

# Manual 4 - 2do Torneo de Programación Competitiva

Lions R.C.

Julio 2019

## Contents

<b>1</b>	<b>Arreglos multidimensionales</b>	<b>2</b>
<b>2</b>	<b>Structs</b>	<b>2</b>
<b>3</b>	<b>Grafos</b>	<b>5</b>
3.1	Nodos, ramas, hojas y raíces . . . . .	5
3.2	Grafos dirigidos y cíclicos . . . . .	7
3.3	Grafos con pesos . . . . .	7
3.4	Creando grafos . . . . .	8
3.5	Arboles binarios . . . . .	8
3.6	Aplicaciones . . . . .	8
<b>4</b>	<b>Algoritmos de busqueda</b>	<b>8</b>
4.1	Busqueda en anchura . . . . .	8
4.2	Busqueda en profundidad . . . . .	8
4.3	Algoritmo de Dijkstra . . . . .	8
4.4	Algoritmos heuristicos . . . . .	8
4.5	A* . . . . .	8



## 1 Arreglos multidimensionales

A veces es conveniente manejar datos como si fueran a estar en una matriz de más de una dimensión, así que C++ te permite crear arreglos multidimensionales para facilitar este proceso. Casi nunca se requieren más de tres dimensiones para resolver un problema así que el usuario debe definir previamente cuantas dimensiones tiene su arreglo, además la memoria que se requiere para el arreglo incrementa exponencialmente con cada dimensión.

Para definir un arreglo de dimensión N, se debe escribir el tipo de dato, el nombre del arreglo y N corchetes []. Si queremos un arreglo de 7 x 3 x 3 enteros, podemos definirlo con `int miArreglo[7][3][3]`; También se pueden definir los datos iniciales de este arreglo utilizando múltiples llaves anidadas:

Listing 1: Asignando valores

```
#include <iostream>

using namespace std;

int main() {
    int cuboide[2][3][3] = {{{1, 2, 3}, {4, 5, 6}, {7, 8, 9}},
                             {{10, 11, 12}, {13, 14, 15}, {16, 17, 18}}};
}
```

## 2 Structs

A veces es frustrante tener que manejar grupos de datos que deben ir juntos debido a que se tienen que crear pares de pares o múltiples arreglos. Esto se puede solucionar con los structs, que son parte de la programación orientado a objetos.

Los structs son estructuras que un usuario puede definir para guardar múltiples variables bajo un solo "objeto".

Por ejemplo, digamos que trabajas para un banco y quisieras guardar los datos importantes de tus clientes: su nombre, su apellido, su número de tarjeta y la cantidad de dinero que tiene. Si quisieramos guardar estos valores convencionalmente, tendríamos que usar cuatro arreglos o cuatro pares de pares anidados.

Usando structs, podemos definir un struct por cada cliente con estos tipos de datos y crear un solo arreglo o vector de clientes. No se requiere ninguna librería para definir un struct y se puede crear de la siguiente manera:

Listing 2: Definición de un struct

```
#include <iostream>

using namespace std;
```

```

struct Cliente {
    string nombre;
    string apellido;
    int tarjeta[16];
    float dinero;
};

int main() {
}

```

Como se puede ver, los structs siempre deben ir antes de nuestra función `main` y deben tener un punto y coma despues de su llave de cierre. Luego dentro de las llaves debe tener una lista de todas las variables que se desean agrupar.

Para crear una instancia de un struct, se debe poner el nombre del struct como el tipo de dato seguido por el nombre especifico de esa instancia:

Listing 3: Instanciamiento

```

#include <iostream>

using namespace std;

struct Cliente {
    string nombre;
    string apellido;
    int tarjeta[16];
    float dinero;
};

int main() {
    Cliente jorge;
    Cliente pablo;
}

```

Como se puede observar, se crearon dos clientes, **jorge** y **pablo**. Podemos modificar sus datos escribiendo el nombre de cada variable despues de un punto:

Listing 4: Modificando valores

```

#include <iostream>

using namespace std;

struct Cliente {
    string nombre;
    string apellido;
    int tarjeta[16];
}

```

```

        float dinero;
    };

    int main() {
        Cliente jorge;
        jorge.nombre = "Jorge";
        jorge.apellido = "Velazquez";
        jorge.tarjeta = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5};
        jorge.dinero = 50726.35;
        Cliente pablo;
        pablo.nombre = "Pablo";
        pablo.apellido = "Cesar";
        pablo.tarjeta = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7, 9, 2};
        pablo.dinero = 999999999.9999;
    }

```

Para simplificar este proceso, es más fácil guardar las estructuras en un arreglo o vector:

Listing 5: Clientes bancarios

```

#include <iostream>
#include <vector>

using namespace std;

struct Cliente {
    string nombre;
    string apellido;
    int tarjeta[16];
    float dinero;
};

int main() {
    int numeroDeClientes = 3;
    vector<Cliente> clientes;
    for(int i = 0; i < numeroDeClientes; i++) {
        Cliente nuevo;
        cout << "Nombre_del_cliente:_" << endl;
        cin >> nuevo.nombre;
        cout << "Apellido_del_cliente:_" << endl;
        cin >> nuevo.apellido;
        string tarjeta;
        cout << "Tarjeta_del_cliente:_" << endl;
        cin >> tarjeta;
        for(int i = 0; i < 16; i++) {
            nuevo.tarjeta[i] = tarjeta[i] - '0';
        }
    }
}

```

```

    }
    cout << "Dinero:_" << endl;
    cin >> tarjeta;
    clientes.push_back(nuevo);
    cout << "Cliente_" << nuevo.nombre << "_guardado_con_exito" << endl;
}
cout << clientes.size() << "_clientes_guardados" << endl;
for(int i = 0; i < clientes.size(); i++) {
    cout << clientes[i].nombre << endl;
}
}

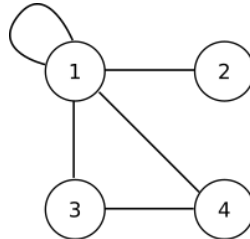
```

[Liga al código](#)

El último código guarda 5 clientes en un vector y pide sus datos al usuario. Después, se imprimen los nombres de estos clientes.

### 3 Grafos

Un conjunto de datos con relaciones entre otros datos se puede decir que es un grafo. Cada grafo debe de poder ser dibujado en un plano con los datos encerrados entre círculos y con líneas entre estos datos.



En esta imagen, hay cuatro elementos unidos a si mismos. Podemos ver que el 1 tiene enlaces con 1, 2, 3 y 4, el 2 solo tiene un enlace con 1, el 3 tiene enlace con 1 y 4 y el 4 tiene enlace con 1 y 3.

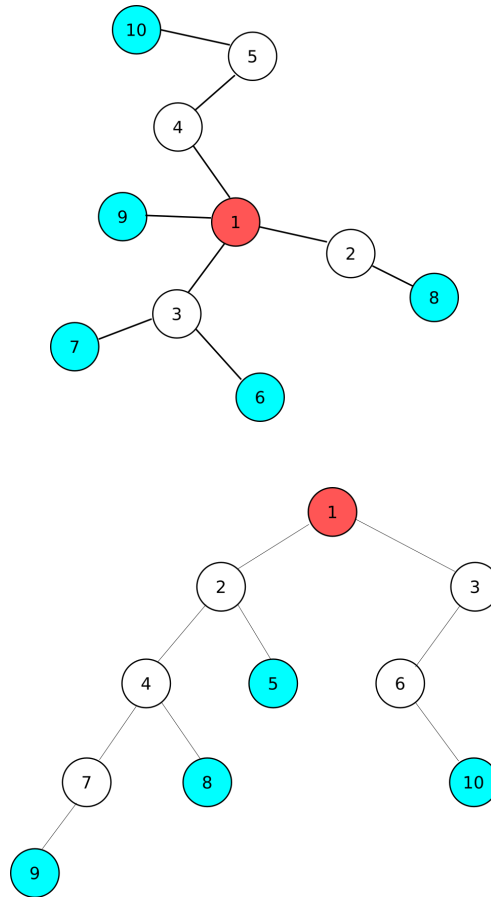
Cada uno de estos elementos puede representar un número, un caracter, un punto en 3D o cualquier cosa que desees que representen, mientras que cada enlace puede tener un significado importante de ese elemento.

Es importante definir que cada enlace debe consistir en la unión de dos elementos, y estos elementos pueden ser el mismo (por ejemplo el enlace que esta unido al 1 dos veces).

#### 3.1 Nodos, ramas, hojas y raíces

Se le conoce como nodo o vertice a cada elemento del grafo y se le conoce como rama, enlace o arista a cada enlace. En el ejemplo de arriba, podemos ver que existen cuatro nodos (1, 2, 3, 4) y cinco ramas (1:1, 1:2, 1:3, 1:4, 3:4).

En casos de ciertos grafos, es conveniente pensar que ciertos nodos son hojas o raíces. Abajo hay dos ejemplos de grafos que presentan estos nodos con las hojas marcadas en azul y la raíz marcada en rojo



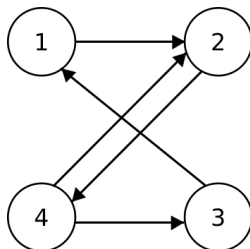
Como se puede observar en ambos grafos, existe una raíz o nodo con la mayor cantidad de uniones y que es céntrico a todos los demás nodos, y existen varias hojas que se pueden considerar como nodos que estan en la orilla.

Cada nodo se puede considerar como "hijo" de otro nodo excepto la raíz, y cada nodo se puede considerar como "padre" de otro nodo excepto las hojas. En el primer grafo con la raíz y las hojas señaladas, se puede decir que 2, 3, 4 y 9 son hijos de 1 y que 1 es padre de 2, 3, 4 y 9.

Esta relación de padre y hijo es útil será útil después para optimizar operaciones relacionados con grafos.

### 3.2 Grafos dirigidos y cíclicos

Hasta ahorita hemos visto grafos no dirigidos, pero también existen grafos dirigidos que tienen ramas de un solo sentido:



Podemos ver que para llegar desde el nodo 1 al nodo 3 se tiene que pasar por los nodos 2 y 4 porque no hay un camino orientado hacia el nodo 3.

Un ejemplo de un grafo dirigido puede ser el mapa de todas las calles de un pueblo. En el pueblo, puede haber calles de doble sentido o calles de un solo sentido, y se puede representar cada calle como una rama.

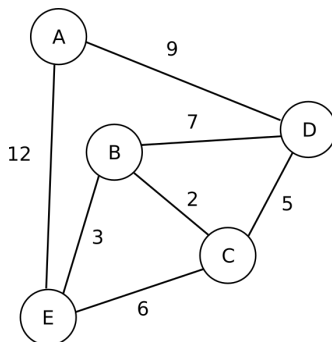
Otra propiedad de los grafos ocurre cuando un grafo contiene un ciclo (es decir que puedes llegar a un mismo nodo pasando por ramas distintas en cada salto), entonces ese grafo puede ser considerado como cíclico.

Se puede observar que los dos grafos de la sección **Nodos, ramas, hojas y raíces** son acíclicos mientras que los otros dos son cíclicos.

### 3.3 Grafos con pesos

Muchas veces es conveniente darle pesos a las ramas de algún grafo para modificar la manera en la que se distribuyen los nodos. Digamos que quieres representar un país con N ciudades o nodos y quisieras saber cual es la mejor ruta de una ciudad a otra.

Para resolver este problema se puede considerar cada rama como una carretera de una ciudad a otra y se le puede poner un peso con la distancia real de esa carretera.



Como se puede observar en el grafo de arriba, hay 5 ciudades (A, B, C, D, E) y varias carreteras con ciertas distancias (en este caso no nos importan las unidades).

Existen muchas posibles maneras de irse de la ciudad E y llegar a la ciudad D, pero solo hay un camino más óptimo que los demás. Por ejemplo, podemos tomar el camino E - A - D, pero la suma de las distancias de cada carretera es de  $12 + 9$  o 21. La mejor opción es el camino de E - B - C - D con una suma total de  $3 + 2 + 5 = 10$ . Se puede ver que a pesar de que se visitaron más nodos se recurrió menos distancia.

La siguiente sección cubre maneras de poder encontrar este camino más óptimo dado cualquier grafo.

### **3.4 Creando grafos**

### **3.5 Árboles binarios**

### **3.6 Aplicaciones**

## **4 Algoritmos de búsqueda**

### **4.1 Búsqueda en anchura**

### **4.2 Búsqueda en profundidad**

### **4.3 Algoritmo de Dijkstra**

### **4.4 Algoritmos heurísticos**

### **4.5 A\***