoIzqdo(nodo n, const T& e);`
  - *Pre*: El nodo n existe en el árbol.
  - *Post*: Inserta el elemento en el hijo izquierdo de n, si existe, el anterior se convierte en su hermano derecho.
- `nodo hermDrcho(nodo n)const;`
  - *Pre*: El nodo n existe en el árbol.
  - *Post*: Devuelve el hermano derecho del nodo n, si no existe devuelve `NODO_NULO`.

Árbol General Similar

Enunciado: *Árboles generales similares. Dos árboles generales son similares si tienen la misma estructura y el contenido de sus nodos hojas deben de ser el mismo.*

Para saber si son similares el árbol general tiene que cumplir dos cosas:

1. Tiene que tener la misma estructura de ramificación → el nodo a del árbol A y el nodo b del árbol B tienen el mismo número de hijos.
2. Los nodos hojas de ambos árboles tienen que tener el mismo contenido.

Vamos a craear un programa que nos compruebe ambas cosas.

```
#include <iostream>
#include "agen.h"

template <typename T>
bool Similares(const Agen<T> &A, const Agen<T> &B){
  //Si ambos tienen solamente un nodo (raiz), diremos que son similares
  if(A.raiz() == Agen<T>::NODO_NULO && B.raiz() == Agen<T>::NODO_NULO)
    return true;
  return Similares_rec(A.raiz(),B.raiz(),A,B);
}

//Función auxiliar que nos calcula el número de hijos que tiene un nodo
template <typename T>
size_t nHijos(typename Agen<T>::nodo n, const Agen<T> &A){
  size_t nhijos = 0;
  if (n == Agen<T>::NODO_NULO)
    return nhijos;
  else{
    //Vamos a recorrer toda la lista de hijos del nodo n
    typename Agen<T>::nodo hijo = A.hijoIzqdo(n);
    while(hijo != Agen<T>::NODO_NULO){
      nhijos++;
      hijo = A.hermDrcho(hijo);
    }
    return nhijos;
  }
}
//Función que nos comprueba si el nodo es una hoja
template <typename T>
bool esHoja(typename Agen<T>::nodo n, const Agen<T> &A){
  return A.hijoIzqdo(n) == Agen<T>::NODO_NULO;
}

template <typename T>
```

```
bool Similares_rec(typename Agen<T>::nodo a, typename Agen<T>::nodo b,
↪  const Agen<T> &A; const Agen<T> &B){
  //Ahora tenemos que comprobar las diferentes condiciones
  if(a == Agen<T>::NODO_NULO && b == Agen<T>::NODO_NULO) return true;
  else if(a == Agen<T>::NODO_NULO || Agen<T>::NODO_NULO)
    return false; //si existe a pero no b, o viceversa no son similares.
  else if(esHoja(a,A) && esHoja(b,B)){
    //Comprobamos el contenido de las hojas
    if(A.elemento(a) == B.elemento(b))
      return true; //mismo elemento
    else return false;
  }
  else{ //no son hojas, vamos a comprobar la ramificación
  //nos creamos una flag que nos devolerá si son similares o no
    bool flag = nHijos(a,A) == nHijos(b,B); //tiene los mismos hijos, true
    while(flag){ //mientra que sean similares, comprobamos los hijos de
    ↪  los hijos
      flag &= Similares_rec(A.hijoIzqdo(a),B.hijoIzqdo(b),A,B); //llamamos
      ↪  con sus hijos
      a = A.hermDrcho(a);
      b = B.hermDrcho(b);
    }
    return flag; //devolvemos si son similares o no.
  }
}
```

Como no tenemos un número determinados del árbol general, para la resolución de este ejercicio, hemos hecho uso de la representación enlazada del TAD agen, siendo su parte privada:

```
  private:
    struct celda{
      T elto;
      nodo padre, hizq, hder;
      celda(const T& e, nodo p = NODO_NULO): elto(e), padre(p),
        hizq(NODO_NULO), hder(NODO_NULO){}
    };
  nodo r; //nodo raíz del árbol
};
```

También hemos hecho uso de los métodos públicos del TAD Agen:

- `const T& elemento (nodo n)const;`
  - *Pre*: El nodo n existe en el árbol.
  - *Post*: Devuelve el elemento del nodo n.
- `nodo raiz()const;`
  - *Post*: Devuelve el nodo raiz del árbol, si vacío devuelve `NODO_NULO`.

- `nodo hijoIzqdo(nodo n)const;`
  - *Pre*: El nodo n existe en el árbol.
  - *Post*: Devuelve el hijo izquierdo del árbol, si no existe devuelve `NODO_NULO`.
- `nodo hermDrcho(nodo n)const;`
  - *Pre*: El nodo n existe en el árbol.
  - *Post*: Devuelve el hermano derecho del nodo n, si no existe devuelve `NODO_NULO`.

Tres Nietos

Enunciado: *Un nodo verde es aquel nodo que cumple una cierta codición, por ejemplo en este caso, nuestra condición es aquel que tiene estrictamente 3 nietos.*
Es decir, vamos a devolver el número de nodos que tienen 3 en el árbol.

Este ejercicio la vamos a hacer tanto para árboles binarios como árboles generales.

Tres Nietos Árboles Binarios

```
#include <iostream>
#include "abin.h"

template <typename T>
size_t ContarNodosVerdesAbin(const Abin<T> &A){
  if(A.arbolVacio()) return 0;
  return ContarNodosVerdesAbin_rec(A.raiz(),A);
}

//Para tener 3 nietos tiene que tener si o si dos hijos
template <typename T>
bool DosHijos(typename Abin<T>::nodo n, const Abin<T> &A){
  return (A.hijoIzqdo(n) != Abin<T>::NODO_NULO && A.hijoDrcho(n) !=
   ↪  Abin<T>::NODO_NULO);
}

template <typename T>
size_t TresNietos(typename Abin<T>::nodo n, const Abin<T> &A){
  if(n == Abin<T>::NODO_NULO) return 0;
  else{
    size_t nietos = 0;
    if(DosHijos(n,A)){
      //Vamos a recorrer los nietos de un nodo para ver si tiene 3
       ↪  (descendientes propios de él).
      if(A.hijoIzqdo(A.hijoIzqdo(n)) != Abin<T>::NODO_NULO) nietos++;
       ↪  //tiene hijo izquierdo y nieto izquierdo
      if(A.hijoIzqdo(A.hijoDrcho(n)) != Abin<T>::NODO_NULO) nietos++;
       ↪  //tiene hijo izquierdo y nieto derecho
      if(A.hijoDrcho(A.hijoIzqdo(n)) != Abin<T>::NODO_NULO) nietos++;
       ↪  //tiene hijo derecho y nieto izquierdo
      if(A.hijoDrcho(A.hijoDrcho(n)) != Abin<T>::NODO_NULO) nietos++;
       ↪  //tiene hijo derecho y nieto derecho
    }
    return nietos;
  }
}
```

```
template <typename T>
size_t ContarNodosVerdesAbin_rec(typename Abin<T>::nodo n, const Abin<T>
↪  &A){
  if(n == Abin<T>::NODO_NULO) return 0;
  else if(TresNietos(n,A) == 3) //condición a cumplir
      return 1 + ContarNodosVerdesAbin_rec(A.hijoIzqdo(n),A) +
       ↪  ContarNodosVerdesAbin_rec(A.hijoDrcho(n),A);
  else
    return 0 + ContarNodosVerdesAbin_rec(A.hijoIzqdo(n),A) +
     ↪  ContarNodosVerdesAbin_rec(A.hijoDrcho(n),A);
}
```

Hemos hecho uso de un árbol binario (ya que no lo especificaba el enunciado) con la representación enlazada, siendo la parte privada del TAD Abin:

```
  private:
    struct celda{
      T elto;
      nodo padre, hizq, hder;
      celda(const T& e, nodo p = NODO_NULO):elto(e),padre(p),
        hizq(NODO_NULO),hder(NODO_NULO){}
    };
  nodo r; //raíz del árbol
};//fin del TAD
```

Además hemos hecho uso de los métodos públicos del TAD Abin:

- `bool arbolVacio()const;`
  - *Post*: Devuelve true si el árbol está vacío, si no false.
- `nodo raiz() const;`
  - *Post*: Devuelve el nodo que es raíz del árbol, si vacío devuelve `NODO_NULO`.
- `nodo hijoIzqdo(nodo n)const;`
  - *Pre*: El nodo n existe en el árbol.
  - *Post*: Devuelve el hijo derecho del nodo n, si no existe devuelve `NODO_NULO`.
- `nodo hijoDrcho(nodo n)const;`
  - *Pre*: El nodo n existe en el árbol.
  - *Post*: Devuelve el hijo derecho del nodo n, si no existe devuelve `NODO_NULO`.

Tres Nietos Árboles General

```cpp
#include <iostream>
#include "agen.h"

template <typename T>
size_t ContarNodosVerdesAgen(const Agen<T> &A){
  if(A.raiz() == Agen<T>::NODO_NULO) return 0;
  return ContarNodosVerdesAgen_rec(A.raiz(),A);
}

//Ahora para que un nodo pueda tener nietos, como mínimo tiene que tener
↪  un hijo izquierdo.

//Función para calcular el número de hijos que tiene un nodo, tenemos que
↪  realizar una búsqueda en anchura
template <typename T>
size_t TresNietosAgen(typename Agen<T>::nodo n, const Agen<T> &A){
  if(n == Agen<T>::NODO_NULO) return 0;
  else{
    size_t nietos = 0;
    typename Agen<T>::nodo hijo = A.hijoIzqdo(n);
    while(hijo != Agen<T>::NODO_NULO){
      //Vamos a recorrer los nietos en anchura
      typename Agen<T>::nodo nieto = A.hijoIzqdo(hijo);
      while(nieto != Agen<T>::NODO_NULO){
        nietos++;
        nieto = A.hermDrcho(nieto);
      }
      hijo = A.hermDrcho(hijo);
    }
    return nietos;
  }
}

template <typename T>
size_t ContarNodosVerdesAgen_rec(typename Agen<T>::nodo n, const Agen<T>
↪  &A){
  if(n == Agen<T>::NODO_NULO) return 0;
  else{
    if(TresNietosAgen(n,A) == 3)//cumple la codición, se suma.
      return 1+ContarNodosVerdesAgen_rec(A.hijoIzqdo(n),A);
    else
      return 0+ContarNodosVerdesAgen_rec(A.hijoIzqdo(n),A);
  }
}
```

Para la resolución de este ejercicio, hemos hecho uso de la representación enlazada del TAD agen, siendo su parte privada:

```
  private:
    struct celda{
      T elto;
      nodo padre, hizq, hder;
      celda(const T& e, nodo p = NODO_NULO): elto(e), padre(p),
        hizq(NODO_NULO), hder(NODO_NULO){}
    };
  nodo r; //nodo raíz del árbol
};
```

También hemos hecho uso de los métodos públicos del TAD Agen:

- `nodo raiz()const;`
  - *Post*: Devuelve el nodo raiz del árbol, si vacío devuelve `NODO_NULO`.
- `nodo hijoIzqdo(nodo n)const;`
  - *Pre*: El nodo n existe en el árbol.
  - *Post*: Devuelve el hijo izquierdo del árbol, si no existe devuelve `NODO_NULO`.
- `nodo hermDrcho(nodo n)const;`
  - *Pre*: El nodo n existe en el árbol.
  - *Post*: Devuelve el hermano derecho del nodo n, si no existe devuelve `NODO_NULO`.

Flotar Nodos

Enunciado: *Dado un nodo cualquiera flotarlo hasta que se cumpla una condición. Por ejemplo, flotar un nodo hasta que el elemento del mismo sea menor que el de su padre.*

Vamos a implementar un ABB mediante un Abin, para ello vamos a ir recorriendo el árbol binario comparando los elementos de los nodos y ordenándolos.

Como vamos a ir modificando el árbol, este será una referencia no constante y se devolerá por referencia.

```
#include <iostream
#include "abin.h"

template <typename T>
void FlotarNodos(Abin<T> &A){
  if(!A.arbolVacio()){
    //llamamos al método flotar
    FlotarNodos_rec(A.raiz(),A);
  }
  //Si está vacío no se hace nada
}

template <typename T>
void FlotarNodos_rec(typename Abin<T>::nodo n, Abin<T> &A){
  //Tenemos que comprobar que ni él ni el padre sean nulos.
  if(n != Abin<T>::NODO_NULO || A.padre(n) != Abin<T>::NODO_NULO){
    if(A.elemento(n) > A.elemento(A.padre(n))){
      //Si el elemento de n es mayor, flotamos.
      T elto_aux = A.elemento(n);
      A.elemento(n) = A.elemento(A.padre(n));
      A.padre(n) = elto_aux;
      //llamamos a la recursiva con el padre de n
      FlotarNodos_rec(A.padre(n),A);
    }
    //si el elemento del hijo < padre, termina
  }
  //si alguno es nulo, no hace nada
}
```

Para llevar a cabo este ejercicio hemos hecho uso de un árbol binario (ya que no lo especificaba el enunciado) con la representación enlazada, siendo la parte privada del TAD Abin:

```
  private:
    struct celda{
      T elto;
      nodo padre, hizq, hder;
      celda(const T& e, nodo p = NODO_NULO):elto(e),padre(p),
```

```
        hizq(NODO_NULO),hder(NODO_NULO){}
    };
  nodo r; //raíz del árbol
};//fin del TAD
```

Además hemos hecho uso de los métodos públicos del TAD Abin:

- `bool arbolVacio()const;`
  - *Post*: Devuelve true si el árbol está vacío, si no false.
- `nodo raiz() const;`
  - *Post*: Devuelve el nodo que es raíz del árbol, si vacío devuelve `NODO_NULO`.
- `const T& elemento(nodo n)const;`
  - *Pre*: El nodo n existe en el árbol.
  - *Post*: Devuelve el elemento del nodo n.
- `nodo padre(nodo n)const;`
  - *Pre*: El nodo n existe en el árbol.
  - *Post*: Devuelve el padre del nodo n, si no existe devuelve `NODO_NULO`.

2. Ejercicios Grafos

En este apartado vas a encontrar todos los ejercicios tipo examen sobre Grafos, donde haremos uso de los diversos algoritmos vistos en las clases de teoría y seminarios.

Ciudades Rebeldes

Enunciado: *Se necesita hacer un estudio de las distancias mínimas necesarias para viajar entre dos ciudades cualesquiera de un país llamado Zuelandia.*

El problema es sencillo pero hay que tener en cuenta unos pequeños detalles:

a) La orografía de Zuelandia es un poco especial, las carreteras son muy estrechas y por tanto solo permiten un sentido de la circulación.

b) Actualmente Zuelandia es un país en guerra. Y de hecho hay una serie de ciudades del país que han sido tomadas por los rebeldes, por lo que no pueden ser usadas para viajar.

c) Los rebeldes no sólo se han apoderado de ciertas ciudades del país, sino que también han cortado ciertas carreteras, (por lo que estas carreteras no pueden ser usadas).

d) Pero el gobierno no puede permanecer impasible ante la situación y ha exigido que absolutamente todos los viajes que se hagan por el país pasen por la capital del mismo, donde se harán los controles de seguridad pertinentes.

Dadas estas cuatro condiciones, se pide implementar un subprograma que dados el grafo (matriz de costes) de Zuelandia en situación normal, la relación de las ciudades tomadas por los rebeldes, la relación de las carreteras cortadas por los rebeldes y la capital de Zuelandia, calcule la matriz de costes mínimos para viajar entre cualesquiera dos ciudades zuelandesas en esta situación.

Partimos de la base de que tenemos un grafo ponderado y dirigido (ya que solo se puede ir en una dirección), contamos con el conjunto de ciudades y carreteras (unión de dos ciudades) que están tomadas por los rebeldes, a las cuales no podremos acceder. Además tenemos la capital de Zuelandia, que nos servirá para hacer que todos los viajes se realicen por la misma.

Como queremos obtener las distancias mínimas de viajar entre dos ciudades cualesquiera de Zuelandia, haremos uso del algoritmo de Floyd, ya que este calcula los costes de viajar entre cada par de vértices de todo el grafo.

```
//Definimos los tipos de datos a usar, representaremos una ciudad como un
↪  entero sin signo.
typedef std::pair<size_t,size_t> Carretera;
typedef matriz<size_t>CostesViajes;

CostesViajes ZuelandiaRebeldes(GrafoP<size_t> &G, const vector<size_t>
↪  &CiudadesRebeldes, const vector<Carretera> &CarreterasRebeldes,
↪  size_t Capital){
  //Como tenemos ciudades que no pueden ser accesibles, vamos a modificar
  ↪  el grafo, haciendo que el coste de estas sea infinito.
  for(auto &ciudadrebelde : CiudadesRebeldes)
    for(size_t i = 0; i < G.numVert(); i++)
      G[i][ciudadrebelde] = GrafoP<size_t>::INFINITO;

  //Ahora vamos a hacer inaccesibles dichas carreteras
  for(auto & carreterarebelde : CarreterasRebeldes)
    G[carreterarebelde.first][carreterarebelde.second] =
    ↪  GrafoP<size_t>::INFINITO;

  //Ahora vamos a obligar que todos los viajes se lleven a cabo por la
  ↪  Capital
  for(size_t i = 0; i < G.numVert(); i++)
    if(i != Capital){
      for(size_t j = i+1; j <G.numVert(); j++){
        if(j!= Capital)
        G[i][j] = GrafoP<size_t>::INFINITO;
      }
    }
  //Ya teniendo tanto las ciuades rebeldes como la carreteras rebeldes
  ↪  indicadas en el grafo, y todos los viajes cruzan la capital, vamos a
  ↪  poder calcular los costes mínimos de los caminos entre cada par de
  ↪  vértices del grafo,es decir floyd, para ello, creamos las matrices
  ↪  de costes mínimos y vértices.
  matriz<size_t>Vertices(G.numVert()),CosteMinimos(G.numVert());
  return CosteMinimos = Floyd(G,Vertices);
}
```

Hemos hecho uso de las operaciones de los grafos y prototipos de las funciones:

- `size_t numVert() const;`
  - *Post*: Devuelve el número de vértices del grafo.
- `matriz<size_t>(G.numVert());`
  - *Post*: Crea y devuelve una matriz de enteros sin signos vacía de tamaño G.numVert().
- `vector<tCoste>& operator[](vertice v)const;`
  - *Post*: Devuelve un vector con los costes de las aristas adyacentes a v.

- `const T& GrafoP<tCoste>::INFINITO = std::numeric_limits<T>::max();`
  - *Post*: Devuelve el valor máximo permitido.
- `matriz<tCoste> Floyd(const GrafoP<size_t> &G, vector<typename GrafoP<tCoste>::vertice> &P);`
  - *Pre*: Recibe un Grafo ponderado y una matriz de vértices.
  - *Post*: Devuelve una matriz con los costes mínimos y una matriz con los vértices por los que pasan los caminos.