1. Punteros

1) Refugio de Animales: Un refugio de animales necesita un sistema para poder gestionar los ingresos y adopciones de los perros y gatos. Para los perros es que el primero que llega es el primero en ser adoptado (FIFO). Para los gatos el orden es inverso (LIFO). El sistema debe:

→ Poder ingresar perros y/o gatos al sistema:
→ Poder sacar del sistema a los que son adoptados (pop).
→ Cada vez que uno es adoptado, el sistema debe elegir el animal correcto de acuerdo a las políticas del refugio.

```
struct Animal{
   string nombre;
   int edad;
   string lugar;
};

struct NodoAnimal{
   Animal animal;
   NodoAnimal* siguiente;
};

// crear variables y pedir su ingreso
Animal crearAnimal(){
   Animal animal;
   cout << "Ingrese los datos del animal:" << endl;
   cout << "Nombre: ", cin >> animal.nombre;
   cout << "Edad: ", cin >> animal.edad;
   cout << "Lugar: ", cin >> animal.lugar;
return animal;
}

// crear nodo del animal
NodoAnimal* crearNodo(Animal animalIngresado) {
   NodoAnimal* nuevoNodo = new NodoAnimal;
   nuevoNodo->animal = animalIngresado;
   nuevoNodo->siguiente = nullptr;
return nuevoNodo;
}

// FUNCIÓN 1 // Ingresar un elemento en una cola
void ingresarPerro(NodoAnimal*& frente, NodoAnimal*& final){
   NodoAnimal* nuevoNodo = crearNodo(crearAnimal());
   if (frente == nullptr){
      frente = final = nuevoNodo;
   } else {
      final->siguiente = nuevoNodo;
      final = nuevoNodo;
```

```
   }
}
// FUNCIÓN 2 // Ingresar un elemento en una pila
void ingresarGato(NodoAnimal*& pilaGatos){
   NodoAnimal* nuevoNodo = crearNodo(crearAnimal());
   nuevoNodo->siguiente = pilaGatos;
   pilaGatos = nuevoNodo;
}
// FUNCIÓN 3 // Eliminar el primer elemento de una cola
void adoptarPerro(NodoAnimal*& frente, NodoAnimal*& final){
   if (frente == nullptr) {
      cout << "La cola está vacía." << endl;
   }
   else{
      NodoAnimal* temp = frente; // Guardar el nodo a eliminar en una
variable temporal
      frente = frente->siguiente; // Actualizar 'frente' al siguiente nodo
      if (frente == nullptr) {
      final = nullptr; // Si se elimina el último elemento, actualizar
'final'
      }
delete temp;
}
}

// FUNCIÓN 4 // Eliminar el primer elemento de una pila
void adoptarGato(NodoAnimal*& pilaGatos){
   if (pilaGatos == nullptr) {
      cout << "La pila está vacía." << endl;
   } else {
      NodoAnimal* temp = pilaGatos;
      pilaGatos = pilaGatos->siguiente;
   delete temp;
   }
}

// FUNCIÓN 5: Ordenado según orden en el que se adoptan
void mostrarPerros(NodoAnimal* frente){
   int i = 1;
   cout << "Elementos en la cola:" << endl;
   while (frente != nullptr){  // Desencolar y mostrar el elemento del
frente de la cola
   cout << i << ". " << frente->animal.nombre << endl;
   frente = frente->siguiente;
   i++;
   }
}

// FUNCIÓN 6: Ordenado según orden en el que se adoptan
void mostrarGatos(NodoAnimal* pilaGatos){
   int i = 1;
   cout << "Elementos en la pila:" << endl;
```

```
   while (pilaGatos != nullptr){
   cout << i << ". " << pilaGatos->animal.nombre << endl;
   pilaGatos = pilaGatos->siguiente;
   i++;
   }
}
```

2) Banco:

→ **Cola del banco:** Implementa una estructura de datos para representar una cola en un banco. Añade funciones para ingresar un cliente a la cola (encolar), atender al próximo cliente (desencolar) y mostrar al próximo cliente en ser atendido (primero en la cola).

→ **Cola de prioridad:** Implementa una cola de prioridad en la cola del banco. Puedes, por ejemplo, dar prioridad a las personas mayores de 60 años.

→ **Registro de espera:** El banco quiere implementar un sistema para medir la espera de los clientes en el banco para saber cuándo contratar más asistentes. Para lo cual, quieren tener un historial que diga la cantidad de personas que una persona tiene delante cada vez que se suma a una cola.

```
struct Cliente{
   string nombre;
   int edad;
};

struct Nodo{
   Cliente cliente;
   Nodo* siguiente;
};

struct NodoHistorial{
   int cantidad;
   NodoHistorial* siguiente;
};

Cliente crearCliente(){
   Cliente cliente;
   cout << "Ingrese el nombre del cliente: ", getline(cin, cliente.nombre);
   cout << "Ingrese la edad del cliente: ", cin >> cliente.edad;
   cin.ignore();
return cliente;
}

Nodo* crearNodo(Cliente clienteIngresado) {
   Nodo* nuevoNodo = new Nodo;
   nuevoNodo->cliente = clienteIngresado;
   nuevoNodo->siguiente = nullptr;
return nuevoNodo;
}

NodoHistorial* crearNodo(int valor) {
   NodoHistorial* nuevoNodo = new NodoHistorial;
   nuevoNodo->cantidad = valor;
```

```
   nuevoNodo->siguiente = nullptr;
return nuevoNodo;
}

void encolar(Nodo*& colaFrente, Nodo*& colaFinal, Cliente cliente){
   Nodo* nuevoNodo = crearNodo(cliente);
   if (colaFrente == nullptr){
     colaFrente = colaFinal = nuevoNodo;
   } else {
     colaFinal->siguiente = nuevoNodo;
     colaFinal = nuevoNodo;
   } }

string desencolar(Nodo*& colaFrente, Nodo*& colaFinal){
   if (colaFrente == nullptr) {
     return "";
   } else {
     string clienteAtendido = colaFrente->cliente.nombre;
     Nodo* temp = colaFrente; // Guardar el nodo a eliminar en una variable
temporal
     colaFrente = colaFrente->siguiente; // Actualizar 'colaFrente' al
siguiente nodo
     if (colaFrente == nullptr) {
     colaFinal = nullptr; // Si se elimina el último elemento, actualizar
'colaFinal'
     }
   delete temp;
return clienteAtendido;
} }

void frente(Nodo* colaFrente){
   if(colaFrente == nullptr){
     cout << "No hay clientes en espera." << endl;
   } else{
     cout << "Próximo cliente: " << colaFrente->cliente.nombre << endl;
   } }
```

```
// FUNCIÓN 7: Contar elementos dentro de una cola

int espera(Nodo* colaFrente, Nodo* prioridadFrente, bool prioridad){
   int cantidad = 0;

   // Cantidad de personas en la cola de prioridad
   while (prioridadFrente != nullptr){
     prioridadFrente = prioridadFrente->siguiente;
     cantidad++;
   }

   // Cantidad de personas en la cola de espera
   if (!prioridad){
     while (colaFrente != nullptr){
       colaFrente = colaFrente->siguiente;
       cantidad++;
```

```
   } }
return cantidad - 1;
}
```

```
// FUNCIÓN 8: Enlistar elementos
```

```
void cargarLista(NodoHistorial*& inicio, int valor) {
   NodoHistorial* nuevoNodo = crearNodo(valor);
   if (inicio == nullptr){
     inicio = nuevoNodo;
   } else {
     NodoHistorial* actual = inicio;
     while (actual->siguiente != nullptr) {
        actual = actual->siguiente;
   }
   actual->siguiente = nuevoNodo;
   }
}
```

3) Tickets: El sistema debe poder cargar una lista de espectáculos donde cada uno tiene: nombre del artista, cantidad de público máximo, cantidad de público que compró el ticket, cola de espera para comprar.

→ Sumar nuevas personas a la cola de espera correspondiente.

→ Vender los tickets de un espectáculo hasta que la cola quede vacía o se quede sin tickets

```
struct NodoCola{
   int dni;
   NodoCola* siguiente;
};

struct Espectáculo{
   string nombre;
   int tickets;
   int ventas = 0;
   NodoCola* colaFrente = nullptr;
   NodoCola* colaFinal = nullptr;
};

struct NodoLista{
   Espectaculo espectaculo;
   NodoLista* siguiente;
};

Espectáculo ingresarEspectaculo(){
   Espectaculo espectaculo;
   cout << "Ingrese el nombre del artista: ", cin >> espectaculo.nombre;
   cin.ignore();
   cout << "Ingrese la cantidad del publico maximo: ", cin >>
espectaculo.tickets; cin.ignore();
   return espectaculo;
```

```
}

NodoLista* crearNodo(Espectaculo espectaculoIngresado) {
   NodoLista* nuevoNodo = new NodoLista;
   nuevoNodo->espectáculo = espectaculoIngresado;
   nuevoNodo->siguiente = nullptr;
   return nuevoNodo;
}

// FUNCIÓN 8: Enlistar elementos

void enlistar(NodoLista*& lista, Espectaculo espectaculo){
   NodoLista* nuevoNodo = crearNodo(espectáculo);
   if (lista == nullptr){
     lista = nuevoNodo;
   } else {
     NodoLista* actual = lista;
     while (actual->siguiente != nullptr) {
     actual = actual->siguiente;
   }
   actual->siguiente = nuevoNodo;
   } }

// FUNCIÓN 9: Mostrar elementos de una lista sin perderlos, elegir uno en
específico
Espectaculo buscarEspectaculo(NodoLista* lista){
   cout << "Elija un espectáculo entre los disponibles: " << endl;
   int i = 1; // no se utiliza un for porque se desconoce la cant total
   NodoLista* aux = lista;
   while (aux != nullptr) {
     cout << i << ". " << aux->espectaculo.nombre << endl; aux = aux-
>siguiente;
     i++;
}

   int espectaculoIngresado;
   Espectaculo espectaculoBuscado;
   bool valido;

   do{
     cin >> espectaculoIngresado;
     cin.ignore();

   if (espectaculoIngresado > 0 && espectaculoIngresado <= i){

     int j = 1;
     aux = lista;
     while (aux != nullptr) {
        if(j == espectaculoIngresado){
        espectaculoBuscado = aux->espectaculo;
        valido = true;
        break;
        }
        aux = aux->siguiente;
```

```
        j++;
      }
    } else{
      cout << "Opcion invalida. Ingrese nuevamente." << endl;
      valido = false;
    }
    } while(!valido);
return espectaculoBuscado;
}

NodoCola* crearNodo(int dniIngresado) {
    NodoCola* nuevoNodo = new NodoCola;
    nuevoNodo->dni = dniIngresado;
    nuevoNodo->siguiente = nullptr;
return nuevoNodo;
}

void encolar(NodoCola*& colaFrente, NodoCola*& colaFinal, int dni){
NodoCola* nuevoNodo = crearNodo(dni);
    if (colaFrente == nullptr){
      colaFrente = colaFinal = nuevoNodo;
    } else {
      colaFinal->siguiente = nuevoNodo;
      colaFinal = nuevoNodo;
    }
}

// FUNCIÓN 9: Eliminar un nodo específico en una lista

NodoLista* eliminarNodo(NodoLista*& inicio, Espectaculo espectaculo){

    if (inicio == nullptr){
      return nullptr; // Lista vacía, no hay nada que eliminar
    }

    // si el elemento a eliminar es el inicial
    if (inicio->espectaculo.nombre == espectaculo.nombre){
      NodoLista* temp = inicio;
      inicio = inicio->siguiente;
      delete temp;
      return inicio; // Devuelve la lista actualizada
    }

    NodoLista* actual = inicio;

    // si el elemento a eliminar no es el inicial
    while (actual->siguiente != nullptr){
      if (actual->siguiente->espectaculo.nombre == espectaculo.nombre){
        NodoLista* temp = actual->siguiente;
        actual->siguiente = actual->siguiente->siguiente;
        delete temp;
        return inicio; // Devuelve la lista actualizada
```

```
     }
     actual = actual->siguiente;
   }
return inicio; // Valor no encontrado en la lista
}


int desencolar(NodoCola*& colaFrente, NodoCola*& colaFinal){
   if (colaFrente == nullptr) {
     return -1;
   }
   else{
     int dni = colaFrente->dni;
     NodoCola* temp = colaFrente;
     colaFrente = colaFrente->siguiente;
     if (colaFrente == nullptr) {
     colaFinal = nullptr;
   }
   delete temp;

return dni;
} }
```

4) Spotify:

```
struct Cancion{
    string nombre;
    int puntuacion;
};

struct NodoCancion{
    Cancion cancion;
    NodoCancion* siguiente;
};

Cancion crearCancion(){
    Cancion cancion;
    cout << "Ingrese el nombre de la cancion: ", getline(cin,
cancion.nombre);
    return cancion;
}

NodoCancion* crearNodo() {
    Cancion cancionIngresada = crearCancion();
    NodoCancion* nuevoNodo = new NodoCancion;
    nuevoNodo->cancion = cancionIngresada;
    nuevoNodo->siguiente = nullptr;
    return nuevoNodo;
```

```
}
```

```
// FUNCIÓN 10: Ingresar un elemento en una pila
void push(NodoCancion*& pila){
    NodoCancion* nuevoNodo = crearNodo();
    nuevoNodo->siguiente = pila;
    pila = nuevoNodo;
}
```

```
// FUNCIÓN 11: Quitar un elemento en una pila
string pop(NodoCancion*& pila, NodoCancion*& historial){
    if (pila == nullptr) {
        cout << "La pila esta vacía." << endl;
        return "";
    } else {
        string cancionEscuchada = pila->cancion.nombre;
        NodoCancion* temp = pila;
        pila = pila->siguiente;
        delete temp;
        return cancionEscuchada;
    }
}
```

```
// FUNCIÓN 12: Mostrar un dato del próximo elemento de la pila
void top(NodoCancion* pila){
    if(pila == nullptr){
        cout << "No se encuentra cancion en la pila." << endl;
    } else{
        cout << "Nombre de la proxima cancion: " << pila->cancion.nombre
<< endl;
    }
}
```

```
// FUNCIÓN 13: Revertir orden de una pila
// siguiente <-- 1 <-- 2 <-- 3
void revertirPlaylist(NodoCancion*& pila){
    if (pila == nullptr) {
        cout << "La pila esta vacía." << endl;
    } else{
        NodoCancion* aux = pila; // Creo un aux igual a pila
        pila = nullptr; // Vació la pila
        while (aux != nullptr){
            NodoCancion* temp = aux;
            aux = aux->siguiente;
            temp->siguiente = pila;
```

```
            pila = temp;
        }
    }
}

void pushHistorial(NodoCancion*& historial, string cancionEscuchada){
    NodoCancion* nuevoNodo = new NodoCancion;
    nuevoNodo->cancion.nombre = cancionEscuchada;
    cout << "Ingrese la puntuacion del 1 al 5: ", cin >> nuevoNodo-
>cancion.puntuacion;
    cin.ignore();
    nuevoNodo->siguiente = historial;
    historial = nuevoNodo;
}

string popHistorial(NodoCancion*& historial){
    if (historial == nullptr) {
        cout << "No se ha escuchado ninguna cancion recientemente." <<
endl;
        return "";
    } else{
        string cancionEscuchada = historial->cancion.nombre;
        NodoCancion* temp = historial;
        historial = historial->siguiente;
        delete temp;

        return cancionEscuchada;
    }
}

void listarHistorial(NodoCancion* historial){
    int filtrado;
    cout << "Desea hacerlo con filtrado?" << endl;
    cout << "1. Si." << endl;
    cout << "2. No." << endl;
    cin >> filtrado;
    cin.ignore();

    if(filtrado == 1){
        if (historial == nullptr) {
            cout << "El historial esta vacío." << endl;
        } else{
            int puntaje;
            cout << "Ingrese el puntaje a filtrar: ", cin >> puntaje;
```

```
            cin.ignore();
            NodoCancion* aux = historial; // Creo un aux igual a historial
            historial = nullptr; // Vació el historial
            while (aux != nullptr){ // Invertir el orden del historial
                NodoCancion* temp = aux;
                aux = aux->siguiente;
                temp->siguiente = historial;
                historial = temp;
            }
            int i = 1;
            while (historial != nullptr){
                string cancionEscuchada = historial->cancion.nombre;
                if (historial->cancion.puntuacion == puntaje){
                    cout << i << ". " << cancionEscuchada << endl;
                    i++;
                }
                NodoCancion* temp = historial;
                historial = historial->siguiente;
                delete temp;
            }
        }
    } else if (filtrado == 2){
        if (historial == nullptr) {
            cout << "El historial esta vacío." << endl;
        } else{
            NodoCancion* aux = historial; // Creo un aux igual a historial
            historial = nullptr; // Vacío el historial
            while (aux != nullptr){ // Invertir el orden del historial
                NodoCancion* temp = aux;
                aux = aux->siguiente;
                temp->siguiente = historial;
                historial = temp;
            }
            int i = 1;
            while (historial != nullptr){
                string cancionEscuchada = historial->cancion.nombre;
                cout << i << ". " << cancionEscuchada << endl;
                NodoCancion* temp = historial;
                historial = historial->siguiente;
                delete temp;
                i++;
            }
        }
    } else{
        cout << "Opcion no valida" << endl;
    }
}
```

```
void promedio(NodoCancion* historial){
    if (historial == nullptr) {
        cout << "El historial esta vacío." << endl;
    } else{
        int suma = 0;
        int cantidad = 0;
        while (historial != nullptr){
            suma += historial->cancion.puntuacion;
            NodoCancion* temp = historial;
            historial = historial->siguiente;
            cantidad++;
            delete temp;
        }

        float promedio = suma/cantidad;

        cout << "La puntuancion promedio es de " << promedio << "
estrellas." << endl;

    }
}
```

5) Partido de Fútbol:

```
// FUNCIÓN 13: Calcular la cantidad de personas en espera
int espera(Nodo* colaFrente, Nodo* prioridadFrente, bool prioridad){
    int cantidad = 0;

    // Cantidad de personas en la cola de prioridad
    while (prioridadFrente != nullptr){
        prioridadFrente = prioridadFrente->siguiente;
        cantidad++;
    }

    // Cantidad de personas en la cola de espera
    if (!prioridad){
        while (colaFrente != nullptr){
            colaFrente = colaFrente->siguiente;
            cantidad++;
        }
    }
    return cantidad - 1;
}

// Función para cargar la cantidad de personas en espera en la lista de
espera
```

```
void cargarLista(NodoHistorial*& inicio, int valor) {
    NodoHistorial* nuevoNodo = crearNodo(valor);
    if (inicio == nullptr){
        inicio = nuevoNodo;
    } else {
        NodoHistorial* actual = inicio;
        while (actual->siguiente != nullptr) {
            actual = actual->siguiente;
        }
        actual->siguiente = nuevoNodo;
    }
}
```

---

2. Árboles:

1) Árbol de Clientes: Implementa un programa que permita agregar clientes a un árbol binario de búsqueda basado en su número de DNI. Cada cliente debe tener un nombre y un DNI.

→ Agregar clientes.
→ Mostrar clientes en inorden según su DNI.

```
// FUNCIÓN 1: Ingresar un cliente, parámetro: dir. mmria raíz árbol
void agregarCliente (TreeNode*& root){
   string nombre;
   int dni;
   cout << "Ingrese el Nombre del cliente: " << endl, cin >> nombre;
   cout << "Ingrese el DNI del cliente: " << endl, cin >> dni;
   root = insertar(root, dni, nombre);
   cout << "Los datos se registraron correctamente!" << endl;
}
```

```
// FUNCIÓN 2: Crear Nodo de árbol con un valor dado
TreeNode* crearNodo(int dni, string nombre){
   TreeNode *nuevoNodo = new TreeNode;
   nuevoNodo->dni = dni;
   nuevoNodo->nombre = nombre;
   nuevoNodo->left = nullptr;
   nuevoNodo->right = nullptr;
   return nuevoNodo;
}
```

```
// FUNCIÓN 3: Insertar elemento en un árbol
TreeNode* insertar(TreeNode *root, int dni, string nombre){
   if (root == nullptr){
     return crearNodo(dni, nombre);
   }

   if (dni < root->dni){
     root->left = insertar(root->left, dni, nombre);
   } else if (dni > root->dni){
     root->right = insertar(root->right, dni, nombre);
   }
   return root;
}
```

```
// FUNCIÓN 4: Recorrido Inorden
void inorden(TreeNode* root){
   if (root == nullptr) {
     return;
   }

   inorden(root->left);
   cout << "Nombre: " << root->nombre << " con DNI: " << root->dni << endl;
inorden(root->right);
```

```
}
```

2) Búsqueda de Clientes : Los clientes pueden buscar su info a partir de su DNI. Si no se encuentran en el árbol, se muestra un mensaje adecuado.

```
bool buscarCliente(Cliente* root, Cliente cliente, bool& encontrado){
   if(root == nullptr){
     return false;
   }
   // Buscar en el subárbol izquierdo
   buscarCliente(root->left, cliente, encontrado);
   if(root->dni == cliente.dni){
     encontrado = true;
     cliente.nombre = root->nombre;
     cout << "Nombre: " << cliente.nombre << endl;
     cout << "DNI: " << cliente.dni << endl;
   }
   if(encontrado){
     return true;
   }
   // Buscar en el subárbol derecho
   buscarCliente(root->right, cliente, encontrado);
   if(encontrado){
     return true;
   } }
```

---

3. Ejercicio de Final

```
struct NodoCola{
    char paciente[41];
    NodoCola* sgte;
};
struct TipoInfoPrioridades{
    NodoCola* frente;
    NodoCola* fin;
};
struct NodoPrioridad{
    TipoInfoPrioridades info;
    NodoPrioridad* sgte;
};
```

```
// FUNCIÓN 1: Crea una cola de prioridad con cinco nodos, en cada uno encontraremos una lista. Devuelve un puntero al primer nodo de la cola de prioridad.
NodoPrioridad* crearColaPrioridad(){
   NodoPrioridad* lista = nullptr;
   TipoInfoPrioridades info;
   info.frente = nullptr;
   info.fin = nullptr;
   for(int i = 0; i < 5; i++){
     enlistar(lista, info);
   }
   return lista;
}

// FUNCIÓN 2: Insertar un paciente en la correcta lista de prioridad
void insertarEnPrioridad(NodoPrioridad* colaPrioridad, int prioridad, char paciente[]){
   NodoPrioridad* aux = colaPrioridad;
   for(int i = 0; i < 5; i++){
     if (i == prioridad){
       encolar(aux->info.frente, aux->info.fin, paciente);
       break;
     }
     aux = aux->sgte;
```

```
    i++;
  } }

// FUNCIÓN 3: Busca el primer elemento no vacío en una lista de prioridad, devuelve su nodo
NodoPrioridad* elementoMinimo(NodoPrioridad* lista){
  NodoPrioridad* aux = lista;
  for (int i = 0; i < 5; i++){
    if (aux->info.frente != nullptr){
      return aux;
    }
    aux = aux->sgte;
    i++;
  } }
// FUNCIÓN 4: Busca el primer elemento no vacío en una lista de prioridad, devuelve su contenido
string quitarMinimo(NodoPrioridad* lista){
  NodoPrioridad* aux = lista;
  for (int i = 0; i < 5; i++){
    if (aux->info.frente != nullptr){
      return aux->info.frente->paciente;
    }
    aux = aux->sgte;
    i++;
  } }

// FUNCIÓN 5: Verifica si la lista de prioridad está vacía.
int prioridadVacia(NodoPrioridad* lista){
  NodoPrioridad* aux = lista;
  for (int i = 0; i < 5; i++){
    if (aux->info.frente != nullptr){
      return 0;
    }
    aux = aux->sgte;
    i++;
  }
  return 1;
}
```

---

4. Parciales

1) 1er Parcial:

```
struct Pedido{
    int numeroPedido;
    string descripcion;
    int urgencia;
};

struct NodoPedido{
    Pedido pedido;
    NodoPedido* sgte;
};

struct ColaPedidos{
    NodoPedido* frente;
    NodoPedido* fin;
};

ColaPedidos colas[5];

void inicializarColas(ColaPedidos colas[], int cantidad){
    for (int i = 0; i < cantidad; i++){
        colas[i].frente = nullptr;
        colas[i].fin = nullptr;
    }}

void agregarPedido(ColaPedidos colas[], int numeroPedido, string
descripcion, int urgencia){ // también se puede incluir funciones de
crearPedido y crearNodo
    NodoPedido* nuevoNodo = new NodoPedido;
    nuevoNodo->pedido.numeroPedido = numeroPedido;
    nuevoNodo->pedido.descripcion = descripcion;
    nuevoNodo->pedido.urgencia = urgencia;
    nuevoNodo->sgte = nullptr;
    if (urgencia > 0 && urgencia < 6){
        if (colas[urgencia - 1].frente == nullptr){
            colas[urgencia - 1].frente = colas[urgencia - 1].fin =
nuevoNodo;
        } else {
            colas[urgencia - 1].fin->sgte = nuevoNodo;
            colas[urgencia - 1].fin = nuevoNodo;
        }
    } else {
        cout << "La urgencia debe estar en el rango de 1 a 5." << endl;
```

```
        delete nuevoNodo;
    }}

// FUNCIÓN 1: Retorna el pedido más urgente de la primera cola no vacía.
// Versión A (con cantidad)
Pedido pedidoMasUrgente(ColaPedidos colas[], int cantidad){
    Pedido pedidoMasUrgente;
    pedidoMasUrgente.urgencia = 6; // Inicializar con un valor fuera del
rango.
    for (int i = 0; i < cantidad; i++){
        if (colas[i].frente != nullptr){
            pedidoMasUrgente = colas[i].frente->pedido;
            NodoPedido* temp = colas[i].frente;
            colas[i].frente = colas[i].frente->sgte;
            if (colas[i].frente == nullptr){
                colas[i].fin = nullptr;
            }
            delete temp;
            return pedidoMasUrgente;
        }}
    return pedidoMasUrgente;
}

// Versión B (sin cantidad)
Pedido pedidoMasUrgente(ColaPedidos colas[]){
    int i = 0;
    while (colas[i].frente == nullptr){
        i++;
    }
    string pedidoUrgente = colas[i].frente->pedidos.numeroPedido;
    Nodo* temp = colas[i].frente-> sgte;
    if(colas[i].frente == nullptr) {
      colas[i].fin = nullptr
    }
    delete temp;
    return pedidoUrgente;
}

int colaVacia(ColaPedidos colas[], int cantidad){
    for (int i = 0; i < cantidad; i++){
        if (colas[i].frente != nullptr){
```

```
            return 0; // Al menos una cola no está vacía.
        }}
    return 1; // Todas las colas están vacías.
}
```

2) 2do Parcial:

```
struct NodoTarea{
    string descripcion;
    NodoTarea* sgte;
};

struct ListaPorTipo{
    NodoTarea* pilaDeTareas;
    int codigoTipo;
    ListaPorTipo *sgte;
};

ListaPorTipo* lista;

// Crear lista de tareas según su tipo
ListaPorTipo* agregarNuevaListaPorTipo(ListaPorTipo *lista, int
codigoTipo){
    ListaPorTipo* tipoActual = new ListaPorTipo;
    tipoActual->codigoTipo = codigoTipo;
    tipoActual->pilaDeTareas = nullptr;
    tipoActual->sgte = nullptr;

    // Agregar la nueva lista al final de la lista principal
    ListaPorTipo* tipoAnterior = nullptr;
    ListaPorTipo* tipoAuxiliar = lista;
    while (tipoAuxiliar != nullptr){
        tipoAnterior = tipoAuxiliar;
        tipoAuxiliar = tipoAuxiliar->sgte;
    }
    // Si la lista principal está vacía, hacer que la nueva lista sea la
primera
    if (tipoAnterior == nullptr){
        lista = tipoActual;
    }
    // De lo contrario, agregar la nueva lista al final de la lista
```

```
principal
    else{
        tipoAnterior->sgte = tipoActual;
    }
    return tipoActual;
}

/*Agrega una tarea a la pila de acuerdo a su código. Si no existe pila
de dicho tipo debe agregarla, puedes usar la función del punto 1.
*/
void agregarTarea(ListaPorTipo *lista, int codigoTipo, string
descripcion){
    ListaPorTipo* tipoActual = lista;
    ListaPorTipo* tipoAnterior = nullptr;
    // Buscar la lista correspondiente al códigoTipo
    while (tipoActual != nullptr && tipoActual->codigoTipo != codigoTipo){
        tipoAnterior = tipoActual;
        tipoActual = tipoActual->sgte;
    }
    // Si no se encuentra la lista correspondiente, crear una nueva
    if (tipoActual == nullptr){
        tipoActual = agregarNuevaListaPorTipo(lista, codigoTipo); //
Llamamos a la función para crear una nueva lista.
    }
    // Agregar la tarea a la pila de tareas
    NodoTarea* nuevaTarea = new NodoTarea;
    nuevaTarea->descripcion = descripcion;
    nuevaTarea->sgte = tipoActual->pilaDeTareas;
    tipoActual->pilaDeTareas = nuevaTarea;
}

// Retorna la próxima tarea correspondiente al código expuesto. Retorna
“No tengo” si no hay
string dameProximaTarea(ListaPorTipo* lista, int codigoTipo){
    ListaPorTipo* tipoActual = lista;
    // Buscar la lista correspondiente al códigoTipo
    while (tipoActual != nullptr && tipoActual->codigoTipo != codigoTipo){
        tipoActual = tipoActual->sgte;
    }
    if (tipoActual == nullptr || tipoActual->pilaDeTareas == nullptr){
        return "No tengo";
    }
```

```
    // Obtener la próxima tarea de la pila
    string descripcion = tipoActual->pilaDeTareas->descripcion;
    NodoTarea* tareaAEliminar = tipoActual->pilaDeTareas;
    tipoActual->pilaDeTareas = tipoActual->pilaDeTareas->sgte;
    delete tareaAEliminar;
    return descripcion;
}

// Retorna la cantidad de pilas con tareas pendientes.
int dameCantidadDePilasConTareasPendientes(ListaPorTipo *lista){
    int cantidad = 0;
    ListaPorTipo* tipoActual = lista;
    while (tipoActual != nullptr){
        if (tipoActual->pilaDeTareas != nullptr){
            cantidad++;
        }
        tipoActual = tipoActual->sgte;
    }
    return cantidad;
}
```

- Como no se sabe la cantidad final, podemos poner un int i = 1 al principio, para no utilizar un for. Lo utilizamos para pila y cola. Para listas, utilizamos el actual y el anterior.