

Structs

UNRN

Universidad Nacional
de Río Negro



Argumentos por línea de comandos

```
int main(int argc, char*argv[])
```

Siempre contiene el nombre del archivo

`argv[0]`



cantidad

```
int main(int argc, char *argv[])
```



argumentos

Y el resto hasta argv, los argumentos

Para un invocación como

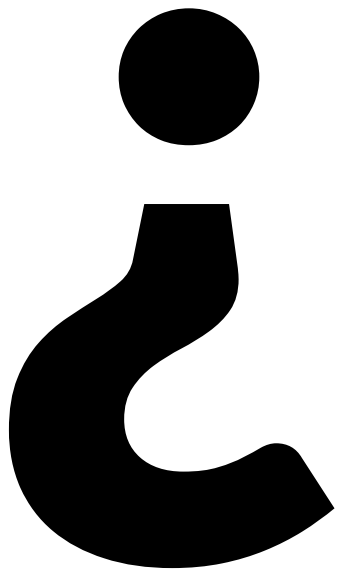
```
./a.exe hola mundo args
```

argc	3
-------------	---

argv	0	→	a	.	e	x	e	0
	1	→	h	o	l	a	0	
	2	→	m	u	n	d	o	0
	3	→	a	r	g	s	0	

iA la terminal!





**Qué podemos
hacer**



{ **string.h**
stdlib.h **}**

Comparar

```
int strcmp( char* izq, char* dch);
```

Compara lexicográficamente dos cadenas

Si el argumento es igual a algo que indiquemos

Copiar

```
char* strcpy( char *dst, char *src);
```

Copia la cadena `src` en `dst`, con el terminador en el largo de `src`.

Convertir

```
int atoi( char *str );
```

Convierte la cadena a un int

Convertir

```
long strtol( char *str, char **str_end, int base );
```

Convierte a int el primer número en la cadena `str` en un número con la base indicada.

iA la terminal!





¿Preguntas?

constante

adj. Dicho de cosa: Que se mantiene invariable.

Aplica a variables y argumentos

```
const int VALOR = 5;  
VALOR = 10; // Error: VALOR es constante
```

En argumentos es *particularmente* útil

```
int largo_seguro(const int capacidad, char *cadena);
```

Para aquellos parámetros que no deben cambiar accidentalmente y no darle *más de un rol* a los mismos.



¿Preguntas?

const + punteros

en funciones

Puntero a dato constante

```
const int *ptr;
```

El valor al que apunta el puntero no puede ser modificado.

Puntero constante

```
int * const ptr;
```

El puntero no puede apuntar a otra dirección de memoria

Puntero constante a un dato constante

```
const int * const ptr;
```

Ni el valor al que apunta ni la dirección pueden modificarse.

Un ejemplo

```
// Dado un valor de tipo int
int valor = 42;
// no se puede modificar el valor a través de ptr1
const int *ptr1 = &valor;
// ptr2 no puede apuntar a otro lugar
int * const ptr2 = &valor;
// ni el puntero ni el valor pueden cambiar
const int * const ptr3 = &valor;
```



¿Preguntas?

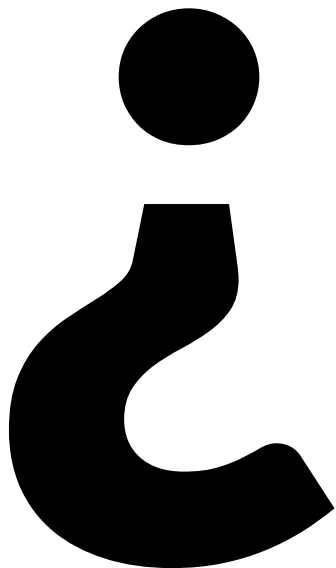
Estructuras de información (structs)

Declaración

```
struct identificador
{
    tipo identificador_miembro_1;
    tipo identificador_miembro_2;
    ...
    tipo identificador_miembro_n;
};
```

Declaración

```
struct fraccion  
{  
    int numerador;  
    int denominador;  
};
```

**Para que se
usan**



Inicialización

```
struct fraccion f2 = {1, 4};  
struct fraccion f2 = {0};
```

Inicialización

(más explícita)

```
struct fraccion f3 = { .denominador = 1, .numerador = 3 };
```

Uso y acceso a miembros

```
struct fraccion f1;  
f1.numerador = 10;  
f1.denominador = 10;
```



Otro ejemplo

```
struct Persona
{
    char nombre[50];
    int edad;
    float altura;
};
```

```
struct Persona persona1 = {"Juan",
30, 1.75};
```

```
struct Persona persona2;
strcpy(persona2.nombre, "Ana");
persona2.edad = 25;
persona2.altura = 1.60;
```

Se pueden anidar!

```
struct Fecha
{
    int dia;
    int mes;
    int año;
};

struct Persona
{
    char nombre[50];
    int edad;
    struct Fecha nacimiento;
};
```

```
struct Persona persona1 = {"Juan",
30, 1.75};
```

```
struct Persona persona2;
strcpy(persona2.nombre, "Ana");
persona2.edad = 25;
persona2.nacimiento = {1, 2, 2000};
persona2.nacimiento.dia = 2;
```



¿Preguntas?

Operaciones con structs

Copia de estructuras

```
struct fraccion f1 = {1, 4};  
struct fraccion f2 = f1;
```

¿Comparación directa?

f2 == f1

no funciona

Miembro a miembro sí

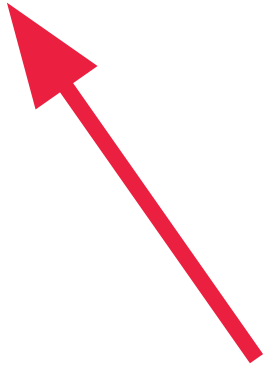
f1.denominador > f2.denominador



¿Preguntas?

Uso en funciones

```
struct fraccion suma_fraccion(struct fraccion frac, int numero);
```



Con esto en mente...

UNRN

Universidad Nacional
de Río Negro

```
struct division {  
    int cociente;  
    int resto;  
};
```

```
struct division division_lenta(int dividendo, int divisor);
```




¿Preguntas?

Sobrenombres (alias)



```
typedef struct fraccion fraccion_t;
```

```
fraccion_t f3;
```

Declaración y alias

```
typedef struct  
{  
    int cociente;  
    int resto;  
} division_t;
```

1

**Un tipo con
sobrenombre tiene
el sufijo
_t**

6

1

**Usen structs
anónimos siempre
que sea posible**

7



¿Preguntas?

```
fraccion_t multiple[MAX];
```




¿Preguntas?

punteros en estructuras

UNRN

Universidad Nacional
de Río Negro

Dada la siguiente estructura

```
typedef struct Cadena{  
    int capacidad;  
    char* cadena;  
}cadena_t;
```

```
cadena_t* crear(int capacidad)
```

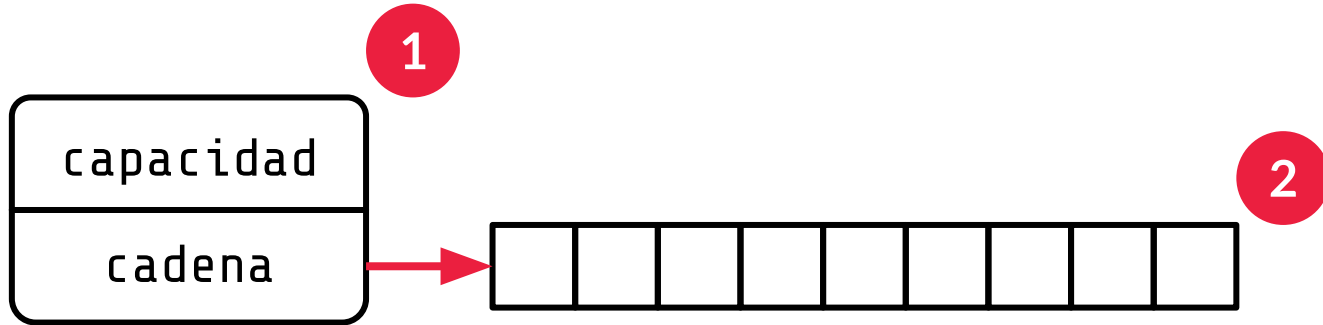


iA la terminal!



—

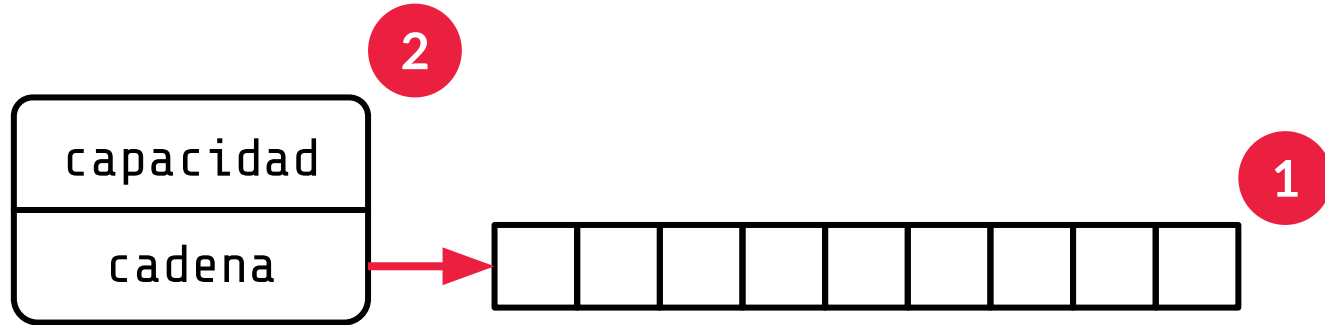
```
cadena_t* crear(int capacidad)
```



Con un malloc para cada parte

**Y para liberar al mismo
nivel**

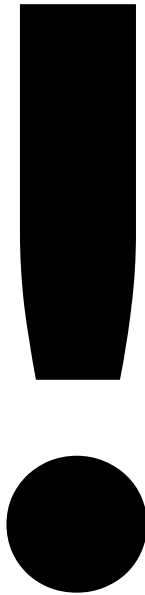
—
`void destruir(cadena_t* cadena)`



Al revés que se crea



Si no usaron la constante
**todo lo de
cadena segura
funciona igual**



La documentación para crear

```
/**
 * Reserva la memoria para una cadena segura
 * @param capacidad el tamaño el arreglo para
 *                 almacenar los caracteres
 * @returns una cadena segura creada dinámicamente
 * @pre capacidad es mayor a 1
 * @post La estructura y arreglo de caracteres creados
 *       dinámicamente
 * @post El arreglo se garantiza en \0 en toda su extensión
 * @nota Usar la función 'destruir' para liberar la memoria
 */
cadena_t* crear(int capacidad);
```



¿Preguntas?

punteros a estructuras

UNRN

Universidad Nacional
de Río Negro

Dada esta estructura, como implementamos 'crear'

```
struct persona
{
    char nombre*;
    int edad;
}

struct persona* crear(char nombre[], int edad)
```

iA la terminal!




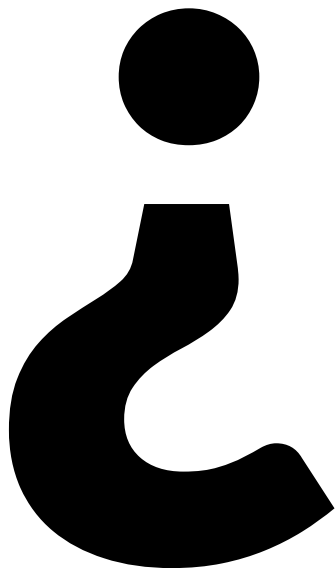
Resuelto

```
persona_t* crear(char nombre[], int edad)
{
    persona_t* nuevo = malloc(sizeof(persona_t));
    if (nuevo != NULL)
    {
        nuevo->nombre = malloc(sizeof(char)*strlen(nombre)+1);
        strcpy(nuevo->nombre, nombre);
        nuevo->edad = edad;
    }
    return nuevo;
}
```

¿Que son estas dos cosas?

```
persona_t* crear(char nombre[], int edad)
{
    persona_t* nuevo = malloc(sizeof(persona_t));
    if (nuevo != NULL)
    {
        nuevo->nombre = malloc(sizeof(char)*strlen(nombre)+1);
        strcpy(nuevo->nombre, nombre);
        nuevo->edad = edad;
    }
    return nuevo;
}
```

Two red arrows originate from the bottom right of the slide. One arrow points diagonally upwards and to the left, terminating at the line 'nuevo->edad = edad;'. The other arrow points diagonally upwards and to the left, terminating at the line 'nuevo->nombre = malloc(sizeof(char)*strlen(nombre)+1);'.



Que es la

->




```
t_persona* usr;
```

usr  nombre

es igual a

usr .nombre

La flecha desreferencia el puntero



**Que falta
ahora**



Para liberar al mismo nivel

```
void destruir(struct persona* persona)
```

iA la terminal!

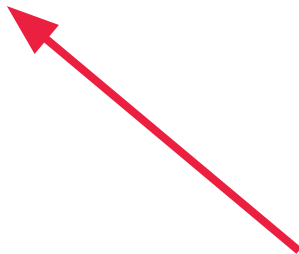


¡De dentro hacia afuera!

```
void destruir (persona_t* persona)
{
    free(persona->nombre);
    free(personal);
}
```

¿Por que en este orden?

```
void destruir (persona_t* persona)
{
    free(persona->nombre);
    free(personal);
}
```





**Más algunas
operacione**

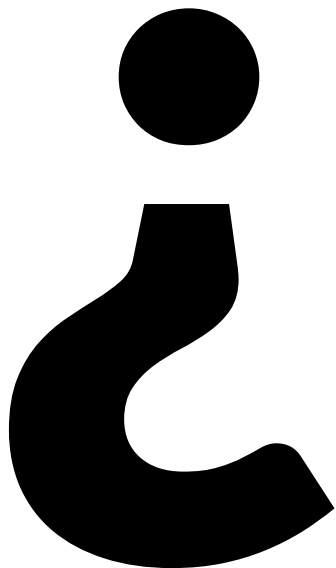
Operaciones interesantes

```
int modificar_edad(persona_t* persona, int nueva_edad)  
int comparar_por_edad(persona_t* p1, persona_t* p2)
```

y más



¿Preguntas?



**Que espacio
ocupa un
struct en
memoria**



Lógicamente, la suma de sus miembros

Fraccio n	numerador	denominador
----------------------	-----------	-------------

Sin importar que sea

Fecha	día	mes	año
--------------	------------	------------	------------

Persona	char[50] Nombre	edad	Fecha nacimiento		
			día	mes	año



¿Preguntas?

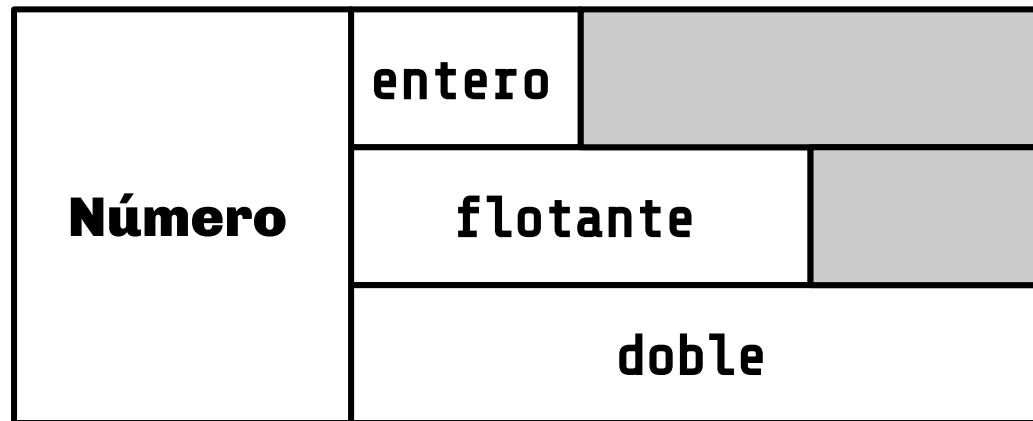
Uniones

Declaración

```
union identificador {  
    tipo identificador_miembro_1;  
    tipo identificador_miembro_2;  
    ...  
    tipo identificador_miembro_n;  
};
```

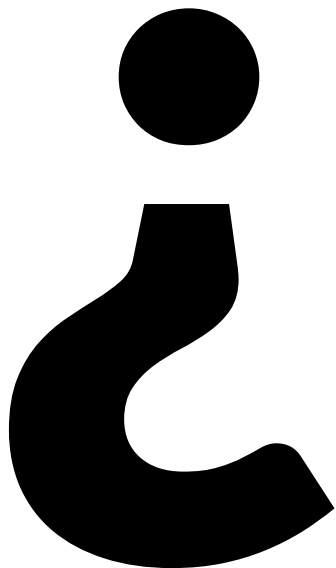

El tamaño es del más grande, el **double** en este caso

```
union Numero  
{  
    int entero;  
    float flotante;  
    double doble;  
};
```



Permite recibir una "figura" sin importar cual sea

```
union figura {  
    struct {  
        float radio;  
    } circulo;  
    struct {  
        float base, altura;  
    } rectangulo;  
    struct {  
        float lado1, lado2, lado3;  
    } triangulo;  
};
```



**Podemos
saber que
vino**



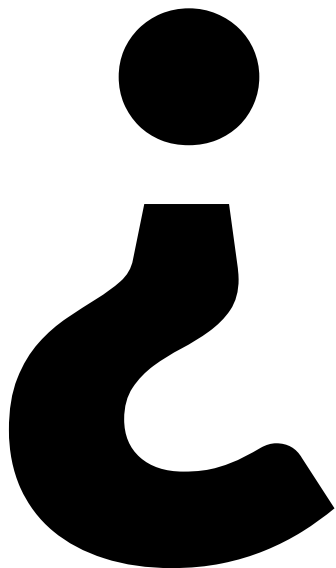
i

nope

!

Por eso se los combina con `struct`s

```
struct figura_geometrica
{
    int tipo; // 0: círculo, 1: rectángulo, 2: triángulo
    union figura datos;
};
```



**Para que se
usan**



Podemos acceder fácilmente a cada byte en un word

```
union ColorRGBA
{
    struct bytes{
        uint8_t rojo;
        uint8_t verde;
        uint8_t azul;
        uint8_t transparencia;
    };
    uint32_t hexadecimal;
};
```

Y podemos

0A	FA	34	12
----	----	----	----

```
union ColorRGBA color = {0};  
color.bytes.rojo = 0xA;  
color.bytes.verde = 0xFA;  
color.bytes.azul = 0x34;  
color.bytes.transparencia = 0x12;
```

```
color.hexadecimal => 0x0AFA3412;
```


**Se usan mucho en
 μ controladores**



¿Preguntas?

enumeraciones

Definición

```
enum nombre_enum {  
    constante1, //0  
    constante2, //1  
    constante3, //2  
    // ...  
};
```

Uso

```
enum dias_semana {  
    LUNES,  
    MARTES,  
    MIERCOLES,  
    JUEVES,  
    VIERNES,  
    SABADO,  
    DOMINGO  
};
```

```
enum dias_semana hoy = MARTES;  
if (hoy == VIERNES)  
{  
    printf("¡Es fin de semana!\n");  
}  
  
int dia_numerico = DOMINGO;
```

Uso

```
enum dias_semana {  
    LUNES = 1,  
    MARTES,  
    MIERCOLES,  
    JUEVES,  
    VIERNES,  
    SABADO,  
    DOMINGO  
};
```

```
enum dias_semana hoy = MARTES;  
if (hoy == VIERNES)  
{  
    printf("¡Es fin de semana!\n");  
}  
  
int dia_numerico = DOMINGO;  
    // DOMINGO es ahora 7
```

¡Como retorno de función!

```
typedef enum {  
    EXITO,  
    ERROR_ARCHIVO_NO_ENCONTRADO,  
    ERROR_MEMORIA_INSUFICIENTE,  
    // ... otros posibles errores  
} t_estado_archivo;
```

```
t_estado_archivo abrir_archivo(const char *nombre_archivo);
```

1

6

**Un struct o union
va en CamelCase**

23/
9



¿Preguntas?

Tipos de datos abstractos

introducción

Abstracción

Se enfoca en el "qué" hace un dato, no en el "cómo" está implementado.

Encapsulación

Ocultar los detalles de la implementación, protegiendo los datos de accesos no autorizados.

Modularidad

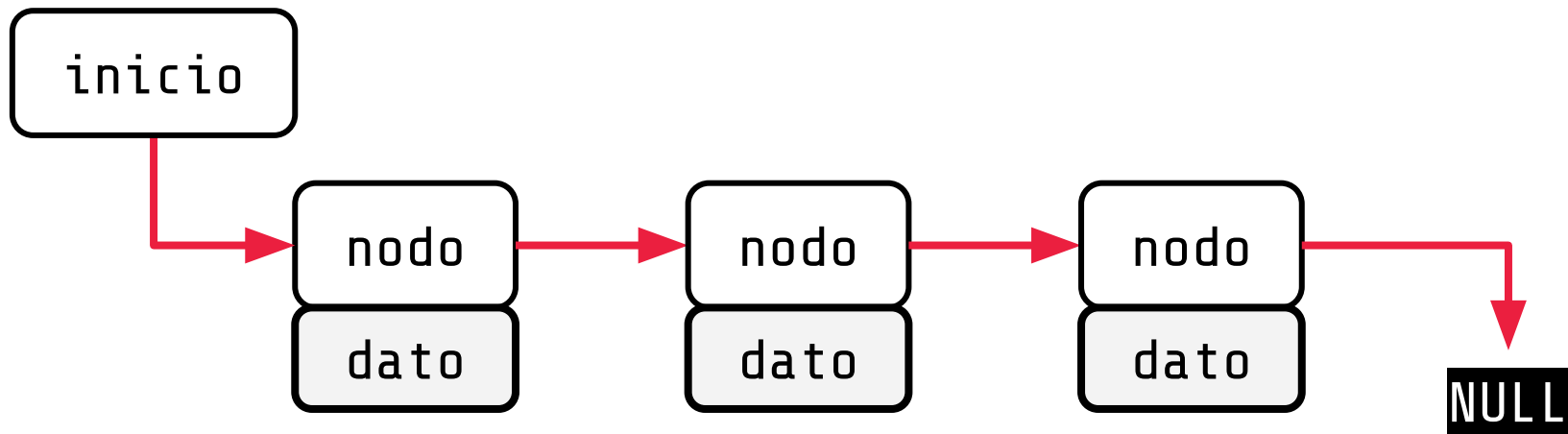
Divide el problema en partes más pequeñas y manejables

Reutilización

Los TDAs pueden ser reutilizados en diferentes partes de un programa o incluso en otros programas.

Introducción a estructuras de datos

Lista enlazada

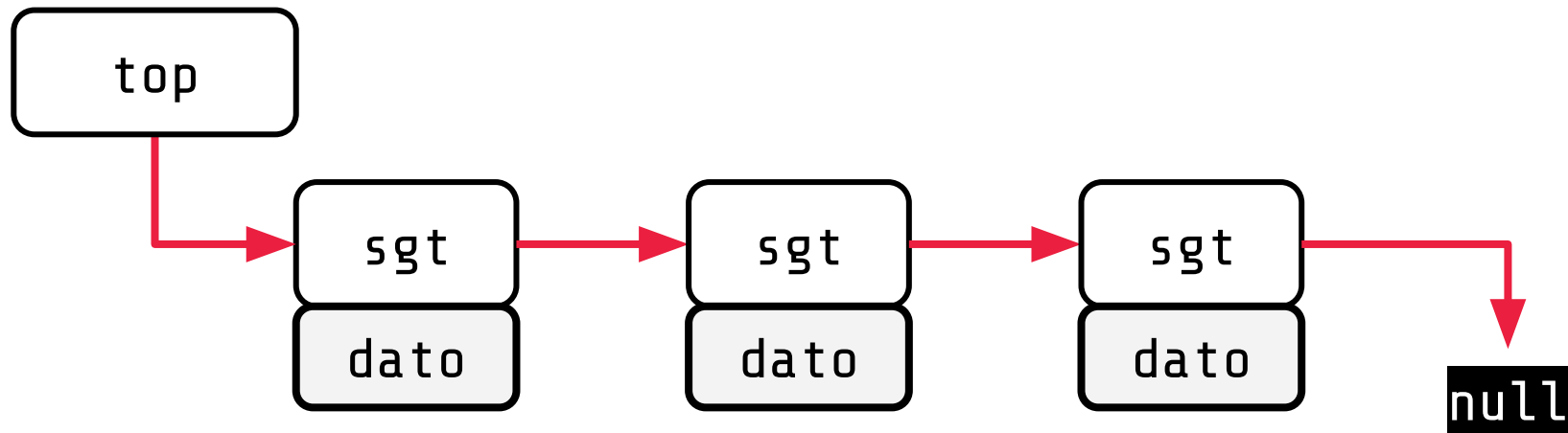


Son la base de otras estructuras

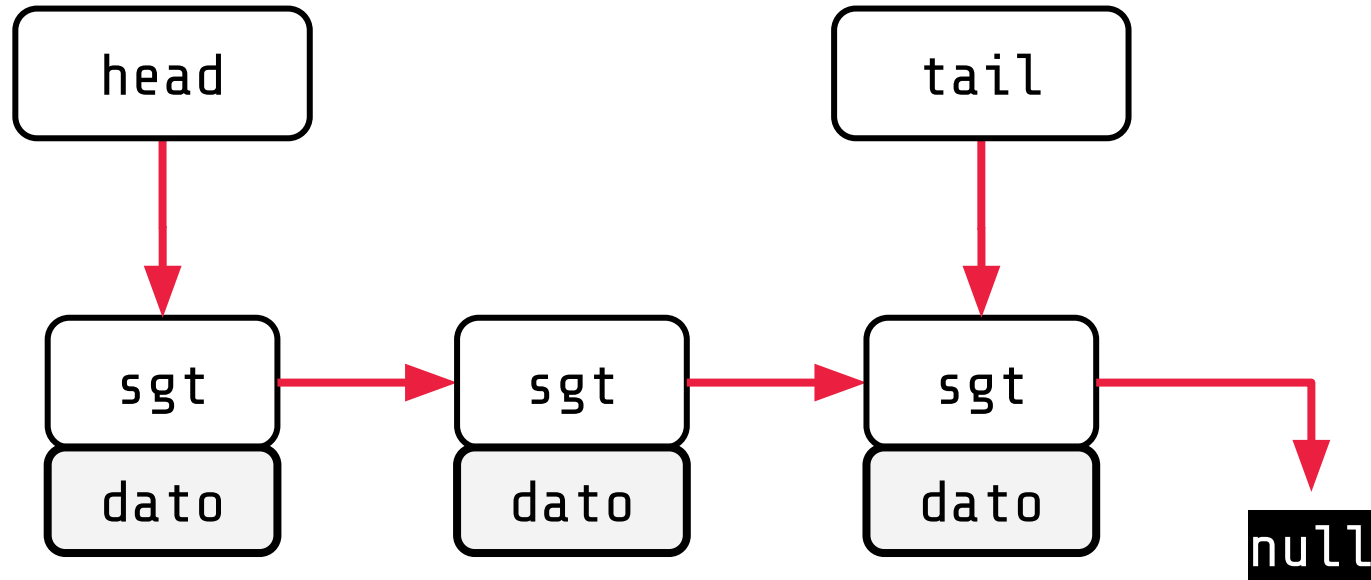
UNRN

Universidad Nacional
de Río Negro

Pila (stack)



Cola (queue)

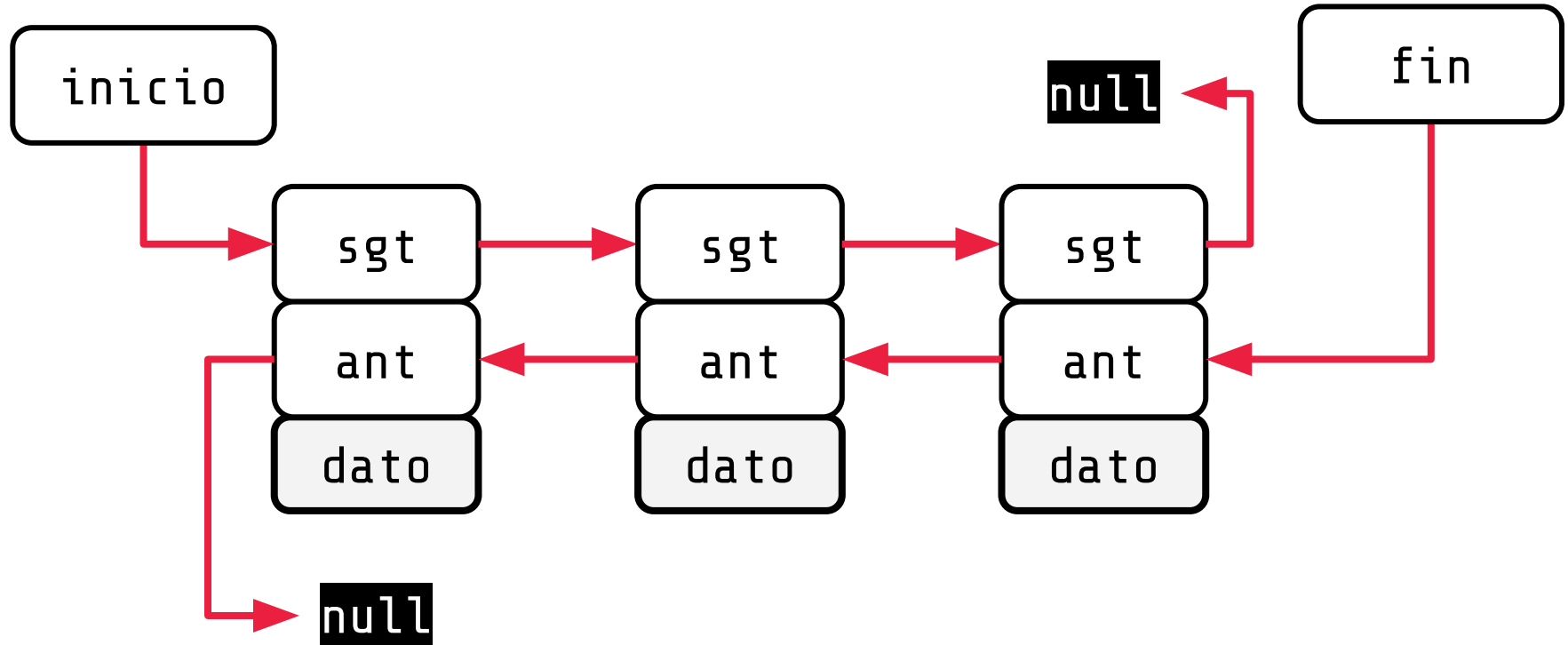


Pero tambien otras mas avanzadas

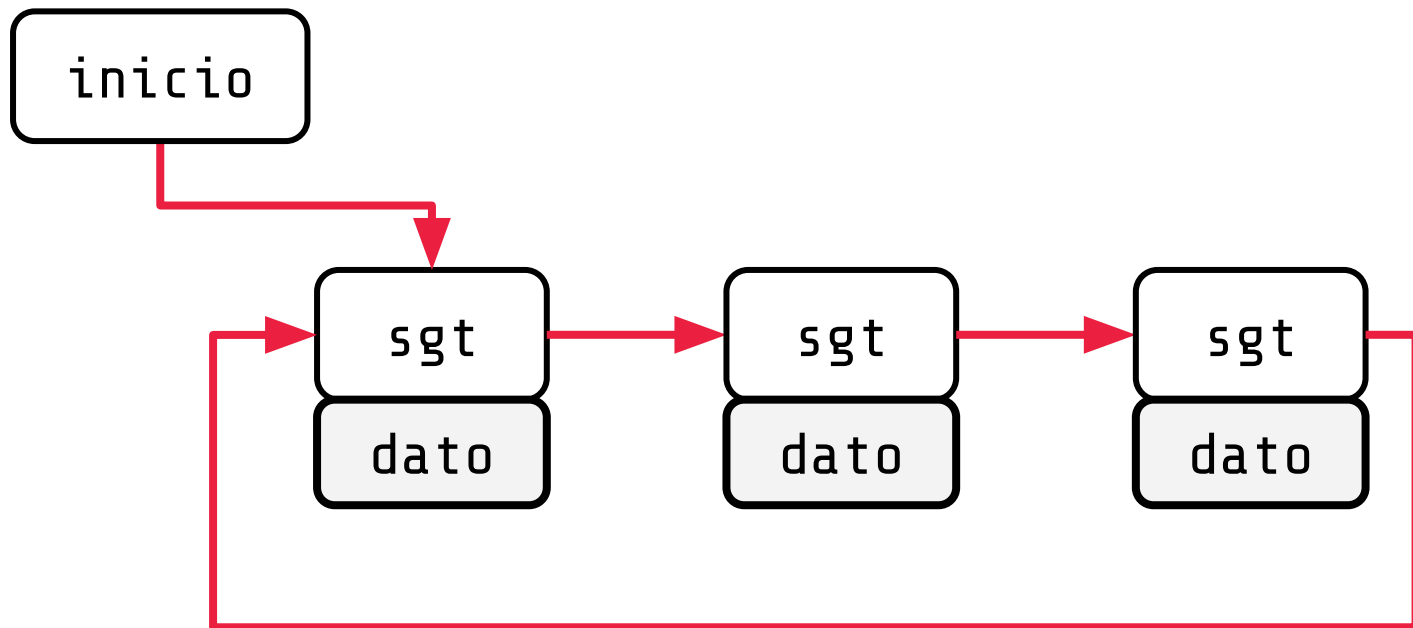
UNRN

Universidad Nacional
de Río Negro

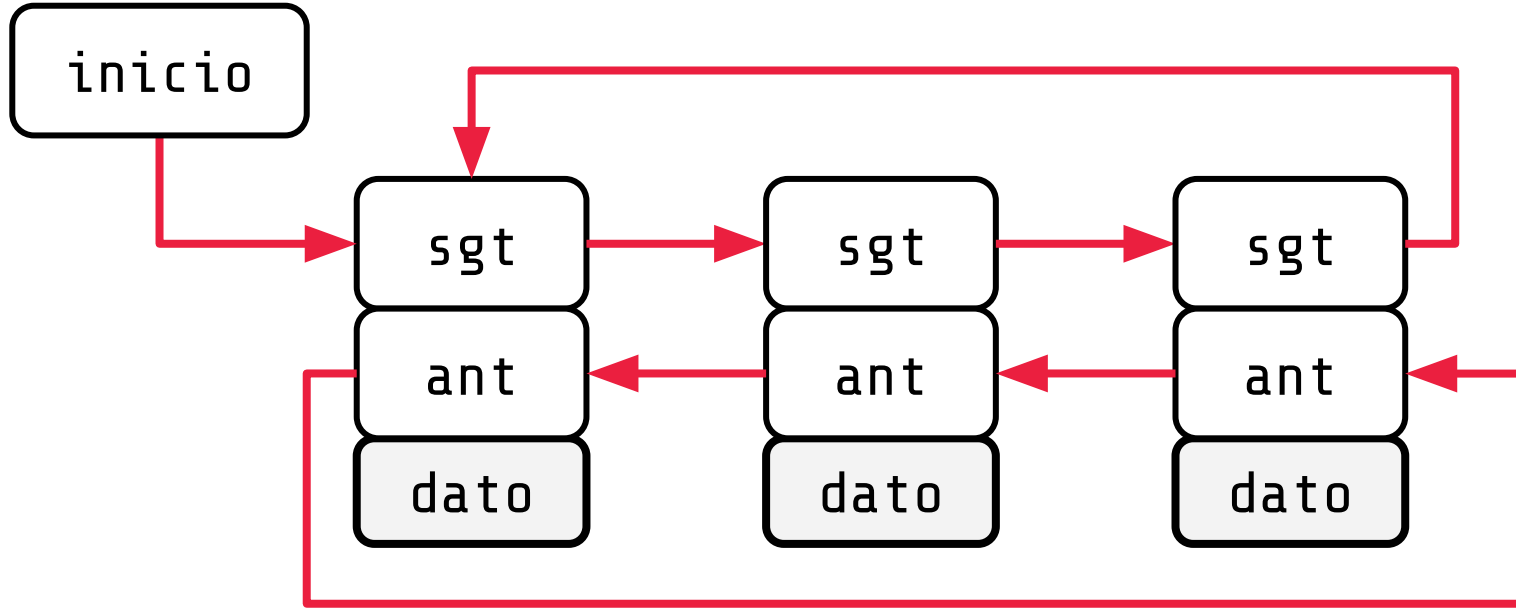
Listas doblemente enlazadas



Lista enlazada circular



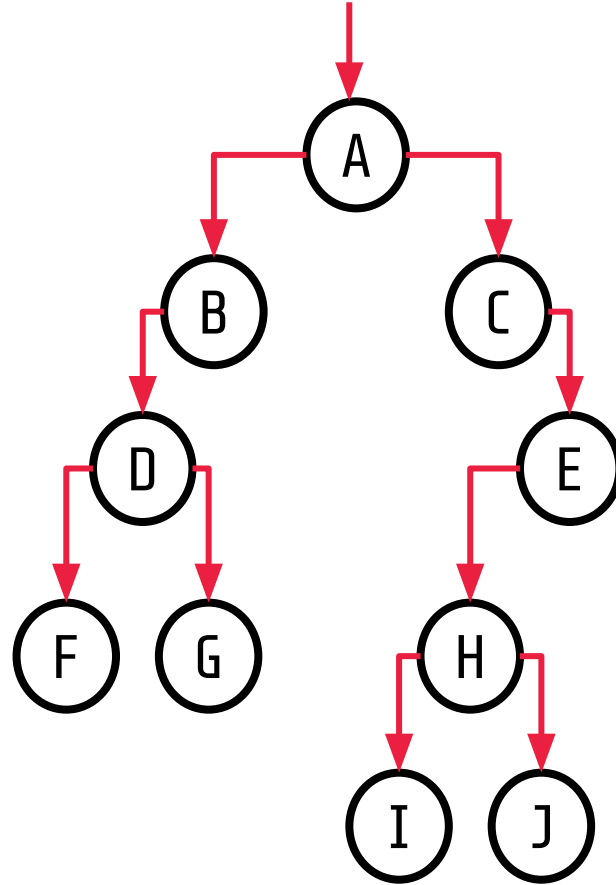
Lista enlazada doble circular



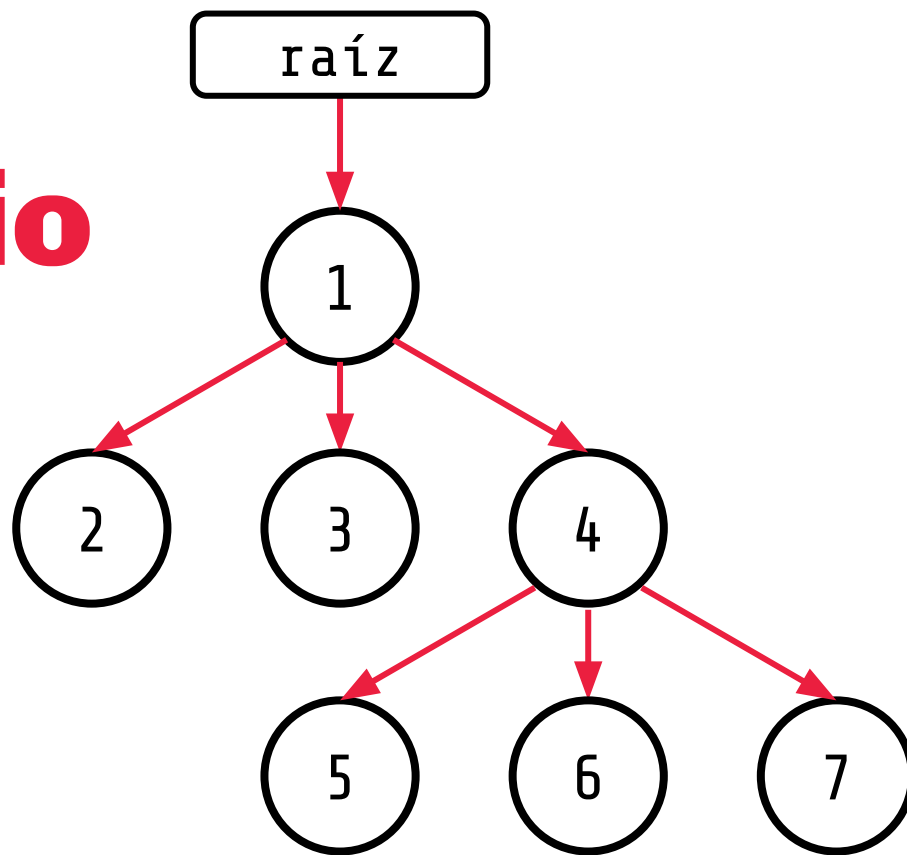
—

Pero no tienen por que ser lineales!

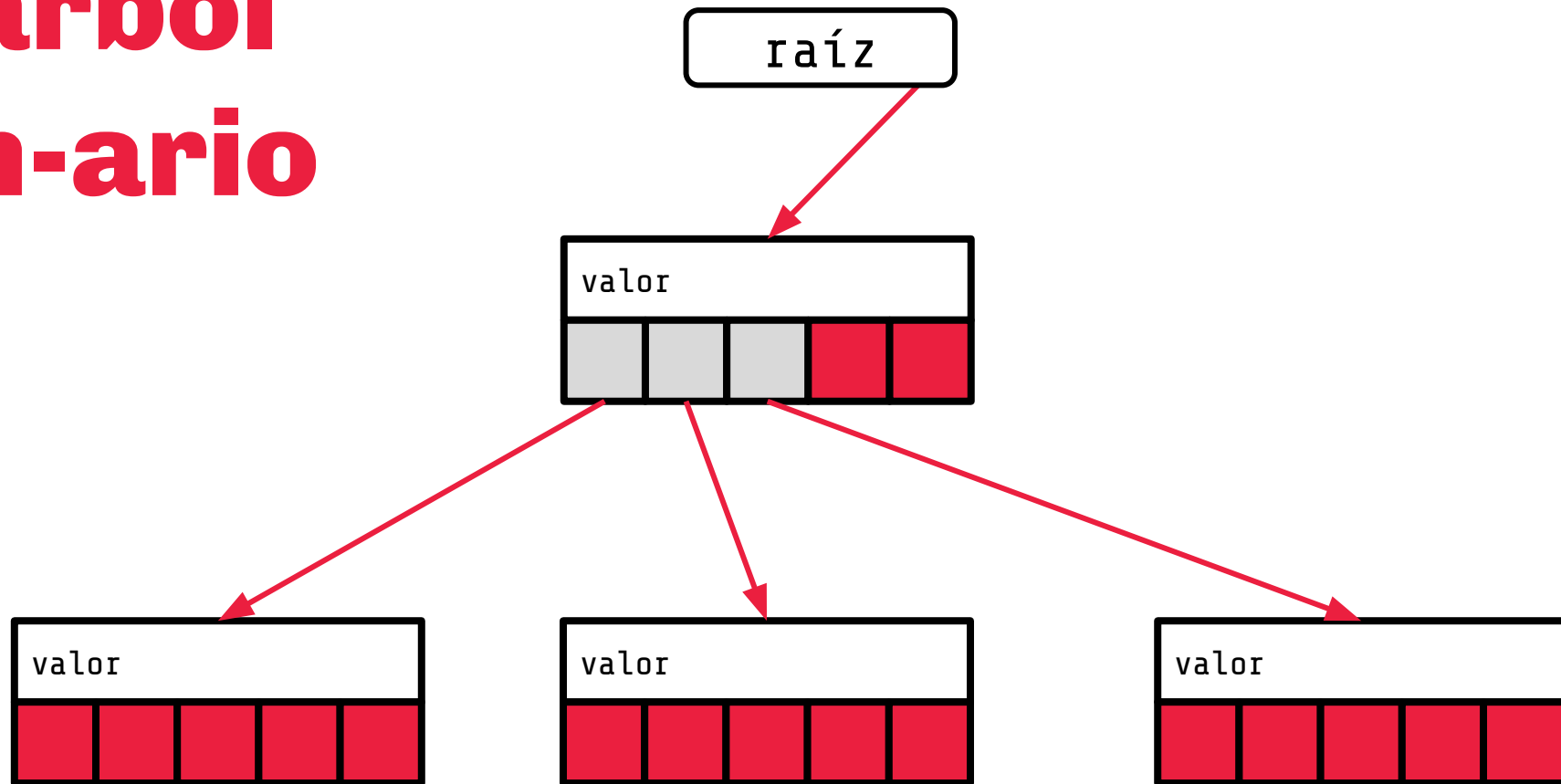
árboles binarios



árbol ternario



árbol n-ario



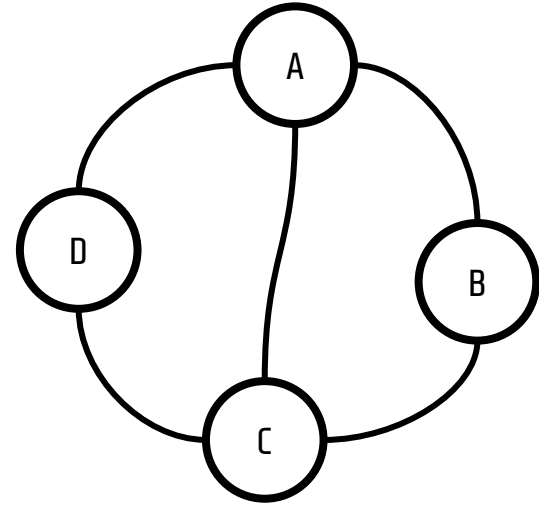
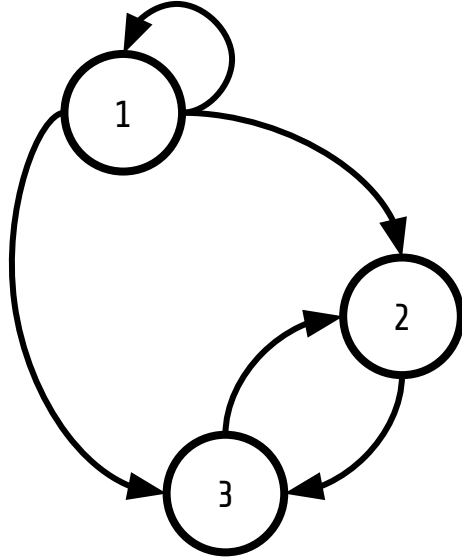
—

O pueden ir en cualquier dirección

UNRN

Universidad Nacional
de Río Negro

Grafos



unrn.edu.ar

UNRN

Universidad Nacional
de **Río Negro**



| **unrionegro**