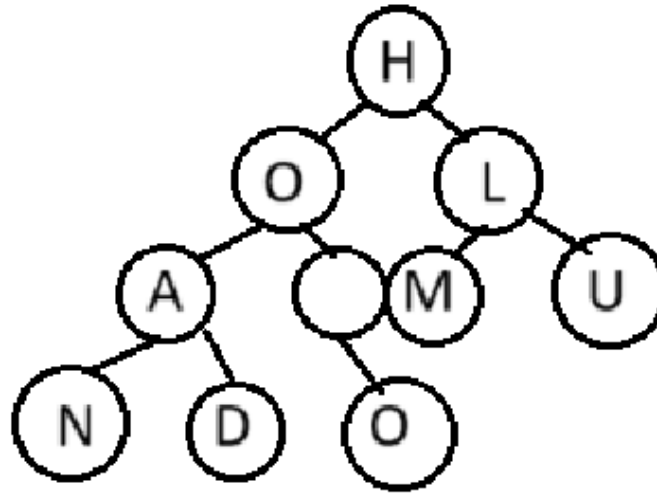
EDNL – EXAMEN FEBRERO 2025 RESUELTO

AVISO: Puede que las soluciones aquí propuestas no sean 100% correctas ni las más eficientes. Se trata de proporcionar una solución a modo de guía a lo que se pedían en los ejercicios del examen. No me hago responsable de los posibles suspensos por copiar justo lo siguiente si volviera a caer.

NOTA 1: Como siempre, es absolutamente necesario escribir en el examen todos los TADs conocidos (parte privada) así como los métodos públicos de ellas y los algoritmos de grafos vistos en clases (cabecera).

NOTA 2: En este examen está expresamente prohibido utilizar contenedores de la STL en ambos ejercicios. 🤡

1. Dada una cadena de texto, se pretende realizar un algoritmo criptográfico que cifre la cadena utilizando árboles binarios. El método de cifrado será disponer los caracteres en anchura y posteriormente leerlos en preorden para obtener su cifrado.
   Es decir, si tenemos la cadena “HOLA MUNDO”



   El resultado debería ser: HOAND OLMU

   Recordamos que no se pueden usar vectores, pilas, colas, etc... de la STL.

```cpp
#include "abin.h"
#include "abin_E-S.h"
#include <iostream>

using namespace std;

template <typename T = char>
void volcarTextoAlArbol2(Abin<T>& A, const T* texto)
{
    A.insertarRaiz(texto[0]); // Primer elemento

    typename Abin<T>::nodo actual = A.raiz();

    // Inicializamos un arreglo dinámico para simular la cola
    typename Abin<T>::nodo* cola = new typename Abin<T>::nodo[100]; // Suponemos un tamaño máximo de 100 elementos
    size_t inicio = 0, fin = 0;

    cola[fin++] = actual;

    size_t i = 1; // Segundo elemento
    while (texto[i] != '\0')
    {
        actual = cola[inicio]; // Obtener el primer elemento de la "cola"
        inicio++;

        if (texto[i] != '\0')
        {
            A.insertarHijoIzqdo(actual, texto[i++]);
            cola[fin++] = A.hijoIzqdo(actual); // Añadir al final de la "cola"
        }
        if (texto[i] != '\0')
        {
            A.insertarHijoDrcho(actual, texto[i++]);
            cola[fin++] = A.hijoDrcho(actual); // Añadir al final de la "cola"
        }
    }

    preorden(A.raiz(), A);

    delete[] cola;
}

template <typename T = char>
void preordenAbin(typename Abin<T>::nodo n, const Abin<T>& A)
{
    if (n != Abin<T>::NODO_NULO)
    {
        cout << A.elemento(n);
        preordenAbin(A.hijoIzqdo(n), A);
        preordenAbin(A.hijoDrcho(n), A);
    }
}
```

2. El archipiélago de Grecoland (Zuelandia) está formado únicamente por dos islas, Fobos y Deimos, que tienen N1 y N2 ciudades, respectivamente, de las cuales C1 y C2 ciudades son costeras (obviamente C1 ≤ N1 y C2 ≤ N2). Se desea construir un puente que una ambas islas. Nuestro problema es elegir el puente a construir entre todos los posibles, sabiendo que el coste de construcción del puente se considera irrelevante. Por tanto, escogeremos aquel **puente que minimice el coste global de viajar entre todas las ciudades de las dos islas**, teniendo en cuenta las siguientes premisas:
    1. Se asume que el coste viajar entre las dos ciudades que una el puente **coincide con la distancia euclídea de ambas.**
    2. Para poder plantearse las mejoras en el transporte que implica la construcción de un puente frente a cualquier otro, se asume que se realizarán exactamente el mismo número de viajes entre cualesquiera ciudades del archipiélago. Por ejemplo, se considerará que el número de viajes entre la ciudad P de Fobos y la Q de Deimos será el mismo que entre las ciudades R y S de la misma isla. Dicho de otra forma, todos los posibles trayectos a realizar dentro del archipiélago son igual de importantes.

       Dadas:
       - **Matriz de adyacencia de Fobos (carreteras).**
       - **Matriz de adyacencia de Deimos (carreteras).**
       - **Coordenadas cartesianas de las ciudades de Fobos.**
       - **Coordenadas cartesianas de las ciudades de Deimos.**
       - **Relación de las ciudades costeras de Fobos.**
       - **Relación de las ciudades costeras de Deimos.**

       Implementa un subprograma que calcule las dos ciudades que unirá el puente.

```cpp
// Es el EJ 12 de P7 pero sin usar la STL y con coste de viajar por el puente
// Nos dan:
// Matriz de adyacencia de las ciudades de Fobos y Deimos (existencia de carreteras)
// Coordenadas cartesianas de las ciudades de Fobos y Deimos
// Relación de las ciudades costeras de Fobos y Deimos
// Implementa un subprograma que calcule las dos ciudades que unirá el puente

#include "../Material para las prácticas de grafos/alg_grafoPMC.h"
#include "../Material para las prácticas de grafos/alg_grafo_E-S.h"
#include "../Material para las prácticas de grafos/grafoPMC.h"
#include <algorithm>
#include <iostream>
#include <cmath>

using namespace std;

typedef struct{
    double coordX, coordY;
}ciudad;

template <typename tCoste>
typedef struct{
    typename GrafoP<tCoste>::vertice ciudad1, ciudad2;
}solucion_puente;

template <typename tCoste>
solucion_puente grecolandNoSTL(const GrafoP<tCoste>& Fobos, const GrafoP<tCoste>& Deimos,
                               const ciudad* ciudadesFobos, const ciudad* ciudadesDeimos,
                               const ciudad* costerasFobos, const ciudad* costerasDeimos)
{
    // Primero calculamos los costes mínimos entre todas las ciudades usando Floyd
    typedef typename GrafoP<tCoste>::vertice vert;
    GrafoP<tCoste> super(Fobos.numVert() + Deimos.numVert()); // Supergrafo para simular el viaje con el puente

    for(size_t i = 0; i < Fobos.numVert(); i++)
    {
        for(size_t j = 0; j < Fobos.numVert(); j++)
        {
            if(Fobos[i][j])
                super[i][j] = sqrt(pow(ciudadesFobos[i].coordX - ciudadesFobos[j].coordX, 2) + pow(ciudadesFobos[i].coordY - ciudadesFobos[j].coordY, 2));
        }
    }

    for(size_t i = 0; i < Deimos.numVert(); i++)
    {
        for(size_t j = 0; j < Deimos.numVert(); j++)
        {
            if(Deimos[i][j])
                super[i + Fobos.numVert()][j + Fobos.numVert()] = sqrt(pow(ciudadesDeimos[i].coordX - ciudadesDeimos[j].coordX, 2) + pow(ciudadesDeimos[i].coordY - ciudadesDeimos[j].coordY, 2));
        }
    }

    size_t puenteMinimo = GrafoP<tCoste>::INFINITO;
    size_t puenteAct = sqrt(pow(costerasFobos[0].coordX - costerasDeimos[0].coordX, 2) + pow(costerasFobos[0].coordY - costerasDeimos[0].coordY, 2));

    matriz<vert> mVert(super.numVert());

    solucion_puente<tCoste> solu;

    // Comparamos los puentes entre las ciudades costeras de Fobos y Deimos
    for(size_t i = 0; i < costerasFobos.size(); i++)
    {
        for(size_t j = 0; j < costerasDeimos.size(); j++)
        {
            puenteAct = sqrt(pow(costerasFobos[i].coordX - costerasDeimos[j].coordX, 2) + pow(costerasFobos[i].coordY - costerasDeimos[j].coordY, 2));
            if(puenteAct < puenteMinimo)
            {
                puenteMinimo = puenteAct;
                super[i][j] = puenteMinimo; // Contamos el puente para hacer Floyd (SU COSTE NO ES 0)

                matriz<tCoste> mCostes = Floyd(super, mVert); // Simulamos el viaje

                // Si minimiza el coste global de viajar con ese puente, guardamos las ciudades costeras
                if(mCostes[i][j] < puenteMinimo)
                {
                    solu.ciudad1 = i; // Ciudad de Fobos
                    solu.ciudad2 = j; // Ciudad de Deimos
                }

                super[i][j] = GrafoP<tCoste>::INFINITO; // Por tanto, luego debemos eliminarlo para no poner todos los puentes por error
            }
        }
    }

    return solu;
}
```