Javier Isaías Coyoy Marín **202030556**

Manual Técnico

Para empezar, tenemos las descripciones de las clases que se utilizaron en el programa para empezar tener la cadena de 3 componentes conformados por Conjunto, Campo y NodoValor, estos 3 elementos son los que van a estar enlazados para la comunicación de información y obtención de datos según se los pida el usuario y la descripción de cada una de las clases es la siguiente:

Clase Conjunto

Esta Clase es la primera que se utiliza al iniciar nuestro programa y esta va formar parte del arreglo inicial de 5 elementos de conjuntos que se van a ir inicializando con forme se creen los grupos, entre sus atributos tenemos el nombre que va guardar el nombre del grupo, el size que va guardar la longitud del arreglo y un arreglo que campos que va guardar la cantidad de campos de cada grupo, y el método más importante es el de arreglarCampo() ya que ubica el nuevo elemento en una índice del arreglo y le manda de parámetros su nombre, su tipo, y su orden, valores que se especificarán en el apartado de campos.

```
class Conjunto {
    private:
        string nombre = "";
        Campo* campos = NULL;
        int size = 0;

    public:
        Conjunto() {}
        void setDatos(string nombre, int size) {
            this->nombre = nombre;
            this->campos = new Campo[size];
            this->size = size;
        }
        void agregarCampo(string nombre, string tipo, int orden) {
            int ubicacion = funcionHash(nombre, this->size);

            for (int i = 0; i < size; i++) {
                if (campos[ubicacion].getNombre() == "") {
                    campos[ubicacion] = Campo(nombre, tipo, orden);
                    break;
                } else {
```

```
                    if (ubicacion != (size - 1)) {
                        ubicacion++;
                    } else {
                        ubicacion = 0;
                    }
                }
            }
        }
        string getNombre() { return this->nombre; }
        Campo* getCampos() { return this->campos; }
        int getSize() { return this->size; }
};
```

Clase Campo

Esta Clase inicia su vida dentro de cada Conjunto inicializado y este va guardar el nombre del campo, el tipo de dato si es String, Integer, entre otros, y el orden del campo este es utilizado para verificar cuando se haga un ingreso de datos que el valor entrante va ir a guardarse en su campo correspondiente, y por ultimo tenemos un atributo valores que sera el arreglo de valores que formaran el campo. Entre sus métodos el más relevante es el de guardarDatos(), y en este método obtenemos el contenido a guardar y buscamos un índice dentro del arreglo para poder poner nuestro elemento.

```
class Campo {
    private:
        string nombre;
        string tipo;
        NodoValor* valores;
        int size, orden;

    public:
        Campo() {}
        Campo(string nombre, string tipo, int orden) {
            this->nombre = nombre;
            this->tipo = tipo;
            this->valores = NULL;
            this->size = 5;
            this->orden = orden;
        }
        void guardarValor(string contenido) {
            int ubicacion = funcionHash(contenido, this->size);

            if (valores == NULL) {
```

```
                this->valores = new NodoValor[size];
            }
            for (int i = 0; i < size; i++) {
                if (valores[ubicacion].getContenido() == "") {
                    valores[ubicacion] = NodoValor(contenido);
                    break;
                } else {
                    if (ubicacion != (size - 1)) {
                        ubicacion++;
                    } else {
                        ubicacion = 0;
                    }
                }
            }
        }
        string getNombre() {
            return this->nombre;
        }
        string getTipo() {
            return this->tipo;
        }
        NodoValor* getValores() {
            return this->valores;
        }
        int getSize() {
            return this->size;
        }
        int getOrden() {
            return this->orden;
        }
};
```

Clase Nodo

Esta Clase es el último enlace entre los tres componentes principales del sistema, en esta clase se almacenará los valores de los datos que se ingresen en cada uno de los campos de un grupo seleccionado y también dentro de esta clase se va poder apuntar al valor siguiente del contacto para que así al momento de realizar una búsqueda al localizar el primer elemento de la “Lista” de nodos, por último el valor booleano apuntado es solo para determinar evitar que se haga un ciclo al momento de realizar las conexiones entre los nodos, y el valor booleano primero

es solo para establecer cuál es el primer valor de todos tomando como referencia los campos ingresados por el usuario.

```
struct Nodo {
    string contenido;
    int numero;
    int nivel;
    bool apuntando;
    bool primero;
    Nodo* compañero;
    Nodo* nodoDerecho;
    Nodo* nodoIzquierdo;
};


Nodo* crearNodo(string contenido, int numero) {
    Nodo* nuevoNodo = new Nodo();
    nuevoNodo->contenido = contenido;
    nuevoNodo->numero = numero;
    nuevoNodo->nivel = 1;
    nuevoNodo->apuntando = false;
    nuevoNodo->primero = false;
    nuevoNodo->compañero = nullptr;
    nuevoNodo->nodoIzquierdo = nullptr;
    nuevoNodo->nodoDerecho = nullptr;
    return nuevoNodo;
}
```

Main

Este método es el inicial y es donde se encuentra el Menú Principal del programa y a partir de aquí es donde se comunicará con el resto del programa y como su único atributo importante está el del grupo de Conjuntos que inicia con un espacio de 5 pero que durante la ejecución del programa va ir aumentando y en si en dado caso. Por otro lado, en caso el usuario ingrese un valor numérico que no está dentro de los valores preestablecidos se mostrara un mensaje de error al usuario para que pueda rectificar su error.

```
int main() {
    Conjunto gruposActuales[5] = {Conjunto()};
    int size = 5, opcion = 0;
    bool salir = false;

    while(!salir) {
        cout << endl;
        cout << "Menu Principal" << endl;
        cout << "   1. Terminal del Lenguaje" << endl;
        cout << "   2. Menu de Reportes" << endl;
        cout << "   3. Graficas de las Estructuras" << endl;
        cout << "   4. Importacion de Contactos" << endl;
        cout << "   5. Salir" << endl;
        cin >> opcion;

        switch (opcion) {
            case 1: opcionesDeLaTerminal(gruposActuales, size);
                break;
            case 2: reportes(gruposActuales, size);
                break;
            case 3: cout << "Javier" << endl;
                break;
            case 4: exportarContactos(gruposActuales, size);
                break;
            case 5: salir = true;
                break;
            default:
                cout << endl;
                cout << "+==============================+" << endl;
                cout << "| Error, Numero Fuera de Rango |" << endl;
                cout << "+==============================+" << endl;
                break;
        }
    }
}
```

Opciones de la Terminal

Este método se invoca al momento de seleccionar la opción número 1 de nuestro Menú Principal y en este nuevo menú vamos a poder seleccionar la acción que necesitemos ejecutar si deseamos crear un nuevo grupo de datos debemos seleccionar crear, si queremos ingresar un grupo de datos a un Conjunto especifico seleccionamos insertar y si queremos buscar un dato especifico ponemos buscar y antes de invocar a cada método correspondiente para cada situación se le pedirá al usuario el comando correspondiente para la ejecución del método y de igual forma si se coloca un número incorrecto se le notificará al usuario para que corrija su error.

```
void opcionesDeLaTerminal(Conjunto gruposActuales[], int size) {
    bool salir = false;
    int seleccion = 0;
    string comando;

    while(!salir) {

        cout << endl;
        cout << "Terminal:" << endl;
        cout << "   1. Crear" << endl;
        cout << "   2. Insertar" << endl;
        cout << "   3. Buscar" << endl;
        cout << "   4. Atras" << endl;
        cin >> seleccion;

        switch (seleccion) {
            case 1:
                cout<<endl;
                cout<<"----------------------------------------"<<endl;
                cout<<"Ingrese comando para crear un Nuevo Grupo:"<<endl;
                cout<<"Escribir: "; getline(cin, comando); getline(cin, comando);
                cout<<"----------------------------------------"<<endl;
                crearGrupoDeContactos(comando, gruposActuales, size);
                salir = true;
                break;
            case 2:
                cout<<endl;
                cout<<"----------------------------------------"<<endl;
                cout<<"Ingrese comando para insertar Valores:"<<endl;
                cout<<"Escribir: "; getline(cin, comando); getline(cin, comando);
                cout<<"----------------------------------------"<<endl;
                insertarContactos(comando, gruposActuales, size);
                salir = true;
                break;
            case 3:
```

```
                cout<<endl;
                cout << "----------------------------------------" << endl;
                cout << "Ingrese comando de Busqueda:" << endl;
                cout << "Escribir: "; getline(cin,comando); getline(cin,comando);
                cout << "----------------------------------------" << endl;
                buscarContactos(comando, gruposActuales, size);
                salir = true;
                break;
            case 4: salir = true;
                break;
            default:
                cout << endl;
                cout << "+==============================+" << endl;
                cout << "| Error, Numero Fuera de Rango |" << endl;
                cout << "+==============================+" << endl;
                break;
        }
    }
}
```

Crear Grupo de Datos:

Este método es utilizado para crear un nuevo grupo de contactos y para esto unimos los caracteres del string del comando en uniones específicas para obtener el nombre del nuevo grupo, y los campos que se deben crear dentro de cada grupo, una cosa importante es que aquí es donde se les asigna a los campos el número de orden de cada uno que será utilizado para la búsqueda de información de algún contacto.

```
void crearGrupoDeContactos(string comando, Conjunto gruposActuales[], int size) {
    string grupo = obtenerGrupo(comando, 14), campo = "", tipo = "";
    int contador = 14 + grupo.size() + 9, valor = funcionHash(grupo, size),
indiceArreglo = 0, contador_2 = contador, orden = 0;

    while (contador_2 < comando.size()) {
        if (comando[contador_2] == ',') {
            indiceArreglo++;
        } else if (comando[contador_2] == ')') {
            indiceArreglo++;
            break;
        }
        contador_2++;
    }
    gruposActuales[valor].setDatos(grupo, indiceArreglo);

    while (contador < comando.size()) {
```

```
        if (comando[contador] != ' ') {
            campo += comando[contador];
            contador++;
        } else {
            contador++;

            for (int j = contador; j < comando.size(); j++) {
                if (comando[j] != ',' && comando[j] != ')') {
                    tipo += comando[j];
                } else {
                    contador = j;
                    break;
                }
            }
            if (comando[contador] == ',') {
                gruposActuales[valor].agregarCampo(campo, tipo, orden);
                campo = ""; tipo = "";
                contador += 2;
                orden++;
            } else {
                gruposActuales[valor].agregarCampo(campo, tipo, orden);
                break;
            }
        }
    }
}
```

Ingresar Contactos:

Este método es utilizado cuando se desea ingresar información de algún contacto específico para esto se le debe pedir al usuario que ingrese el comando de ingreso de datos en la sección del menú anterior e igual que la sección anterior se divide todo el string del comando agrupando los datos en un orden especifico con el fin de obtener el nombre del grupo y los datos que hay que ingresar en cada uno de los campos del grupo y por medio del orden del campo que fue especificado a la hora de crear el grupo.

```
void insertarContactos(string comando, Conjunto gruposActuales[], int size) {
    string grupo = obtenerGrupo(comando, 15), valor = "";
    int contador = (15 + grupo.size() + 9), ubicacion = funcionHash(grupo, size),
indice = 0, orden = 0, contadorValoresCopiados = 0;
    Campo* campos = gruposActuales[ubicacion].getCampos();
    int numeroDeCampos = gruposActuales[ubicacion].getSize();
    string* valoresIngresados = new string[numeroDeCampos];

    while (contador < (comando.size() - 2)) {

        if (comando[contador] != ',') {
            valor += comando[contador];
        } else {
            contador++;

            for (int i = 0; i < numeroDeCampos; i++) {
                if (campos[i].getOrden() == orden) {
                    indice = i;
                    orden++;
                    break;
                }
            }
            campos[indice].guardarValor(valor);
            valoresIngresados[contadorValoresCopiados] = valor;
            indice++; contadorValoresCopiados++;
            valor = "";
        }
        contador++;
    }

    for (int i = 0; i < numeroDeCampos; i++) {
        if (campos[i].getOrden() == orden) {
            indice = i;
            break;
        }
    }
```

```
    campos[indice].guardarValor(valor);
    valoresIngresados[contadorValoresCopiados] = valor;

    int limiteValores = campos[0].getSize();

    for (int i = 0; i < numeroDeCampos; i++) {
        NodoValor* valores_1 = NULL;
        NodoValor* valores_2 = NULL;
        int numeroDeIndice = 0;
        int ind_1 = i, ind_2 = i + 1;

        if (i == (numeroDeCampos - 1)) {
            ind_2 = 0;
        }

        while(true) {
            if (campos[numeroDeIndice].getOrden() == ind_1) {
                valores_1 = campos[numeroDeIndice].getValores();
                break;
            }
            numeroDeIndice++;
        }
        numeroDeIndice = 0;
        while(true) {
            if (campos[numeroDeIndice].getOrden() == ind_2) {
                valores_2 = campos[numeroDeIndice].getValores();
                break;
            }
            numeroDeIndice++;
        }

        int ubicacionValor_1 = funcionHash(valoresIngresados[ind_1],
limiteValores);
        int ubicacionValor_2 = funcionHash(valoresIngresados[ind_2],
limiteValores);
        valores_1[ubicacionValor_1].guardarDatoSiguiente(&valores_2[ubicacionValo
r_2]);
        valores_1[ubicacionValor_1].yaEstaApuntado();
        if (i == 0) {
            valores_1[ubicacionValor_1].esteEsElPrimero();
        }
    }
}
```

Buscar Contactos:

Este método es utilizado para obtener uno o varios contactos del programa dependiendo del grupo que se seleccione y para eso nuevamente se vuelve a solicitar un comando en la sección anterior y este string lo dividimos en partes para obtener la información que necesitamos y al terminar se muestran los resultados encontrados al usuario.

```
void buscarContactos(string comando, Conjunto gruposActuales[], int size) {
    string grupo = obtenerGrupo(comando, 16), campo = "", valor = "";
    int contador = 16 + grupo.size() + 15;

    while (contador < comando.size()) {
        if (comando[contador] != '=') {
            campo += comando[contador];
            contador++;
        } else {
            contador++;

            for (int i = contador; i < (comando.size() - 1); i++) {
                valor += comando[i];
            }
            break;
        }
    }

    int indiceGrupo = funcionHash(grupo, size);
    Campo* campos = gruposActuales[indiceGrupo].getCampos();
    int tamañoCampos = gruposActuales[indiceGrupo].getSize();
    int indiceCampo = funcionHash(campo, tamañoCampos);
    NodoValor* valores = campos[indiceCampo].getValores();
    int limiteValores = campos[indiceCampo].getSize();

    cout << endl;
    cout << "-----------------------------------" << endl;
    cout << "Contacto(s):" << endl;
    cout << endl;

    for (int i = 0; i < limiteValores; i++) {
        if (valores[i].getContenido() == valor) {
            NodoValor *nValor = &valores[i];
            int lim = 0;

            while (true) {
                if (nValor->isPrimero()) {
                    break;
```

```
                } else {
                    nValor = nValor->getNodoSiguiente();
                }
            }

            cout << "  -";
            while (lim < tamañoCampos) {
                if (lim == (tamañoCampos - 1)) {
                    cout << " " << nValor->getContenido();
                } else {
                    cout << " " << nValor->getContenido()<< ", ";
                }
                nValor = nValor->getNodoSiguiente();
                lim++;
            }
            cout << endl;
        }
    }
    cout << "-----------------------------------" << endl;
}
```

Arbol AVL

Este concepto se centra en un árbol binario donde su objetivo es balancear cada lado del nodo de forma equivalente evitando que de un lado tenga más peso que del otro y para esto se hace uso de algunos métodos, además de métodos auxiliares que veremos a continuación:

Izquierda-Izquierda:

Este método es utilizado cuando tenemos un desbalanceo del lado derecho de nuestra rama es decir se tiene 3 valores y todos van a la derecha el uno del otro por lo tanto solo crece en su rama derecha y para solucionarlo debemos mandar el nodo actual y por medio de un nodo temporal hacer que el ultimo nodo apunte a su anterior y luego hacer que ese nodo nuevo apunte al que se mandó y con eso solucionaríamos el desbalance.

```
Nodo* rotacionDobleIzquierda (Nodo* actual) {
    Nodo* temporal = actual->nodoDerecho;
    actual->nodoDerecho = temporal->nodoIzquierdo;
    temporal->nodoIzquierdo = actual;

    actual->nivel = (max(obtenerNivel(actual->nodoDerecho), obtenerNivel(actual-
>nodoIzquierdo)) + 1);
    temporal->nivel = (max(obtenerNivel(temporal->nodoDerecho),
obtenerNivel(temporal->nodoIzquierdo)) + 1);
    return temporal;
}
```

Derecha-Derecha:

Este método es utilizado cuando tenemos un desbalanceo del lado izquierdo de nuestra rama es decir se tiene 3 valores y todos van a la izquierda el uno del otro por lo tanto solo crece en su rama izquierda solucionarlo debemos mandar el nodo actual y por medio de un nodo temporal hacer que el ultimo nodo apunte a su anterior y luego hacer que ese nodo nuevo apunte al que se mandó y con eso solucionaríamos el desbalance.

```
Nodo* rotacionDobleDerecha(Nodo* actual) {
    Nodo* temporal = actual->nodoIzquierdo;
    actual->nodoIzquierdo = temporal->nodoDerecho;
    temporal->nodoDerecho = actual;

    actual->nivel = (max(obtenerNivel(actual->nodoDerecho), obtenerNivel(actual-
>nodoIzquierdo)) + 1);
```

```
    temporal->nivel = (max(obtenerNivel(temporal->nodoDerecho),
obtenerNivel(temporal->nodoIzquierdo)) + 1);
    return temporal;
}
```

Izquierda-Derecha:

Este método es utilizado de manera diferente ya que existe un desbalanceo, pero en este caso no se da de manera progresiva, es decir no crece solo un lado de los nodos, sino que el ultimo nodo a ingresar se ubica del lado contrario de donde está el último nodo por lo tanto no se puede utilizar los mismos métodos anteriores ya que primero debemos convertir este caso en uno de los dos anteriores para poder utilizar los métodos Izquierda-Izquierda o Derecha-Derecha.

```
Nodo* rotacionDerecha(Nodo* actual) {
    Nodo* temporal;
    actual->nodoIzquierdo = rotacionDerecha(actual->nodoIzquierdo);
    temporal = rotacionIzquierda(actual);
    return temporal;
}
```

Derecha-Izquierda:

Este método es utilizado de manera diferente ya que existe un desbalanceo, pero en este caso no se da de manera progresiva, es decir no crece solo un lado de los nodos, sino que el ultimo nodo a ingresar se ubica del lado contrario de donde está el último nodo por lo tanto no se puede utilizar los mismos métodos anteriores ya que primero debemos convertir este caso en uno de los dos anteriores para poder utilizar los métodos Izquierda-Izquierda o Derecha-Derecha.

```
Nodo* rotacionIzquierda(Nodo* actual) {
    Nodo* temporal;
    actual->nodoDerecho = rotacionIzquierda(actual->nodoDerecho);
    temporal = rotacionDerecha(actual);
    return temporal;
}
```

Buscar Lugar:

Este método es utilizado cuando ingresamos un nodo nuevo al árbol, pero primero verificamos a qué lado del nodo actual debemos ingresarlo y luego de esto se verifica si este el ultimo nodo del arreglo, en caso no sea el último se utiliza la recursividad para volver a llamar el método pero ahora pasándole de parámetro el nodo hijo del nodo actual, pero si en caso fuera el ultimo procedemos a ingresar el nodo, por ultimo verificamos los niveles de cada nodo y se hubiera algún error se invoca a los métodos correspondientes para poder solucionarlo.

```
Nodo* buscarLugar(Nodo* actual, Nodo* nuevo) {
    if (nuevo->numero < actual->numero) {
        if (actual->nodoIzquierdo == nullptr) {
            actual->nodoIzquierdo = nuevo;
        } else {
            actual->nodoIzquierdo = buscarLugar(actual->nodoIzquierdo, nuevo);

            if ((obtenerNivel(actual->nodoIzquierdo) - obtenerNivel(actual-
>nodoDerecho)) == 2) {
                if (nuevo->numero < actual->nodoIzquierdo->numero) {
                    actual = rotacionIzquierda(actual);
                } else {
                    actual = rotacionDobleIzquierda(actual);
                }
            }
        }
    } else {
        if (actual->nodoDerecho == nullptr) {
            actual->nodoDerecho = nuevo;
        } else {
            actual->nodoDerecho = buscarLugar(actual->nodoDerecho, nuevo);

            if ((obtenerNivel(actual->nodoDerecho) - obtenerNivel(actual-
>nodoIzquierdo)) == 2) {
                if (nuevo->numero > actual->nodoDerecho->numero) {
                    actual = rotacionDerecha(actual);
                } else {
                    actual = rotacionDobleDerecha(actual);
                }
            }
        }
    }
    if (actual->nodoIzquierdo == nullptr && actual->nodoDerecho != nullptr) {
        actual->nivel = actual->nodoDerecho->nivel + 1;
    } else if (actual->nodoDerecho == nullptr && actual->nodoIzquierdo !=
nullptr) {
        actual->nivel = actual->nodoIzquierdo->nivel + 1;
    } else {
        actual->nivel = (max(obtenerNivel(actual->nodoDerecho),
obtenerNivel(actual->nodoIzquierdo)) + 1);
    }
    return actual;
}
```

Ingresar Dato:

Este método es utilizado cuando se quiere ingresar un nuevo nodo al árbol y para ello hace una pequeña revisión antes de llamar al método anterior, si en caso el nodo inicial sea null, se inicializa y se termina la invocación, en caso ya contrario se crear el nodo nuevo y se manda a llamar al método buscar lugar.

```
Nodo* ingresarDato(Nodo* inicial, int numero) {
    if (inicial == nullptr) {
        inicial = crearNodo(numero);
    } else {
        Nodo *nuevo = crearNodo(numero);
        inicial = buscarLugar(inicial, nuevo);
    }
    return inicial;
}
```

Crear Nodo:

Este método es invocado por el método anterior y su función es solo la de crear un nuevo nodo e inicializar cada uno de sus valores.

```
Nodo* crearNodo(int numero) {
    Nodo* nuevoNodo = new Nodo();
    nuevoNodo->numero = numero;
    nuevoNodo->nivel = 1;
    nuevoNodo->nodoIzquierdo = nullptr;
    nuevoNodo->nodoDerecho = nullptr;
    return nuevoNodo;
}
```

Obtener Nivel:

Este método es utilizado para devolver el nivel del nodo, si en caso no haya nodo se procede a devolver cero.

```
int obtenerNivel(Nodo* actual) {
    if (actual == nullptr) {
        return 0;
    }
    return actual->nivel;
}
```

Función Hash

Este método es utilizado para calcular un índice entre los límites del arreglo para guardar la información ya sea un Conjunto, un Campo o un NodoValor de modo de que al momento de querer utilizar un cierto grupo con solo mandar el nombre del grupo a visualizar seamos capaces por medio de la función Hash de obtener el número que representa ese valor y esto lo hacemos por medio del código Ascii que suma todas las letras del nombre y por medio de un mod obtenemos el valor del índice que se ubica dentro de las dimensiones del arreglo, y al hacerlo dinámico nos garantiza que no tengamos que estar copiando y pegando este método en todo el código.

Como parámetro se le envía el nombre del grupo y el tamaño del arreglo el cual se quiere utilizar y lo que devuelve es la dirección del elemento en el arreglo.

```
int funcionHash(string grupo, int size) {
    int numerohash = 0;

    int sumaLetras = 0;
    for (int i = 0; i < grupo.size(); i++) {
        sumaLetras += grupo[i];
    }
    sumaLetras = sumaLetras % size;
    return sumaLetras;
}
```

Obtener Nombre Del Grupo

Este método surgió de recurrencia en los métodos de crear, insertar y buscar que tenían que buscar el nombre del grupo para poder realizar actividades en él y por eso se determinó que separar este procedimiento era beneficioso ya que al momento de encontrar el nombre del grupo a utilizar devuelve un string con el nombre del grupo para su posterior uso.

```
string obtenerGrupo(string comando, int limite) {
    string grupo = "";

    for (int i = limite; i < comando.size(); i++) {
        if (comando[i] != ' ') {
            grupo += comando[i];
        } else { break; }
    }
    return grupo;
}
```

Reportes

Este método es utilizado para mostrarle al usuario un reporte del estado actual de los grupos mostrándole al usuario los siguientes datos:

- Cantidad de Datos por Grupo, es decir la cantidad de datos en cada campo hay en cada grupo del sistema
- Cantidad Total del Sistema, es decir todos los datos que se encuentren en cada uno de los Conjuntos
- Cantidad de Contactos del mismo Tipo, es decir el número de contactos de todos los grupos que utilicen una o más de los tipos permitidos los cuales son (STRING, INTEGER. CHAR, DATE).
- Por último la cantidad de Contactos por Grupos que es el número de contactos que se han guardado en cada grupo.

```
void reportes(Conjunto gruposActuales[], int size) {
    int* cantidadDeDatosPorGrupo = new int[size];
    int* cantidadDeContactosPorGrupo = new int[size];
    int contadorValores = 0, indice = 0, numeroDeString = 0, numeroDeInteger = 0,
numeroDeChar = 0, numeroDeDate = 0;

    for (int i = 0; i < size; i++) {
        cantidadDeDatosPorGrupo[i] = 0;
        cantidadDeContactosPorGrupo[i] = 0;
    }

    cout << endl;
    cout << "+-----------------REPORTES-----------------+" << endl;
    cout << " Cantidad de Datos por Grupo:" << endl;
    cout << endl;

    for (int i = 0; i < size; i++) {
        if (gruposActuales[i].getNombre() != "") {
            //Grupo
            string nombreGrupo = gruposActuales[i].getNombre();
            int numeroDeCampos = gruposActuales[i].getSize();
            Campo* campos = gruposActuales[i].getCampos();

            //Campo
            NodoValor* valores = campos[0].getValores();
            int numeroDeValores = campos[0].getSize();

            for (int j = 0; j < numeroDeValores; j++) {
                if (valores[j].getContenido() != "") {
                    contadorValores++;
```

```
                }
            }
            int totalGrupo = contadorValores * numeroDeCampos;
            cantidadDeDatosPorGrupo[indice] = totalGrupo;
            cantidadDeContactosPorGrupo[i] = contadorValores;
            cout << " - "<< nombreGrupo << ": " << totalGrupo << " dato(s)"<<
endl;
            contadorValores = 0;
            indice++;
        }
    }
    cout << "+----------------------------------------+" << endl;
    int totalSistema = 0;

    for (int i = 0; i < size; i++) {
        totalSistema += cantidadDeDatosPorGrupo[i];
    }
    cout << " Todos de Datos del Sistema: " << endl;
    cout << endl;
    cout << " - Total: " << totalSistema << " dato(s)" << endl;
    cout << "+----------------------------------------+" << endl;
    cout << " Cantidad de Contactos con el mismo Tipo:" << endl;
    cout << endl;
    for (int i = 0; i < size; i++) {
        if (gruposActuales[i].getNombre() != "") {
            bool stringRegistrado = false, integerRegistrado = false,
charRegistrado = false, dateRegistrado = false;
            int numeroDeCampos = gruposActuales[i].getSize();
            Campo* campos = gruposActuales[i].getCampos();

            for (int j = 0; j < numeroDeCampos; j++) {
                if (campos[j].getTipo() == "STRING") {
                    if (!stringRegistrado) {
                        stringRegistrado = true;
                    }
                } else if (campos[j].getTipo() == "INTEGER") {
                    if (!integerRegistrado) {
                        integerRegistrado = true;
                    }
                } else if (campos[j].getTipo() == "CHAR") {
                    if (!charRegistrado) {
                        charRegistrado = true;
                    }
                } else if (campos[j].getTipo() == "DATE") {
                    if (!dateRegistrado) {
```

```
                        dateRegistrado = true;
                    }
                }
            }
            int contador = 0;
            NodoValor* valores = campos[0].getValores();
            int numeroDeDatos = campos[0].getSize();

            for (int j = 0; j < numeroDeDatos; j++) {
                if (valores[j].getContenido() != "") {
                    contador++;
                }
            }

            if (stringRegistrado) {
                numeroDeString += contador;
            }
            if (integerRegistrado) {
                numeroDeInteger += contador;
            }
            if (charRegistrado) {
                numeroDeChar += contador;
            }
            if (dateRegistrado) {
                numeroDeDate += contador;
            }
        }
    }

    cout << " - STRING: " << numeroDeString << " contacto(s)" << endl;
    cout << " - INTEGER: " << numeroDeInteger << " contacto(s)" << endl;
    cout << " - CHAR: " << numeroDeChar << " contacto(s)" << endl;
    cout << " - DATE: " << numeroDeDate << " contacto(s)" << endl;
    cout << "+----------------------------------------+" << endl;
    cout << " Cantidad de Contactos por Grupo:" << endl;
    cout << endl;
    for (int i = 0; i < size; i++) {
        if (gruposActuales[i].getNombre() != "") {
            string nombreGrupo = gruposActuales[i].getNombre();
            cout << " - "<< nombreGrupo << ": " << cantidadDeContactosPorGrupo[i]
<< " contacto(s)"<< endl;
        }
    }
    cout << "+----------------------------------------+" << endl;
}
```

Exportar Datos

Este método es utilizado para poder en exportar datos del sistema por medio de directorios y archivo txt, para esto se le pedirá al usuario que ingres el nombre del grupo que quiere exportar y luego por medio de un análisis se busca al grupo correspondiente y se procede a realizar ciclos para obtener toda la información necesaria para la crear del directorio y los archivos los cuales adentro tendrán la información total del contacto.

```
void exportarContactos(Conjunto gruposActuales[], int size) {
    string nombreDeLaCarpeta = "";
    cout << endl;
    cout << "-------------------------------------------------" << endl;
    cout << "Escriba el Nombre del Grupo a Exportar: "; cin >> nombreDeLaCarpeta;
    cout << "-------------------------------------------------" << endl;
    cout << endl;
    int contador = 0;
    bool grupoCorrecto = false;

    while (contador < size) {
        if (gruposActuales[contador].getNombre() == nombreDeLaCarpeta) {
            grupoCorrecto = true;
            break;
        }
        contador++;
    }

    if (grupoCorrecto) {
        if (mkdir(nombreDeLaCarpeta.c_str()) == 0) {
            int indiceConjunto = funcionHash(nombreDeLaCarpeta, size);
            Campo* campos = gruposActuales[indiceConjunto].getCampos();
            int limiteCampos = gruposActuales[indiceConjunto].getSize();
            NodoValor* valores = campos[0].getValores();
            int limiteValores = campos[0].getSize();

            for (int i = 0; i < limiteValores; i++) {
                NodoValor *valor = &valores[i];
                string contacto = "", nombreArchivo = "";
                contador = 0;

                if (valor->getContenido() != "") {
                    while (true) {
                        if (valor->isPrimero()) {
                            break;
                        } else {
                            valor = valor->getNodoSiguiente();
```

```
                        }
                    }

                    while (contador < limiteCampos) {
                        if (contador == (limiteCampos - 1)) {
                            contacto += valor->getContenido();
                        } else {
                            contacto +=  valor->getContenido() + ", ";

                            if (contador == 0) {
                                nombreArchivo = valor->getContenido();
                            }
                        }
                        valor = valor->getNodoSiguiente();
                        contador++;
                    }

                    ofstream archivo;
                    archivo.open(nombreDeLaCarpeta + "\\" + nombreArchivo
+".txt", ios::out);

                    if (!archivo.fail()) {
                        archivo << contacto << endl;
                    }
                }
            }
        } else {
            cout << "+======================================+" << endl;
            cout << "| Error, no se pudo crear el Directorio |" << endl;
            cout << "+======================================+" << endl;
        }
    } else {
        cout << endl;
        cout << "+==============================================+" << endl;
        cout << "| Error, no se encontro un grupo con ese Nombre |"; cin >>
nombreDeLaCarpeta;
        cout << "+==============================================+" << endl;
        cout << endl;
    }
}
```

Graficar

Este método es utilizado para graficar cada toda la estructura completa de nuestro programa y lo hace por medio de dos ciclos y 3 variables auxiliares donde introduciremos las instrucciones por separado para al final unir todo en un mismo string pero siempre verificando que los nodos iniciales no sean nulos.

```
void graficarTodo(Conjunto gruposActuales[], int size) {
    string referencias = "";
    string campoDescripcion = "";
    string valores = "";
    int campoNumero = 0;
    int contadorNodo = 0;
    int indice = 0;

    string code = "digraph G {\n"
    "    node [shape=circle, style=filled, fillcolor=lightblue,
fontcolor=black];\n"
    "    graph [rankdir = \"LR\"];\n"
    "    \n";

    for (int i = 0; i < size; i++) {
        if (gruposActuales[i].getNombre() != "") {
            Campo* campos = gruposActuales[i].getCampos(); //
            int numeroDeCampos = gruposActuales[i].getSize(); //

            if (indice == 0) {
                code += "    node" + to_string(contadorNodo) + " [label = \"<f" +
to_string(indice) + "> " + gruposActuales[i].getNombre();
                contadorNodo++;
            } else  {
                code += "| <f" + to_string(indice) + "> " +
gruposActuales[i].getNombre();
            }
            indice++;

            if (i == (size - 1)) {
                code += " \" shape = \"record\"];\n";
            }

            for (int j = 0; j < numeroDeCampos; j++) {
                Nodo* valorInicial = campos[j].getNodoInicial();
                //int numeroDeDatos = campos[j].getNumeroDeDatos();

                if (j == 0) {
```

```
                    campoDescripcion += "    node" + to_string(contadorNodo) + "
[label = \"<f" + to_string(j) + "> " + campos[j].getNombre();

                    for (int k = 0; k < numeroDeCampos; k++) {
                        referencias += "    \"node0\":f" + to_string(indice - 1)
+ " -> \"node" + to_string(contadorNodo) + "\":f" + to_string(k) + ";\n";
                    }
                    campoNumero = contadorNodo;
                    contadorNodo++;
                } else {
                    campoDescripcion += "| <f" + to_string(j) + "> " +
campos[j].getNombre();
                }

                if (valorInicial != nullptr) {
                    valores += "    " + to_string(contadorNodo) + " [label=\"" +
valorInicial->contenido + "\"];\n";
                    referencias += "    \"node" + to_string(campoNumero) +
"\":f"  + to_string(j) + " -> " + to_string(contadorNodo) + ";\n";
                    int numeroReferencia = contadorNodo;
                    contadorNodo++;

                    if (valorInicial->nodoIzquierdo != nullptr) {
                        contadorNodo = graficarNodos(valorInicial->nodoIzquierdo,
valores, referencias, numeroReferencia, contadorNodo);
                    }
                    if (valorInicial->nodoDerecho != nullptr) {
                        contadorNodo = graficarNodos(valorInicial->nodoDerecho,
valores, referencias, numeroReferencia, contadorNodo);
                    }
                }
            }
            campoDescripcion += " \" shape = \"record\"];\n";
        }
    }
    code += campoDescripcion;
    code +=  "\n";
    code += valores;
    code +=  "\n";
    code += referencias;
    code += "}";

    ofstream file("gra.dot");
    if (file.is_open()) {
        file << code << endl;
```

```
        file.close();
        system("dot -Tpng gra.dot -o graf.png");
    }
}
```

Graficar Nodos

Este método es utilizado de manera recursiva para poder graficar cada uno de los nodos de manera que podamos mostrarlos todos.

```
int graficarNodos(Nodo* actual, string& valores, string& referencias, int
numeroDeReferencia, int contadorNodo) {
    valores += "    " + to_string(contadorNodo) + " [label=\"" + actual-
>contenido + "\"];\n";
    referencias += "    " + to_string(numeroDeReferencia) + " -> " +
to_string(contadorNodo) + ";\n";
    numeroDeReferencia = contadorNodo;
    contadorNodo++;

    if (actual->nodoIzquierdo != nullptr) {
        contadorNodo = graficarNodos(actual->nodoIzquierdo, valores, referencias,
numeroDeReferencia, contadorNodo);
    }
    if (actual->nodoDerecho != nullptr) {
        contadorNodo = graficarNodos(actual->nodoDerecho, valores, referencias,
numeroDeReferencia, contadorNodo);
    }
    return contadorNodo;
}
```

ReHashing

Este método es utilizado para poder aumentar la capacidad de nuestro arreglo principal de conjuntos, esto con el fin de que no se tenga que tener mucho espacio en memoria y que lo que se utilice esa lo óptimo para el buen funcionamiento del sistema.

```
Conjunto* verificarReHashing(Conjunto gruposActuales[], int& size) {
    int contador = 0;

    for (int i = 0; i < size; i++) {
        if (gruposActuales[i].getNombre() != "") {
            contador++;
```

```
        }
    }

    double porcentaje = (double) contador/size;
    cout << porcentaje << endl;

    if (porcentaje >= 0.6) {
        for (int i = 0; i < size; i++) {
            if (gruposActuales[i].getNombre() != "") {
                cout << ".. " << i << endl;
            }
        }
        cout << "--------------------" << endl;
        Conjunto *temporal = new Conjunto[contador];

        for (int i = 0; i < contador; i++) {
            if (gruposActuales[i].getNombre() != "") {
                temporal[i] = gruposActuales[i];
            }
        }
        size = size + 3;
        gruposActuales = new Conjunto[size];

        for (int i = 0; i < contador; i++) {
            int indice = funcionHash(temporal[i].getNombre(), size);
            gruposActuales[indice] = temporal[i];
        }

        for (int i = 0; i < size; i++) {
            if (gruposActuales[i].getNombre() != "") {
                cout << ".. " << i << endl;
            }
        }
    }
    return gruposActuales;
}
```

Archivo Log

Este método es utilizado para llevar un control acerca de las actividades que se realizan dentro del código, desde el ingreso de un nuevo grupo de datos, el ingreso de contactos al sistema, búsquedas, visualización de reportes, visualización de las gráficas, entre otras si escribe

dentro de este archivo para siempre tener un control sobre las actividades que se están realizando.

```
void escribirArchivoLog(string accion) {
    ofstream archivoEscribir;

    ifstream archivoLeer;
    string texto;

    time_t tiempo;
    tiempo = time(NULL);
    struct tm *fecha;
    fecha = localtime(&tiempo);

    string fechaAMostrar = "    " + to_string(fecha->tm_hour) + ":" +
to_string(fecha->tm_min) + " " + to_string(fecha->tm_mday) + "/" +
to_string(fecha->tm_mon + 1) + "/" + to_string(fecha->tm_year + 1900);
    archivoLeer.open("Sistema.log", ios::in);

    if (!archivoLeer.fail()) {
        string textoTotal = accion + fechaAMostrar + "+";

        while(!archivoLeer.eof()) {
            getline(archivoLeer, texto);
            textoTotal += (texto + "+");
        }
        archivoLeer.close();
        remove("Sistema.log");

        archivoEscribir.open("Sistema.log", ios::out);

        if (!archivoEscribir.fail()) {
            string textoAEscribir = "";

            for (int i = 0; i < textoTotal.size(); i++) {
                if (textoTotal[i] != '+') {
                    textoAEscribir += textoTotal[i];
                } else {
                    archivoEscribir << textoAEscribir << endl;
                    textoAEscribir = "";
                }
            }
        }
    } else {
```

```
        archivoEscribir.open("Sistema.log", ios::out);
        accion = accion + fechaAMostrar;

        if (!archivoEscribir.fail()) {
            archivoEscribir << accion << endl;
        }
    }
}
```