República Bolivariana de Venezuela
Universidad Central de Venezuela
Facultad de Ciencias
Escuela de Computación
Algoritmos y Estructuras de Datos

PROYECTO: ORQUESTADOR DE REDES DE CONECTIVIDAD

Profesor:
Marcel Isaac

Estudiantes:
César Carios, C.I. 30.136.117
Jhonatan Homsany, C.I. 30.182.893

Junio, 2023

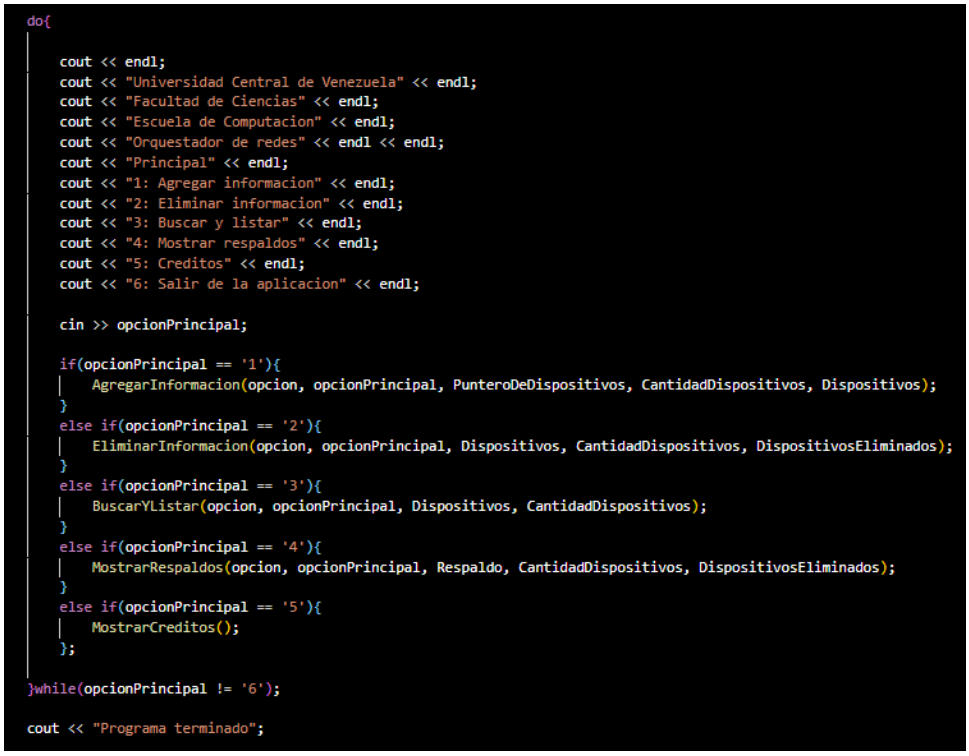
ÍNDICE

Funcionalidades del Programa

A continuación, se explicarán cada una de las funcionalidades del programa y cómo se tradujeron a código y luego se profundizará en la función main().

Menú Principal

En el menú principal, al igual que en todos los demás submenús, lo primero que se visualiza es un membrete identificando la universidad, facultad, escuela, entre otros. Desde este menú se accede a las demás funciones del programa y también tiene una opción para terminar el mismo. Este menú se hizo con una estructura de control do-while, como se observa en la imagen siguiente:



```
do{

    cout << endl;
    cout << "Universidad Central de Venezuela" << endl;
    cout << "Facultad de Ciencias" << endl;
    cout << "Escuela de Computacion" << endl;
    cout << "Orquestador de redes" << endl << endl;
    cout << "Principal" << endl;
    cout << "1: Agregar informacion" << endl;
    cout << "2: Eliminar informacion" << endl;
    cout << "3: Buscar y listar" << endl;
    cout << "4: Mostrar respaldos" << endl;
    cout << "5: Creditos" << endl;
    cout << "6: Salir de la aplicacion" << endl;

    cin >> opcionPrincipal;

    if(opcionPrincipal == '1'){
        AgregarInformacion(opcion, opcionPrincipal, PunteroDeDispositivos, CantidadDispositivos, Dispositivos);
    }
    else if(opcionPrincipal == '2'){
        EliminarInformacion(opcion, opcionPrincipal, Dispositivos, CantidadDispositivos, DispositivosEliminados);
    }
    else if(opcionPrincipal == '3'){
        BuscarYListar(opcion, opcionPrincipal, Dispositivos, CantidadDispositivos);
    }
    else if(opcionPrincipal == '4'){
        MostrarRespaldos(opcion, opcionPrincipal, Respaldo, CantidadDispositivos, DispositivosEliminados);
    }
    else if(opcionPrincipal == '5'){
        MostrarCreditos();
    };

}while(opcionPrincipal != '6');

cout << "Programa terminado";
```

Dependiendo del número que se presione, el programa mostrará un submenú de una función u otra.

Clases del Programa (POO)

Para este programa se crearon dos clases: una clase Dispositivo y una clase Pila.

La clase Dispositivo tiene como atributos dos strings: uno para el hostname y otro para el IP. También tiene una variable de tipo entero que representa el índice del arreglo Relación Se consideró necesario hacer un arreglo llamado Relación de tamaño 12, que tiene como función representar las relaciones de cada uno de los dispositivos. Este arreglo es de tipo Relaciones, un struct declarado previamente con tres características: un entero para representar el ping, un string para el tipo de relación (5G, Fibra Óptica, Mixta) y un apuntador que genera la conexión pedida en la entrada entre dispositivos.

```
class Dispositivo{
    public:
    string hostname;
    string IP;
    int indice;
    Relaciones Relacion[12];

    Dispositivo(){
        indice = 0;
    }
    ~Dispositivo(){}
};
```

```
struct Relaciones{
    Dispositivo *router = NULL;
    int ping;
    string conexion;
};
```

Para manejar la estructura de datos Pila, se creó una clase con el mismo nombre. Esta clase Pila tiene como atributos una variable de tipo entero que hace la función de ser un puntero de la pila y también tiene un apuntador que recorre todos los dispositivos insertados en la pila. Este apuntador recorre un arreglo de punteros a Dispositivos llamado Apilados de tamaño 1000, que es la máxima cantidad de dispositivos que puede haber en el programa.

```
class Pila{
    public:
        int punteroDePila;
        Dispositivo *Apilados[1000];

        Pila(){
            punteroDePila = 0;
        }
```

La clase Pila tiene varios métodos que serán explicados uno por uno de manera breve:

1. Insertar(): Esta acción inserta dispositivos a la pila. Para ello, se ayuda del arreglo de Apilados, explicado anteriormente.

```
void insertar(Dispositivo *Router){
    Apilados[punteroDePila] = Router;
    punteroDePila++;
}
```

2. Extraer(): Como su nombre lo indica, esta acción extrae dispositivos del arreglo de Apilados.

```
Dispositivo *extraer(){
    Apilados[punteroDePila] = NULL;
    punteroDePila--;
    return Apilados[punteroDePila];
}
```

3. EstaEnPila(): Esta función es de tipo bool y verifica si un dispositivo está en pila o no. Si el dispositivo buscado está en la pila, retorna verdadero, en el caso contrario retorna falso.

```
bool EstaEnPila(Dispositivo *Comparacion){
    for(int i = 0; i < punteroDePila; i++){
        if(Comparacion == Apilados[i]){
            return true;
        }
    }
    return false;
}
```

4. ImprimirPila(): Esta acción imprime los elementos que haya en el arreglo de Apilados.

```
void ImprimirPila(){
    for(int i = 0; i < punteroDePila; i++){
        resultados << Apilados[i]->hostname << " ";
        cout << (*Apilados[i]).hostname << " ";
    }
}
```

5. VaciarPila(): Esta acción borra a todos los elementos que haya dentro del arreglo de Apilados.

```
void VaciarPila(){
    for(int i = 0; i < punteroDePila; i++){
        Apilados[i] = NULL;
    }
    punteroDePila = 0;
}
```

6. EstaVacia(): Esta función es de tipo bool y verifica si el arreglo de Apilados está vacío o no validando si el puntero de Pila está a apuntando a alguna posición. Si el puntero de Pila es 0, significa que la Pila está vacía y retorna verdadero, si no, retorna falso.

```
bool EstaVacia(){
    if(punteroDePila == 0){
        return true;
    }
    return false;
}
```

7. ImprimirPing(): Esta función es de tipo entero e imprime el ping de los dispositivos.

```
int ImprimirPing(){
    int aux = 0;
    for(int i = 0; i < punteroDePila; i++){
        for(int j = 0; j < 12; j++){
            if(Apilados[i]->Relacion[j].router == Apilados[i+1]){
                aux += Apilados[i]->Relacion[j].ping;
            }
        }
    }
    return aux;
}
```

Agregar Información

En el submenú de agregar información tenemos dos funciones principales para cumplir con los requisitos del programa: AgregarDispositvo() y AgregarRelacionDesdeConsola(). Estos serán explicados a continuación:

```
void AgregarInformacion(char opcion, char &opcionPrincipal, int &PunteroDeDispositivos, int &cantidadDeDispositivos, Dispositivo Dispositivos[]){
    cout << endl;
    cout << "Universidad Central de Venezuela" << endl;
    cout << "Facultad de Ciencias" << endl;
    cout << "Escuela de Computacion" << endl;
    cout << "Orquestador de redes" << endl << endl;
    cout << "Agregar Informacion" << endl;
    cout << "1: Agregar dispositivos" << endl;
    cout << "2: Agregar ruta" << endl;
    cout << "3: Volver a Principal" << endl;
    cout << "4: Salir de la aplicacion" << endl;
    cin >> opcion;

    switch(opcion){
        case '1':
            AgregarDispositivo(PunteroDeDispositivos, Dispositivos, cantidadDeDispositivos);
        break;
        case '2':
            AgregarRelacionDesdeConsola(Dispositivos, cantidadDeDispositivos);
        break;

        case '4':
            opcionPrincipal = '6';
            return;
        break;
    }
    return;
}
```

- AgregarDispositivo(): Esta acción le pide al usuario el nombre y el IP del dispositivo que va a agregar. Primero verifica si el dispositivo que está siendo agregado no existe, si se da el caso de que el dispositivo en cuestión existe, entonces no será agregado y el programa le comunicará esto al usuario. Pero si el dispositivo no existe, entonces será agregado sin problemas. Para agregar a un dispositivo se usa la variable PunteroDeDispositivos explicada anteriormente en este documento para indicar la posición donde será almacenado el dispositivo en cuestión.

```
void AgregarDispositivo(int &PunteroDeDispositivos, Dispositivo Dispositivos[], int &cantidadDeDispositivos){
    string hostnameAgg;
    string IPAgg;

    cout << "Introduzca el nombre del dispositivo a agregar" << endl;
    cin >> hostnameAgg;
    cout << "Introduzca la direccion IP del dispositivo a agregar" << endl;
    cin >> IPAgg;

    if(hostnameAgg.at(hostnameAgg.length() - 1) == ','){
        hostnameAgg.pop_back();
    }
    if(IPAgg.at(IPAgg.length() - 1) == ','){
        IPAgg.pop_back();
    }

    for(int i = 0; i < cantidadDeDispositivos; i++){
        if(Dispositivos[i].hostname == hostnameAgg || Dispositivos[i].IP == IPAgg){
            resultados << "0" << endl;
            cout << "0: El dispositivo ya existe en el sistema" << endl;
            return;
        }
    }
    Dispositivos[PunteroDeDispositivos].hostname = hostnameAgg;
    Dispositivos[PunteroDeDispositivos].IP = IPAgg;
    PunteroDeDispositivos++;
    cantidadDeDispositivos++;
    resultados << "1" << endl;
    cout << "1: El dispositivo " << hostnameAgg << " ha sido agregado." << endl;
}
```

- AgregarRelacionDesdeConsola(): Esta acción primero le pide al usuario que ingrese los nombres de los dispositivos que quiere conectar o sus IP´s, el ping de la conexión y el tipo de la misma.

```
void AgregarRelacionDesdeConsola(Dispositivo Dispositivos[], int cantidadDeDispositivos){
    string hostnameR1, hostnameR2, pingEntrada, tipoC;
    int ping = 0;
    Dispositivo *Dispositivo1, *Dispositivo2;

    cout << "Ingrese hostname1, hostname2, ping y tipo de conexion" << endl;
    cin >> hostnameR1 >> hostnameR2 >> pingEntrada >> tipoC;

    if(hostnameR1.at(hostnameR1.length() - 1) == ','){
        hostnameR1.pop_back();
    }
    if(hostnameR2.at(hostnameR2.length() - 1) == ','){
        hostnameR2.pop_back();
    }
    if(pingEntrada.at(pingEntrada.length() - 1) == ','){
        pingEntrada.pop_back();
    }

    if(tipoC.at(tipoC.length() - 1) == ','){
        tipoC.pop_back();
    }

    ping = stoi(pingEntrada);

    for(int i = 0; i < cantidadDeDispositivos; i++){
        if(hostnameR1 == Dispositivos[i].hostname || hostnameR1 == Dispositivos[i].IP){Dispositivo1 = &Dispositivos[i];}

        if(hostnameR2 == Dispositivos[i].hostname || hostnameR2 == Dispositivos[i].IP){Dispositivo2 = &Dispositivos[i];}
    }
```

A fines de evitar fallos en el programa, al momento de leer una entrada se valida que el último carácter de su nombre no contenga comas, en caso de contenerla, esta es eliminada.

Posteriormente, asigna los nombres introducidos por el usuario al atributo hostnames de la clase Dispositivos. Luego procede con la validación de los nombres introducidos: se verifica si ambos dispositivos existen, si uno existe o si ya existe una conexión entre ellos.

```
if(Dispositivo1->hostname != hostnameR1 && Dispositivo2->hostname == hostnameR2){
    resultados << "-3" << endl;
    cout << "-3: El dispositivo origen no ha sido encontrado" << endl;
    return;
}
else if(Dispositivo2->hostname != hostnameR2 && Dispositivo1->hostname == hostnameR1){
    resultados << "-2" << endl;
    cout << "-2: El dispositivo destino no ha sido encontrado" << endl;
    return;
}
else if (Dispositivo1->hostname != hostnameR1 && Dispositivo2->hostname == hostnameR2){
    resultados << "-1" << endl;
    cout << "-1: Ninguno de los dispositivos existen" << endl;
    return;
}
for(int i = 0; i < cantidadDeDispositivos; i++){
    if(Dispositivo1->Relacion[i].router == Dispositivo2){
        resultados << "0" << endl;
        cout << "0: Ya existe una conexion entre ambos dispositivos" << endl;
        return;
    }
}
```

Si ninguna de esas verificaciones se cumple, entonces se procede a hacer la conexión entre los dispositivos introducidos asignándolos al arreglo Relacion de la clase Dispositivos en la posición que se encuentre vacía. ArreglarRelacionDesdeEntrada() hace lo mismo, pero funciona desde la entrada del programa.

```
for(int i = 0; i < 12; i++){
    if(Dispositivo1->Relacion[i].router == NULL){
        Dispositivo1->Relacion[i].router = Dispositivo2;
        Dispositivo1->Relacion[i].ping = ping;
        Dispositivo1->Relacion[i].conexion = tipoC;
        Dispositivo1->indice++;
        break;
    }
}
for(int i = 0; i < 12; i++){
    if(Dispositivo2->Relacion[i].router == NULL){
        Dispositivo2->Relacion[i].router = Dispositivo1;
        Dispositivo2->Relacion[i].ping = ping;
        Dispositivo2->Relacion[i].conexion = tipoC;
        Dispositivo2->indice++;
        break;
    }
}
resultados << "1" << endl;
cout << "1: Operacion exitosa" << endl;
```

Eliminar Información

En el submenú de agregar información tenemos dos funciones principales para cumplir con los requisitos del programa: EliminarDispositivo() y EliminarRuta(). Estos serán explicados a continuación:

```
void EliminarInformacion(char opcion, char &opcionPrincipal, Dispositivo Dispositivos[], int cantidadDeDispositivos, int &DispositivosEliminados)
    cout << endl;
    cout << "Universidad Central de Venezuela" << endl;
    cout << "Facultad de Ciencias" << endl;
    cout << "Escuela de Computacion" << endl;
    cout << "Orquestador de redes" << endl << endl;
    cout << "Eliminar Informacion" << endl;
    cout << "1: Eliminar dispositivo" << endl;
    cout << "2: Eliminar ruta" << endl;
    cout << "3: Volver a Principal" << endl;
    cout << "4: Salir de la aplicacion" << endl;
    cin >> opcion;

    switch(opcion){
        case '1':
            EliminarDispositivo(Dispositivos, cantidadDeDispositivos, DispositivosEliminados);
        break;
        case '2':
            EliminarRuta(Dispositivos, cantidadDeDispositivos);
        break;

        case '4':
            opcionPrincipal = '6';
            return;
        break;
    }
    return;
}
```

- EliminarDispositivo(): Esta acción primero pide el nombre del dispositivo que se quiere eliminar. Luego se verifica que el dispositivo introducido exista, esto

se hace con un for que recorre todos los hostnames que han sido introducidos con anterioridad en el programa. Si el dispositivo no existe, entonces le dice al usuario el código de error ocasionado.

```
void EliminarDispositivo(Dispositivo Dispositivos[], int &cantidadDeDispositivos, int &DispositivosEliminados){
    string hostnameEl;
    Dispositivo *Dispositivo1;
    bool Existe = false;

    cout << "Ingrese el nombre del dispositivo a eliminar" << endl;
    cin >> hostnameEl;

    for(int i = 0; i < cantidadDeDispositivos; i++){
        if(Dispositivos[i].hostname == hostnameEl || Dispositivos[i].IP == hostnameEl){
            Dispositivo1 = &Dispositivos[i];
            Existe = true;
            break;
        }
    }

    if(!Existe){
        cout << "El dispositivo no existe" << endl;
        resultados << "0" << endl;
        return;
    }
```

Por otro lado, si el dispositivo existe entonces se procede a eliminar tanto el dispositivo como las rutas directas que tenía. Y luego de que el dispositivo se elimina, se llama a una función que se encarga de tener ordenados los dispositivos por orden alfabético.

```
for(int i = 0; i < 12; i++){
    for(int j = 0; j < cantidadDeDispositivos; j++){
        if(Dispositivos[i].Relacion[j].router == Dispositivo1){
            Dispositivos[i].Relacion[j].router = NULL;
        }
    }
}

for(int i = 0; i < 12; i++){
    Dispositivo1->Relacion[i].router = NULL;
}

for(int i = 0; i < cantidadDeDispositivos; i++){
    if(Dispositivos[i].hostname == hostnameEl){
        DispositivosRespaldo << Dispositivos[i].hostname << " " << Dispositivos[i].IP << endl;
        Dispositivos[i].hostname = "{";
        break;
    }
}

OrdenamientoDispositivosNombre(Dispositivos, cantidadDeDispositivos);
cantidadDeDispositivos--;
DispositivosEliminados++;
cout << "Se ha realizado exitosamente" << endl;
resultados << "1" << endl;
```

- EliminarRuta(): Inicialmente esta función solicita el nombre o dirección IP de los dispositivos a los que se les eliminará todas las rutas entre ellos (directas e indirectas). Se verifica que existen y en caso de que no exista alguno de ellos (o ninguno) se imprime el tipo de error y se retorna.

```
void EliminarRuta(Dispositivo Dispositivos[], int cantidadDeDispositivos){
    Dispositivo *Raiz, *Destino;
    Pila Atravesados, Soluciones;
    string Dispositivo1, Dispositivo2;
    int contador = 0;
    bool Existe1 = false, Existe2 = false, Solucion = false;
    cout << "Ingrese el nombre o direccion IP (solo uno) de ambos dispositivos" << endl;
    cin >> Dispositivo1 >> Dispositivo2;

    for(int i = 0; i < cantidadDeDispositivos; i++){
        if(Dispositivos[i].hostname == Dispositivo1 || Dispositivos[i].IP == Dispositivo1){
            Raiz = &Dispositivos[i];
            Existe1 = true;
        }
        if(Dispositivos[i].hostname == Dispositivo2 || Dispositivos[i].IP == Dispositivo2){
            Destino = &Dispositivos[i];
            Existe2 = true;
        }
    }

    if(Existe1 && !Existe2){
        cout << "-2: El dispositivo destino no existe" << endl;
        resultados << "-2" << endl;
        return;
    }
    else if(!Existe1 && Existe2){
        cout << "-3: El dispositivo origen no existe" << endl;
        resultados << "-3" << endl;
        return;
    }
```

```
else if(!Existe1 && !Existe2){
    cout << "-1: Ambos dispositivos no existen" << endl;
    resultados << "-1" << endl;
    return;
}
```

Si ambos dispositivos introducidos existen, se procede a usar la función BacktrackingEliminar() (Véase el apartado de otras funciones) para verificar que los dispositivos introducidos tengan una relación. Esta función recorre todas las relaciones que tenga el programa en ese momento. Si encuentra una conexión entre los dispositivos introducidos, actualiza una variable booleana llamada Solucion y la inicializa en verdadero. Si Solucion es falsa entonces los dispositivos en cuestión no están conectados y se le comunica al usuario.

```
Atravesados.insertar(Raiz);
BacktrackingEliminar(Raiz, Destino, Raiz->indice, Atravesados, Soluciones, Solucion);
if(!Solucion){
    cout << "0: No existe un camino entre los Dispositivos que  los conecte" << endl;
    resultados.open("resultados.txt");
}
Atravesados.VaciarPila();
```

Por otro lado, si los dispositivos si están conectados entonces se procede a eliminar todas las rutas indirectas que haya entre ellos.

```
Dispositivo *Eliminar1, *Eliminar2;

for(int i = 0; i < Soluciones.punteroDePila - 1; i++){
    Eliminar1 = Soluciones.extraerLectura(i);
    Eliminar2 = Soluciones.extraerLectura(i + 1);
    EliminarRutaDirecta(Eliminar1, Eliminar2);
    OrdenamientoRelaciones(Eliminar1->Relacion);
    OrdenamientoRelaciones(Eliminar2->Relacion);
}

cout << "Entre los dispositivos " << Dispositivo1 << " y " << Dispositivo2 << " se eliminaron las siguientes rutas: " << endl;
for(int i = 0; i < Soluciones.punteroDePila; i++){
    cout << Soluciones.extraerCola(i)->hostname << " ";
    if(Soluciones.extraerCola(i)->hostname == Destino->hostname){
        contador++;
        cout << endl;
    }
}
Soluciones.VaciarPila();

cout << "Total de rutas eliminadas: " << contador << endl;
```

Inicialmente, extraemos los elementos de la pila trabajándola como si se tratase de una cola, es decir, sacando el elemento en la posición i y el elemento en la posición i + 1, para luego enviarlos a la función de EliminarRutaDirecta(). Ahora bien, el funcionamiento de EliminarRutaDirecta() es más sencillo. Esta función recibe ambos dispositivos a los que se les debe eliminar las conexiones directas. Por último, se aumenta un contador que valida la cantidad de rutas que fueron eliminadas entre los dispositivos indicados por el usuario. Por último, se vacía la pila de soluciones, por si se requiere eliminar una ruta entre dos dispositivos nuevamente. La función EliminarRutaDirecta() recibe dos dispositivos y procede a recorrer sus relaciones, cuando se encuentre el otro dispositivo entre el arreglo de relaciones, se procede a eliminar los datos de la misma. Luego, en eliminar ruta, se ordenan las relaciones para que todas sean contiguas dentro del arreglo.

```
void EliminarRutaDirecta(Dispositivo *Eliminar1, Dispositivo *Eliminar2){
    RutasEliminadas++;
    for(int i = 0; i < 12; i++){
        if(Eliminar1->Relacion[i].router == Eliminar2){
            RutasRespaldo << Eliminar1->hostname << " ";
            Eliminar1->Relacion[i].router = NULL;
            Eliminar1->Relacion[i].ping = 0;
            Eliminar1->Relacion[i].conexion = "Vacio";
            Eliminar1->indice--;
        }
        if(Eliminar2->Relacion[i].router == Eliminar1){
            RutasRespaldo << Eliminar2->hostname << " " << Eliminar2->Relacion[i].ping << " " << Eliminar2->Relacion[i].conexion << endl;
            Eliminar2->Relacion[i].router = NULL;
            Eliminar2->Relacion[i].ping = 0;
            Eliminar2->Relacion[i].conexion = "Vacio";
            Eliminar2->indice--;
        }
    }
```

Buscar y Listar Información

En el submenú de buscar y listar información se tienen cuatro funciones principales que llevan a cabo el trabajo requerido para esta sección del programa: ConsultarDispositivos(), ListadoDeDispositivos(), BuscarRuta() y DispositivosAdyacentes(). A continuación, analizaremos una por una.

```
void BuscarYListar(char opcion, char &opcionPrincipal, Dispositivo Dispositivos[], int cantidadDeDispositivo
    cout << endl;
    cout << "Universidad Central de Venezuela" << endl;
    cout << "Facultad de Ciencias" << endl;
    cout << "Escuela de Computacion" << endl;
    cout << "Orquestador de redes" << endl << endl;
    cout << "Buscar y Listar" << endl;
    cout << "1: Consultar dispositivo." << endl;
    cout << "2: Listado de Dispositivos." << endl;
    cout << "3: Buscar ruta(1:5G, 2: fibra optica, 3: ambas)." << endl;
    cout << "4: Dispositivos adyacentes." << endl;
    cout << "5: Volver a Principal." << endl;
    cout << "6: Salir de la aplicacion." << endl;
    cin >> opcion;

    switch(opcion){
        case '1':
            ConsultarDispositivos(Dispositivos, cantidadDeDispositivos);
        break;
        case '2':
            ListadoDeDispositivos(Dispositivos, cantidadDeDispositivos);
        break;
        case '3':
            BuscarRuta(Dispositivos, cantidadDeDispositivos);
        break;
        case '4':
            DispositivosAdyacentes(Dispositivos, cantidadDeDispositivos);
        break;
        case '6':
            opcionPrincipal = '6';
            return;
        break;
    }
    return;
}
```

- ConsultarDispositivos(): Primero se le pide el nombre o la dirección IP del dispositivo que se quiera consultar, luego recorre el arreglo de dispositivos con un ciclo for y busca el nombre o IP introducido. Si la estructura de control implementada encuentra el dispositivo introducido, entonces imprime su nombre y su IP. Si no lo encuentra, se lo comunica a usuario.

```
void ConsultarDispositivos(Dispositivo Dispositivos[], int cantidadDeDispositivos){
    cout << endl;
    cout <<"Ingrese el nombre o la direccion IP del dispositivo a consultar" << endl;
    string Consulta;
    cin >> Consulta;
    for(int i = 0; i < cantidadDeDispositivos; i++){
        if(Consulta == Dispositivos[i].hostname || Consulta == Dispositivos[i].IP){
            cout << Dispositivos[i].hostname << " " << Dispositivos[i].IP << endl;
            return;
        }
    }
    cout << "Dispositivo no encontrado." << endl;
}
```

- ListadoDeDispositivos(): Lo primero que hace esta función es llamar a una función llamada OrdenamientoDispositivosNombre(), la misma se encagar de ordenar alfabéticamente a los dispositivos que estén en el programa.

```
void ListadoDeDispositivos(Dispositivo Dispositivos[], int cantidadDeDispositivos){

    OrdenamientoDispositivosNombre(Dispositivos, cantidadDeDispositivos);
    cout << "Listado de Dispositivos: " << endl;
    for(int i = 0; i < cantidadDeDispositivos; i++){
        cout << Dispositivos[i].hostname << " " << Dispositivos[i].IP << endl;
    }
}
```

Para hablar de la función BuscarRuta() es necesario hablar primero de la función Backtracking(), pues en ella se basa principalmente el programa para encontrar rutas entre dispositivos.

- Backtracking(): Esta función se encarga de emplear la técnica del Backtracking para encontrar las rutas entre dos dispositivos. Inicialmente, utiliza un arreglo de relaciones que es utilizado para obtener las relaciones del dispositivo raíz (el cual es el dispositivo que estamos analizando en la iteración actual del procedimiento recursivo). Posteriormente, se verifica si el dispositivo en la posición i de la relación forma parte de los dispositivos que hemos atravesado, en caso de que se encuentre almacenado en dicha pila, simplemente se observa el siguiente elemento. Luego, en caso contrario, se verifica si el dispositivo en la posición i de la iteración es el dispositivo destino, en caso de serlo, hemos encontrado una solución y procedemos a imprimirla (además de llenar el archivo resultados.out), luego, extraemos el elemento destino de la pila de atravesados y seguimos iterando en busca de más soluciones. En caso de que no sea el dispositivo destino, ingresamos el elemento siguiente a la pila de atravesados e iteramos con dicho elemento.

```
void Backtracking(Dispositivo *Raiz, Dispositivo *Destino, int indice, Pila &Atravesados, bool &Solucion, string tipoC){
    Relaciones RelacionActual[indice];

    for(int i = 0; i < indice; i++){
        RelacionActual[i] = Raiz->Relacion[i];
    }

    for(int i = 0; i < indice; i++){
        if(!Atravesados.EstaEnPila(RelacionActual[i].router)){
            if(RelacionActual[i].router->hostname == Destino->hostname){
                Atravesados.insertar(Destino);
                cout << "Solucion encontrada: " << endl;
                Atravesados.ImprimirPila();
                resultados << " Ping: " << Atravesados.ImprimirPing() << " Saltos: " << Atravesados.punteroDePila << " (Relacion, " << tipoC << "
                cout << " Ping: " << Atravesados.ImprimirPing() << " Saltos: " << Atravesados.punteroDePila << " (Relacion, " << tipoC << ")" <<
                Solucion = true;
                Atravesados.extraer();
            }
            else{
                Atravesados.insertar(RelacionActual[i].router);
                Backtracking(RelacionActual[i].router, Destino, RelacionActual[i].router->indice, Atravesados, Solucion, tipoC);
            }
        }
    }
    Atravesados.extraer();
}
```

- BuscarRuta(): Primero le pide al usuario los nombres de los dispositivos cuya ruta quiere buscar, incluyendo el tipo de conexión. Luego, se procede a verificar que los dispositivos introducidos existen. Si uno de ellos no existe o si no existen ninguno de los dos, se le informa al usuario con un código de error.

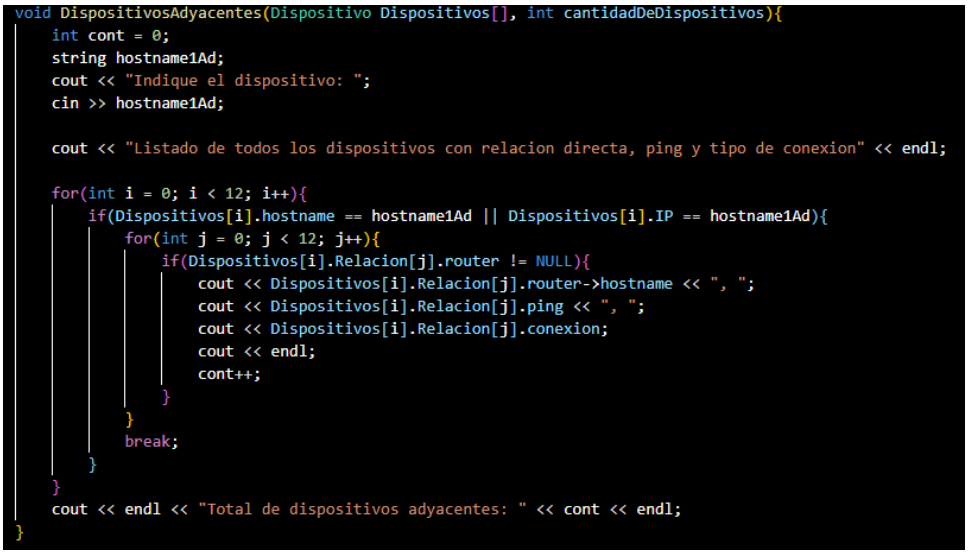
```
void BuscarRuta(Dispositivo Dispositivos[], int cantidadDeDispositivos){
    Pila Atravesados;
    bool Solucion = false, Existe1 = false, Existe2 = false;
    string dispositivo1Buscar, dispositivo2Buscar, Conexion;
    int tipoConexion;
    cout << "Indique los dispositivos" << endl;
    cin >> dispositivo1Buscar >> dispositivo2Buscar >> tipoConexion;

    Dispositivo *Dispositivo1, *Dispositivo2;

    for(int i = 0; i < cantidadDeDispositivos; i++){
        if(Dispositivos[i].hostname == dispositivo1Buscar || Dispositivos[i].IP == dispositivo1Buscar){
            Dispositivo1 = &Dispositivos[i];
            Existe1 = true;
        }
        else if(Dispositivos[i].hostname == dispositivo2Buscar || Dispositivos[i].IP == dispositivo2Buscar){
            Dispositivo2 = &Dispositivos[i];
            Existe2 = true;
        }
    }
```

```
if(Existe1 && !Existe2){
    cout << "-2: El dispositivo destino no existe" << endl;
    resultados << "-2" << endl;
    return;
}
else if(!Existe1 && Existe2){
    cout << "-3: El dispositivo origen no existe" << endl;
    resultados << "-3" << endl;
    return;
}
else if(!Existe1 && !Existe2){
    cout << "-1: Ambos dispositivos no existen" << endl;
    resultados << "-1" << endl;
    return;
}
```

Si los dos dispositivos introducidos existen, la función entra en una estructura de control switch que tiene tres casos dependiendo del tipo de conexión. En cada uno de los casos se llama a la función Backtracking() para que encuentre la ruta entre los dos dispositivos introducidos, si la encuentra entonces devuelve la solución. Si no, entonces devuelve un código de error al usuario.

```
switch(tipoConexion){
    case 1:
        Conexion = "5G";
        Atravesados.insertar(Dispositivo1);
        Backtracking(Dispositivo1, Dispositivo2, Dispositivo1->indice, Atravesados, Solucion, Conexion);
        if(!Solucion){
            resultados << "0" << endl;
            cout << endl << "No existe una relacion entre " << Dispositivo1->hostname << " y " << Dispositivo2->hostname << endl;
        }
        Atravesados.extraer();
    break;
    case 2:
        Conexion = "FibraOptica";
        Atravesados.insertar(Dispositivo1);
        Backtracking(Dispositivo1, Dispositivo2, Dispositivo1->indice, Atravesados, Solucion, Conexion);
        if(!Solucion){
            resultados << "0" << endl;
            cout << endl << "No existe una relacion entre " << Dispositivo1->hostname << " y " << Dispositivo2->hostname << endl;
        }
        Atravesados.extraer();
    break;
    case 3:
        Conexion = "Ambas";
        Atravesados.insertar(Dispositivo1);
        Backtracking(Dispositivo1, Dispositivo2, Dispositivo1->indice, Atravesados, Solucion, Conexion);
        if(!Solucion){
            resultados << "0" << endl;
            cout << endl << "No existe una relacion entre " << Dispositivo1->hostname << " y " << Dispositivo2->hostname << endl;
        }
        Atravesados.extraer();
    break;
}
```

- DispositivosAdyacentes(): Esta función es sencilla. Le pide al usuario el dispositivo del cual quiere saber sus vecinos adyacentes y luego, con la implementación de una estructura de control for, recorre los dispositivos que tengan una relación directa con el dispositivo introducido y luego los imprime. También imprime la cantidad de vecinos que tiene el dispositivo en cuestión. Esto último lo hace con un contador que aumenta su cantidad en cada iteración del for mencionado anteriormente.



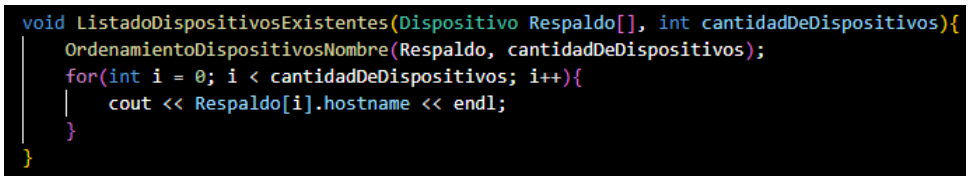
```
void DispositivosAdyacentes(Dispositivo Dispositivos[], int cantidadDeDispositivos){
    int cont = 0;
    string hostname1Ad;
    cout << "Indique el dispositivo: ";
    cin >> hostname1Ad;

    cout << "Listado de todos los dispositivos con relacion directa, ping y tipo de conexion" << endl;

    for(int i = 0; i < 12; i++){
        if(Dispositivos[i].hostname == hostname1Ad || Dispositivos[i].IP == hostname1Ad){
            for(int j = 0; j < 12; j++){
                if(Dispositivos[i].Relacion[j].router != NULL){
                    cout << Dispositivos[i].Relacion[j].router->hostname << ", ";
                    cout << Dispositivos[i].Relacion[j].ping << ", ";
                    cout << Dispositivos[i].Relacion[j].conexion;
                    cout << endl;
                    cont++;
                }
            }
            break;
        }
    }
    cout << endl << "Total de dispositivos adyacentes: " << cont << endl;
}
```

Respaldar Información

El submenú de respaldar información se apoya en tres funciones principales: ListadoDispositivosExistentes(), ListadoDispositivosEliminados() y ListadoRutasEliminadas(). Ahondaremos en cada una de estas funciones a continuación.

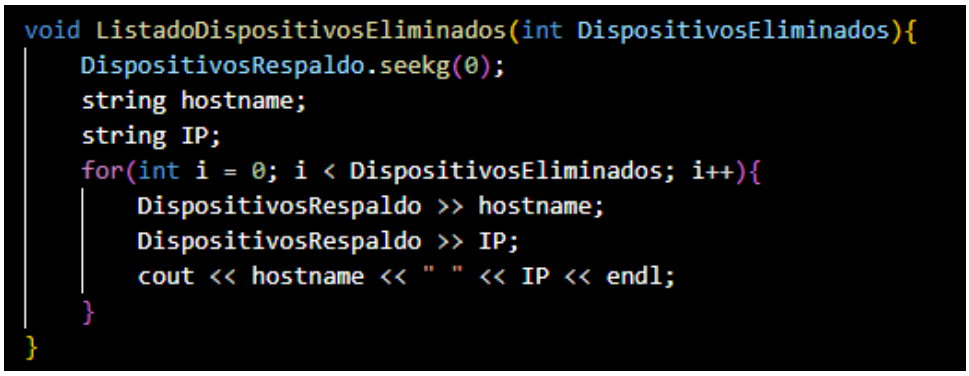
```
void MostrarRespaldos(char opcion, char &opcionPrincipal, Dispositivo Respaldo[], int cantidadDeDispositivos, int DispositivosEliminados){
    cout << endl;
    cout << "Universidad Central de Venezuela" << endl;
    cout << "Facultad de Ciencias" << endl;
    cout << "Escuela de Computacion" << endl;
    cout << "Orquestador de redes" << endl << endl;
    cout << "Mostrar Respaldos" << endl;
    cout << "1: Listado Dispositivos existentes." << endl;
    cout << "2: Listado Dispositivos eliminados." << endl;
    cout << "3: Listado rutas eliminadas." << endl;
    cout << "4: Volver a Principal." << endl;
    cout << "5: Salir de la aplicacion." << endl;
    cin >> opcion;

    switch(opcion){
        case '1':
            ListadoDispositivosExistentes(Respaldo, cantidadDeDispositivos);
        break;

        case '2':
            ListadoDispositivosEliminados(DispositivosEliminados);
        break;

        case '3':
            ListadoRutasEliminadas();
        break;

        case '5':
            opcionPrincipal = '6';
            return;
        break;
    }
    return;
}
```

- ListadoDispositivosExistentes(): Esta función llama a OrdenamientoDispositivosNombre() y luego imprime todos los nombres de los dispositivos que están el programa ordenado alfabéticamente.



```
void ListadoDispositivosExistentes(Dispositivo Respaldo[], int cantidadDeDispositivos){
    OrdenamientoDispositivosNombre(Respaldo, cantidadDeDispositivos);
    for(int i = 0; i < cantidadDeDispositivos; i++){
        cout << Respaldo[i].hostname << endl;
    }
}
```

- ListadoDispositivosEliminados(): Esta función escribe toda la información que haya en el archivo de respaldo de dispositivos. Esto lo hace con una estructura de control for.



```
void ListadoDispositivosEliminados(int DispositivosEliminados){
    DispositivosRespaldo.seekg(0);
    string hostname;
    string IP;
    for(int i = 0; i < DispositivosEliminados; i++){
        DispositivosRespaldo >> hostname;
        DispositivosRespaldo >> IP;
        cout << hostname << " " << IP << endl;
    }
}
```

- ListadoRutasEliminadas(): Similar a la función anterior, mediante un for escribe por consola todo lo que haya en el archivo de respaldo de las rutas.

```
void ListadoRutasEliminadas(){
    RutasRespaldo.seekg(0);
    string hostname1, hostname2, conexion;
    int ping = 0;
    for(int i = 0; i < RutasEliminadas; i++){
        RutasRespaldo >> hostname1;
        RutasRespaldo >> hostname2;
        RutasRespaldo >> ping;
        RutasRespaldo >> conexion;
        cout << hostname1 << " " << hostname2 << " " << ping << " " << conexion << endl;
    }
}
```

Función Main

Inicialmente, se leen y almacenan los datos de los dispositivos recibidos desde el archivo de entrada, se verifica que la dirección IP, así como el hostname no contengan comas al final del string, para así evitar generar errores dentro de la ejecución del programa, en caso de tenerlas, estas son eliminadas. Posteriormente, se realiza el mismo procedimiento al leer las relaciones. Por último, se imprime el menú, para dar comienzo a la simulación.

```
int main(){

    //Lectura de Archivo
    int CantidadDispositivos = 0;
    int PunteroDeDispositivos = 0;
    int DispositivosEliminados = 0;
    int CantidadRelaciones = 0;

    entrada >> CantidadDispositivos;
    Dispositivo Dispositivos[1000];
    Dispositivo Respaldo[CantidadDispositivos];

    while(PunteroDeDispositivos < CantidadDispositivos){
        entrada >> Dispositivos[PunteroDeDispositivos].hostname;
        entrada >> Dispositivos[PunteroDeDispositivos].IP;
        PunteroDeDispositivos++;
    }

    for(int i = 0; i < CantidadDispositivos; i++){
        if(Dispositivos[i].hostname.at(Dispositivos[i].hostname.length() - 1) == ','){
            Dispositivos[i].hostname.pop_back();
        }
        if(Dispositivos[i].IP.at(Dispositivos[i].IP.length() - 1) == ','){
            Dispositivos[i].IP.pop_back();
        }
    }

    entrada >> CantidadRelaciones;

    for(int i = 0; i < CantidadRelaciones; i++){ AgregarRelacionDesdeEntrada(Dispositivos, CantidadDispositivos, entrada); }

    for(int i = 0; i < CantidadDispositivos; i++){
        Respaldo[i] = Dispositivos[i];
    }
```

Otras Funciones

- BacktrackingEliminar(): Esta función realiza un trabajo similar a la función original de Backtracking, con la diferencia de que esta vez al encontrar una solución, no la imprimimos, sino que transferimos lo que se encuentra almacenado en la pila de Atravesados a una pila de Soluciones.

```
void BacktrackingEliminar(Dispositivo *Raiz, Dispositivo *Destino, int indice, Pila &Atravesados, Pila &Soluciones, bool &Solucion){
    Relaciones RelacionActual[indice];

    for(int i = 0; i < indice; i++){
        RelacionActual[i] = Raiz->Relacion[i];
    }

    for(int i = 0; i < indice; i++){
        if(!Atravesados.EstaEnPila(RelacionActual[i].router)){
            if(RelacionActual[i].router->hostname == Destino->hostname){
                Solucion = true;
                for(int i = 0; i < Atravesados.punteroDePila; i++){
                    Soluciones.insertar(Atravesados.Transferencia(i));
                }
                Soluciones.insertar(Destino);
            }
            else{
                Atravesados.insertar(RelacionActual[i].router);
                BacktrackingEliminar(RelacionActual[i].router, Destino, RelacionActual[i].router->indice, Atravesados, Soluciones, Solucion);
            }
        }
    }
    Atravesados.extraer();
}
```

- IntercambiarDispositivos(): Esta función es un algoritmo clásico de intercambiar valores entre dos variables. Específicamente, esta función intercambia la posición entre dos dispositivos. La función OrdenamientoDispositivosNombres() hace uso de esta función.

```
void intercambiarDispositivos(Dispositivo Dispositivos[], int i, int j){
    Dispositivo aux;
    aux = Dispositivos[i];
    Dispositivos[i] = Dispositivos[j];
    Dispositivos[j] = aux;
}
```

- IntercambiarRelaciones(): Análoga a la función IntercambiarDispositivos(), esta función es un algoritmo de intercambio que luego es usado por OrdenamientoRelaciones()

```
void IntercambiarRelaciones(Relaciones Relacion[], int i, int j){
    Relaciones aux;
    aux = Relacion[i];
    Relacion[i] = Relacion[j];
    Relacion[j] = aux;
}
```

- OrdenamientoRelaciones(): Esta función ordena las relaciones que haya en el programa. Es usada por EliminarRuta() para evitar que una relación deje de ser accesible al disminuir el índice de relaciones.

```
void OrdenamientoRelaciones(Relaciones Relacion[]){
    for(int o = 0, t = 12 - 1; o < 12; o++, t--){
        for(int i = 0; i < t; i++){
            if(Relacion[i].router == NULL && Relacion[i + 1].router != NULL){
                IntercambiarRelaciones(Relacion, i, i+1);
            }
        }
    }
}
```

- MostrarCreditos(): Una función que imprime el membrete del programa y los nombres de los autores del mismo, incluyendo la fecha de entrega del proyecto.

```
void MostrarCreditos(){
    cout << endl;
    cout << "Universidad Central de Venezuela" << endl;
    cout << "Facultad de Ciencias" << endl;
    cout << "Escuela de Computacion" << endl;
    cout << "Orquestador de redes" << endl << endl;
    cout << "Jhonatan Homsany C.I. 30.182.893" << endl;
    cout << "Cesar Carios C.I. 30.136.117" << endl;
    cout << "Los Teques, a los 5 dias del mes de Junio del 2023" << endl;
}
```