Práctica 10. TAD grafo no dirigido

Objetivos.

Se trata de construir el TAD GRAFO no dirigido, sin peso, con una implementación acotada. Además, se implementará una biblioteca que incluya recorridos en profundidad y anchura de un grafo.

Enunciado.

Construir el TAD GRAFO según el siguiente módulo de definición:

```
DEFINITION MODULE Grafo;

        FROM TadNodo IMPORT NODO;

        TYPE    GRAFO;
                TIPO_ERROR = <<a definir>>;

        PROCEDURE Error(): TIPO_ERROR;

        PROCEDURE Crear(): GRAFO;

        PROCEDURE EsVacio(g: GRAFO): BOOLEAN;
        PROCEDURE Pertenece(g: GRAFO; n: NODO): BOOLEAN;
        PROCEDURE NumNodos(g: GRAFO): CARDINAL;
        PROCEDURE NumArcos(g: GRAFO): CARDINAL;
        PROCEDURE Adyacentes(g: GRAFO; n1, n2: NODO): BOOLEAN;

        (* Pre: NOT Pertenece(g, n) *)
        PROCEDURE AnyadirNodo (VAR g: GRAFO; n: NODO);
        (* Pre: Pertenece(g, n) *)
        PROCEDURE BorrarNodo(VAR g: GRAFO; n: NODO);

        (* Pre: Pertenece(g, n1) AND Pertenece(g, n2) AND NOT Adyacentes(g, n1, n2) *)
        PROCEDURE Conectar (VAR g: GRAFO; n1,n2: NODO)
        (* Pre: Adyacentes(g, n1, n2) *)
        PROCEDURE Desconectar(VAR g: GRAFO; n1,n2: NODO);

        PROCEDURE Inicializar(g: GRAFO);
        PROCEDURE Elemento(g: GRAFO): NODO;

        PROCEDURE Destruir(VAR g: GRAFO);

        PROCEDURE Copiar (VAR g1: GRAFO; g2: GRAFO);

END Grafo.
```

El TAD GRAFO se representará mediante la siguiente estructura de datos:

```
TYPE    GRAFO = POINTER TO ESTRUCTURA;
        NODOS = SET OF NODO;
        ESTRUCTURA = RECORD
                num_nodos: CARDINAL;
                nodos: NODOS;
                num_arcos: CARDINAL;
                arcos: ARRAY NODO, NODO OF BOOLEAN;
                iter: [ORD(MIN(NODO))..ORD(MAX(NODO)) + 1];
        END;
```

donde nodos es el conjunto de nodos del grafo, arcos es una matriz de adyacencia que representa las conexiones del grafo, num_nodos y num_arcos son contadores de nodos y arcos respectivamente, e iter es un contador para implementar un iterador activo sobre los nodos del grafo. Dado que el grafo es no dirigido, si un nodo n_1 está conectado con otro nodo n_2, entonces n_2 también estará conectado con n_1, por lo que la matriz de adyacencia es simétrica.

Implementar una biblioteca para el TAD GRAFO con las siguientes operaciones:

```
DEFINITION MODULE BibGrafo;

FROM TadNodo IMPORT NODO;
FROM Grafo IMPORT GRAFO;

TYPE TIPO_OPERACION= PROCEDURE(NODO);

PROCEDURE EnProfundidad(g: GRAFO; n: NODO; proc: TIPO_OPERACION);
PROCEDURE EnAnchura(g: GRAFO; n: NODO; proc: TIPO_OPERACION);

END BibGrafo.
```

Estos procedimientos genéricos deben visitar *todos* los nodos conectados con un nodo inicial dado n que se pasa como parámetro, aplicando a cada nodo visitado el procedimiento proc, siguiendo una estrategia de recorrido en profundidad o en anchura, respectivamente.

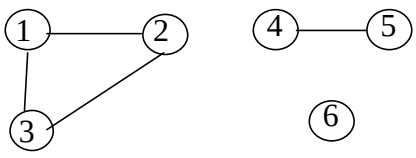
Dado un nodo inicial n, el recorrido en *profundidad* visita primero uno de sus descendientes k, y visita después todos los descendientes de k antes de visitar cualquier otro descendiente del nodo original n. Se trata, pues, de una generalización del recorrido en profundidad de los árboles.

Dado un nodo inicial n, el recorrido en *anchura* visita primero todos sus descendientes directos, y visita después todos los descendientes directos de éstos y así sucesivamente; es decir, visita el grafo por generaciones: hijos, nietos, biznietos, etc. Se trata, pues, de una generalización del recorrido en anchura de los árboles.

Al implementar ambos recorridos se debe poner especial cuidado en no entrar en *ciclos*. Para ello, será necesario mantener una estructura de datos auxiliar que nos indique si un nodo ha sido visitado o no. Dado que el tipo NODO es discreto, se puede emplear para tal finalidad un conjunto de tipo NODO (SET OF NODO).

Práctica Suplementaria

Implementar un algoritmo que calcule las componentes conexas de un grafo. Por ejemplo, el grafo:



tiene tres componentes conexas y cada una de ellas es un grafo.

Hay que implementar una función que cada vez que se ejecute nos devuelva una componente conexa:

```
PROCEDURE CompConexa(g:GRAFO):GRAFO
```

También tendremos que usar otro procedimiento para iniciar el proceso de búsqueda de las componentes conexas:

```
PROCEDURE IniciarConexa(g:GRAFO);
```

Cuando se hayan obtenido todas las componentes conexas la función CompConexa devolverá un grafo vacío. Es decir, como el grafo de la figura anterior tiene 3 componentes conexas, la cuarta vez que ejecutemos la función CompConexa nos devolverá un grafo vacío para indicarnos que ya no hay más componentes.

Para implementar este algoritmo, será necesario disponer de alguna estructura donde se almacenarán los nodos incluidos en las componentes conexas generadas hasta el momento. Esta estructura será inicializada en IniciarConexa, y actualizada a partir del subgrafo devuelto por cada llamada a CompConexa.